Ensuring the Semantic Correctness of a BAUML Artifact-centric BPM

Montserrat Estañol[a,*], Maria-Ribera Sancho[a,b], Ernest Teniente[a]

[a] *Department of Service and Information Systems Engineering*
Universitat Politècnica de Catalunya, Barcelona, Spain
[b] *Barcelona Supercomputing Center, Barcelona, Spain*

Abstract

Context: Using models to represent business processes provides several advantages, such as facilitating the communication between the stakeholders or being able to check the correctness of the processes before their implementation. In contrast to traditional process modeling approaches, the artifact-centric approach treats data as a key element of the process, also considering the tasks or activities that are performed in it.

Objective: This paper presents a way to verify and validate the semantic correctness of an artifact-centric business process model defined using a combination of UML and OCL models - a BAUML model.

Method: We achieve our goal by presenting several algorithms that encode the initial models into first-order logic, which then allows to use an existing satisfiability checking tool to determine their correctness.

Results: An approach to verify and validate an artifact-centric BPM specified in BAUML, which uses a combination of UML and OCL models. To do this, we provide a method to translate all BAUML components into a set of logic formulas. The result of this translation ensures that the only changes allowed are those specified in the model, and that those changes are taking place according the order established by the model. Having obtained this logic representation, these models can be validated by any existing reasoning method able to deal with negation of derived predicates. Moreover, we show how to automatically generate the relevant tests to validate the models. We also show the feasibility of our approach by implementing a prototype tool and applying it to a running example.

Conclusion: It is feasible to ensure the semantic correctness of an artifact-centric business process model in practice.

Keywords: verification, validation, reasoning, tool, business process modelling, uml

1. Introduction

Representing business processes using models has several advantages, such as improving communication between the parties involved in the process or having a reference model to which real executions of the process can be compared to. Moreover, with models it is possible to check their correctness before business processes are deployed. Detecting these errors in the early stages of the process definition will help to avoid the cost of later correction, when the process is already running.

There are different types of tests that can be performed to detect these errors. For example, syntactic tests would ensure that the language used to represent the business process is used correctly and structural tests would find errors such as lack of synchronization or deadlocks caused by errors in the flow. Note that these tests only consider the structure of the model and do not deal with additional elements such as the *actual* changes performed by the tasks in the process.

In contrast, semantic tests take into consideration the *meaning* of the different tasks that are carried out by the process. Therefore, they can provide the stakeholders with valuable information in terms of what the

*Corresponding author
Email addresses: estanyol@essi.upc.edu (Montserrat Estañol), ribera@essi.upc.edu (Maria-Ribera Sancho), teniente@essi.upc.edu (Ernest Teniente)

business process is going to do versus what they expect it to do. However, to be able to define exactly what each of the tasks is doing, the model will require a representation of the underlying data.

It is hard to perform semantic tests using the traditional process-centric approach to process modeling, because the resulting models lack the definition of the data and, in consequence, of the tasks in the process. On the other hand, the artifact-centric approach defines both the structure of the data required by the process and the precise meaning of the tasks that make it up, which makes it possible to perform semantic reasoning on these models. This is why we choose an artifact-centric approach in this work.

There are many different ways to represent business processes from an artifact-centric perspective. We follow here the BAUML framework, which uses a combination of UML and OCL models [1], which are OMG and ISO standards. Looking at the diagrams in Figures 1, 2 and 3, and at OCL expressions in Section 2.4, we see that the model provides lots of information and it can be daunting to determine whether there are any errors. For example, *can there be instances of class RejectedSub?* or *can we really execute task Assign to Session?*.

Most of the existing works that deal with the semantic correctness of an artifact-centric BPM require models grounded on logic as input, a language which is more complex and less intuitive for business modelers than using a graphical and standard notation. Moreover, many of these approaches have been formalized theoretically and there are no tools or prototypes that show the feasibility of the approach in question.

Bearing this in mind, our first contribution is the formalization of an approach to ensure the semantic correctness of an artifact-centric BAUML model. This is achieved by automatically encoding the model into a logic formalization that allows performing semantic reasoning by using satisfiability checking techniques able to deal with negation of derived predicates.

Another contribution of this paper is to present a series of relevant, semantic, tests that can be automatically generated from the input models. We first introduce them intuitively and, after presenting the translation process required to treat our problem as a satisfiability test, formalize them accordingly. In this way, the user does not have to worry about which properties to check to ensure that the models are correct.

Finally, we show that all of this is feasible in practice by implementing a prototype tool and applying it to our running example. This is done transparently from the point of view of the user: given a BAUML model and a selected test, the prototype just informs the user of whether the test is satisfiable or not; and he does not have to worry about the translation nor about the required properties to test.

The work we present here extends our previous work in [2] in two main ways. First of all, we introduce and formalize several properties which can be checked over the initial models. We classify these properties according to whether they check the internal correctness of the model or its external correctness. Secondly, we show the feasibility of our approach through the development of a prototype tool. The prototype not only implements the translation process presented in [2] but is also able to automatically generate some of the tests presented here, and which we did not consider before. All in all, this paper provides a deeper insight into reasoning about BAUML models since it matures and further develops the ideas outlined in [2].

2. BAUML in a Nutshell

BAUML is based on the BALSA framework [3], which establishes four dimensions that should always be present in artifact-centric process models:

- **Business Artifacts:** They represent the data required by the business, whose evolution we wish to track. Each artifact has an identifier and may be related to other artifacts.
- **Lifecycles:** They are used to specify the evolution of an artifact during its life, from the moment it is created until it is destroyed.
- **Associations:** They establish the execution flow for services.
- **Services (aka tasks):** They are atomic units of work in the business process. As such, they make changes to artifacts by creating, updating and deleting them.

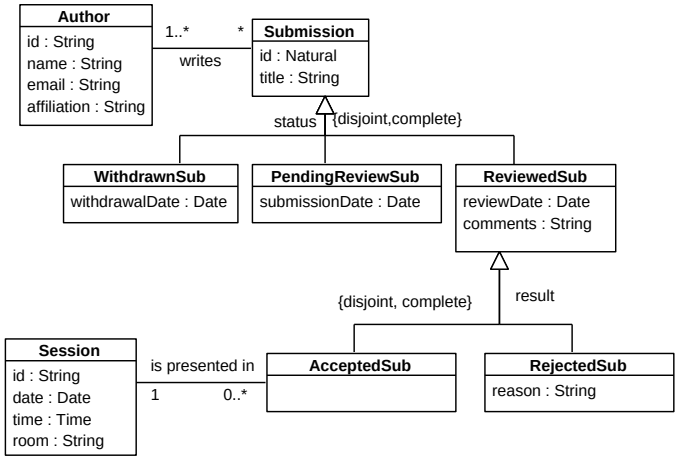

Figure 1: Class diagram for a conference review system.

Businesses also need to keep data whose evolution does not result in relevant states from the point of view of the business. To distinguish this data from artifacts, we will refer to it as *objects*.

The BAUML modeling approach [4, 5] represents the BALSA dimensions using UML and OCL: UML class diagrams for business artifacts; UML state machine diagrams for lifecycles; UML activity diagrams for associations, and OCL operation contracts for services.

2.1. Class Diagram

The class diagram in Figure 1 shows the objects and artifacts in our example. In this case, there is only one artifact: *Submission*. A *Submission* may be a *WithdrawnSub*, if the authors decide to withdraw it; a *PendingRevSub*, while it is waiting to be reviewed; or a *ReviewedSub*, if it has already been reviewed. A *ReviewedSub* will either be an *AcceptedSub*, if it has been accepted for presentation, or a *RejectedSub*.

Each *Submission* is related to a least one *Author*, and an *AcceptedSub* is related to one *Session*, where it will be presented. Both *Author* and *Session* correspond to objects, as their evolution does not result in a specific change of state for the business.

The class diagram is complemented by several OCL constraints which cannot be graphically represented in the class diagram to state that *Submissions*, *Authors* and *Sessions* are identified by their id:

context Submission **inv**: Submission.allInstances()−>isUnique(id)

context Author **inv**: Author.allInstances()−>isUnique(id)

context Session **inv**: Session. allInstances ()−>isUnique(id)

2.2. State Machine Diagram

Figure 2 shows the state machine diagram for artifact *Submission*. Note that each state in the state machine diagram corresponds to one of the subclasses of submission. The exception to this is subclass *ReviewedSub*: as it spans a *disjoint* and *complete* hierarchy, all instances of *ReviewedSub* will either be of type *AcceptedSub* or *RejectedSub*, which do appear in the diagram.

When a *Submission* is created it is in state *PendingReviewSub*. From there, more authors can be added to the submission, and it will remain in state *PendingReviewSub*. If the authors decide to withdraw the submission, it will change to state *WithdrawnSub*. Once a submission has been withdrawn, if the authors regret their decision, they will have to create a new submission. Finally, depending on the outcome of *Review Submission*, it will either change its state to *AcceptedSub* (if the paper is accepted) or to *RejectedSub*.

Our approach assumes that all the integrity constraints established by the class diagram are checked at the end of the execution of a transition in the state machine diagram. That is, while a transition executes, an integrity constraint may be violated, but at the end they must be fulfilled.

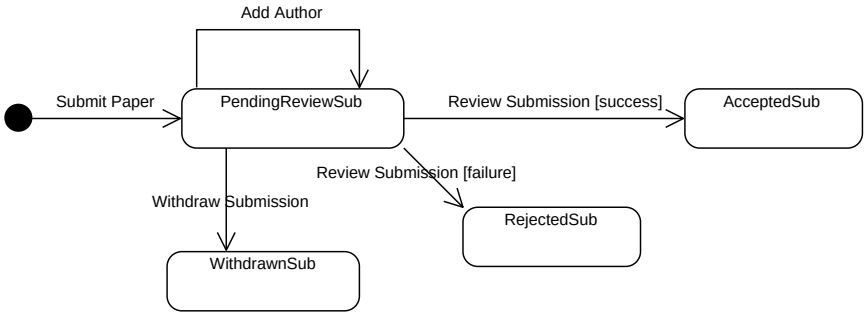

Figure 2: State machine diagram showing the evolution of *Submission*.

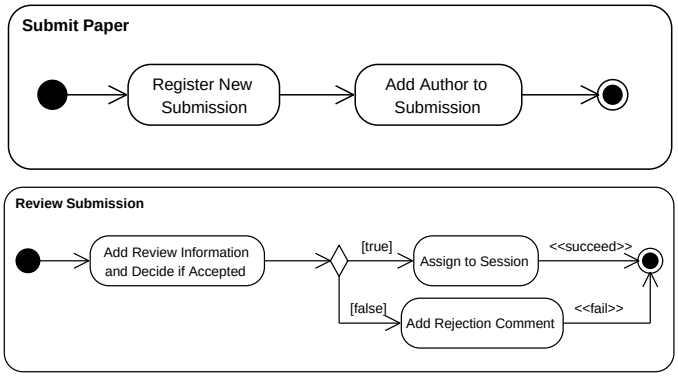

Figure 3: Activity diagram for external event *Submit Paper* and *Review Submission*.

2.3. Activity Diagrams

Each event in the state machine diagram will be specified using a UML activity diagram. Figure 3 shows the details of *SubmitPaper* and *ReviewSubmission*.

SubmitPaper contains only two tasks, *RegisterNewSubmission*, which creates a new instance of *Submission*, and *AddAuthorToSubmission* which is in charge of adding an author to it.

Review Submission contains three different tasks and a decision node. The first task, *Add Review Information and Decide If Accepted* will add some basic information to the review. Then, if reviewers decide to accept the paper, they assign it to a conference session (task *Assign to Session*). Otherwise, they will add a rejection comment (task *Add Rejection Comment*).

2.4. Operation Contracts

For each of the tasks in the activity diagrams, we will have an OCL operation contract which specifies the precise behavior of the task. Below we show the operation contracts for the tasks in *Review Submission*:

```
operation AddReviewInfo(sub:Submission,comms:String)
pre: not(sub.oclIsTypeOf(ReviewedSub))
post: sub.oclIsTypeOf(ReviewedSub) and not (sub.oclIsTypeOf(PendingRevSub)) and
    sub.oclAsType(ReviewedSub).reviewDate=today() and sub.oclAsType(ReviewedSub).comments=comms
```

The first task, *AddReviewInfo*, has as input the submission and the comments. Its precondition ensures that the submission provided as input is not of the *ReviewedSub* type. The postcondition states that the submission will be of the *ReviewedSub* type and assigns values to its attributes: the current date and the comments provided as input.

```
operation AssignToSession(sub:Submission,s:Session)
pre: not sub.oclIsTypeOf(AcceptedSub)
post: sub.oclIsTypeOf(AcceptedSub) and sub.oclAsType(AcceptedSub).session=s
```

Like in the previous task, *AssignToSession* ensures in its precondition that the submission provided as input is not already of the *AcceptedSub* type. The postcondition changes the state of the submission to *AcceptedSub* and relates it to the session given as input.

```
operation AddRejectionReason(sub:Submission, r:String)
pre: not(sub.oclIsTypeOf(RejectedSub))
post: sub.oclIsTypeOf(RejectedSub) and sub.oclAsType(RejectedSub).reason=r
```

Finally, task *AddRejectionReason* changes the state of the submission to *RejectedSub* and assigns the reason r provided as input to its attribute *reason*. Its precondition also ensures that the submission was not already of the *RejectedSub* type.

3. Desirable Properties of a BAUML Model

We distinguish between *verification*, which looks for inherent errors in the model, answering the question *"Is the model right?"*; and *validation* which ensures that the model represents the domain appropriately, answering the question *"Is it the right model?"*. In terms of automation, verification can be performed without user intervention, as it looks for errors and contradictions within the model, while validation requires a user to ensure that reality is represented correctly in the model.

The verification and validation properties that we present in this section are based on or inspired by the following works: [6, 7, 8, 9]. This section is not meant to be an exhaustive list of all the necessary tests to ensure semantic correctness, but rather an illustrative overview of the kind of tests that can be performed over BAUML models.

3.1. Verification

We have classified verification properties according to the dimension of the BAUML model they focus on, although all the dimensions are taken into consideration in the reasoning process.

3.1.1. The Class Diagram in a BAUML Model

Liveliness of the classes and associations. Checking that each class and association in the diagram is lively ensures that there can exist at least one instance of each of the classes and associations. Having a class or an association which cannot be instantiated implies that there is some mistake in the class diagram, as it does not make sense to have an element for which no instances can exist.

Correctness of minimum and maximum cardinalities. Cardinalities in the class diagram may contain errors. In the case of minimum cardinalities, it may be the case that the bound is actually higher than the one stated. The opposite may also hold for maximum cardinalities: the bound may be actually lower than the one that appears in the diagram.

Redundancy of integrity constraints. Although this is not strictly a semantic correctness property by itself, ensuring that the model avoids redundancy and is minimal are also correctness criteria. An integrity constraint is redundant with another when the fulfillment of the first constraint always implies the fulfillment of the second.

3.1.2. The State Machine Diagram in a BAUML model

State reachability. It ensures that every state in which an artifact may be can eventually be reached. As each of the states in the state machine diagram has its corresponding subclass in the class diagram, it is equivalent to checking the liveliness of each of the subclasses of the artifact.

Transition applicability. It ensures that the required conditions are met for an external event or an effect to execute. Note that this property does not ensure that the transition executes successfully, but rather that the conditions can be met for it to begin its execution.

We will consider both external events and effects in the transition as black boxes; therefore, the conditions for applicability will only take into consideration the source state and the OCL condition in the transition, if any. Logically, if the source state is the initial one, then there is no restriction in terms of the source state.

Transition executability. It guarantees that every transition in the state machine diagram can execute successfully. This will happen when the transition itself is applicable and after its execution it leaves the system in a state that fulfills all the integrity constraints.

3.1.3. Activity Diagrams and Operation Contracts in a BAUML Model

Activity diagrams merely establish the order for the execution of the tasks, and it is the tasks themselves the ones that contain the semantic information. Therefore, it suffices to check task-related properties to verify the correctness of activity diagrams.

Precondition Redundancy. A precondition of a task will be redundant if, immediately before the task executes, the precondition is already guaranteed.

Task applicability. A task is applicable if the previous task has executed successfully and its precondition is met. If there is an OCL condition in one of the edges leading to the task, it will also have to be taken into consideration to study its applicability.

Task executability. A task is executable if it is applicable and its postcondition can be met. Note that, since the integrity constraints are not checked until the end of the activity diagram execution, in most cases the tasks will be executable even if they eventually lead to an integrity constraint violation.

3.2. Validation

Validation tests deal with the adequacy of the model in terms of representing the reality appropriately. Therefore, they can be used to check if the business process meets the requirements. We present some tests to detect potential errors, but it is ultimately the modeler's or user's responsibility to interpret the results.

User-defined Tests. Allowing the user to define his or her own tests can be useful to ensure that the model fulfills the requirements elicited in the early stages of the process's definition. This includes the ability to ensure that business rules, which are closely related to business goals [10], have been incorporated correctly in the specification of the business process. For instance, we could have in our example a requirement stating that the emphSession to which an *AcceptedSub* is assigned must have a later *date* than the *reviewDate* of the *AcceptedSub*. A user-defined test would then allow us to check that this property is fulfilled by the model.

Path Inclusion or Exclusion. When there are two different associations which link exactly the same classes, in some cases one relationship should be a subset of the other. In other cases the two paths should be mutually exclusive. Therefore, checking these properties can provide information about a potential error.

Missing irreflexive constraints. For those associations which relate the same class to itself, in many instances there may be an irreflexive constraint missing: that is, one instance of a class cannot be related to itself.

Full transition coverage. The full transition coverage property tests that all possible combinations of transitions, as stated in the state machine diagram, can really take place. If one of these combinations does not execute successfully, then there is the possibility that something is wrong in the definition of the transitions or the external events/actions that make them up.

4. Reasoning about BAUML Models

This section explains how to translate the models and the tests appropriately to be able to specify all tests as satisfiability checking problems. We introduce first the basic concepts and we formalize the models described in Section 2. Afterwards we explain how we translate the BAUML models into first-order logic along the lines of [1]. Finally, we show the formalization of the tests presented in Section 3.

4.1. ***Preliminaries***

For the formalization of our models, we use formulas in first-order logic. A term T is a variable or a constant. If p is a n-ary predicate and $T_1, ..., T_n$ are terms, then $p(T_1, ..., T_n)$ or $p(\overline{T})$ is an atom. An ordinary literal is either an atom or a negated atom. A built-in literal has the form of $A_1 \theta A_2$, where A_1 and A_2 are terms. θ is either $<$, $\leq$, $>$, $\geq$, $=$ or $\neq$.

A normal clause has the form: $A \leftarrow L_1 \wedge ... \wedge L_m$ with $m \geq 0$, where A is an atom and each L_i is an ordinary or built-in literal. All the variables in A, and in each L_i, are assumed to be universally quantified over the whole formula. A is the head and $L_1 \wedge ... \wedge L_m$ is the body of the clause. A normal clause is either a *fact*, $p(\overline{a})$, where $p(\overline{a})$ is a ground atom, or a *deductive rule*, $p(\overline{T}) \leftarrow L_1 \wedge ... \wedge L_m$ with $m \geq 1$, where p is the derived predicate defined by the rule.

A condition is a formula of the (denial) form: $\leftarrow L_1 \wedge ... \wedge L_m$ with $m \geq 1$. Finally, a schema S is a tuple (DR, IC) where DR is a finite set of deductive rules and IC is a finite set of conditions. All formulas are required to be *safe*, i.e. every variable occurring in their head or in negative or built-in literals must also occur in an ordinary positive literal of the same body. An instance of a schema S is a tuple (E, S) where E is a set of facts about base predicates. $DR(E)$ denotes the whole set of ground facts about base and derived predicates that are inferred from an instance (E, S), and corresponds to the fixpoint model of $DR \cup E$.

4.2. ***BAUML Formalization***

This section formalizes the BAUML framework outlined in Section 2 and is structured according to the diagrams that we use for each dimension in the BALSA framework.

4.2.1. Class Diagram and Integrity Constraints

$\mathcal{M}$ is a UML class diagram, in which some classes represent (business) artifacts. Given two classes A and B, we say that A is a B, written $A \sqsubseteq_{\mathcal{M}} B$, if $A = B$ or A is a direct or indirect subclass of B in $\mathcal{M}$. Furthermore, given a class A and a (binary) association R in $\mathcal{M}$, we write $A =_{\mathcal{M}} \exists R$ ($A =_{\mathcal{M}} \exists R^-$ resp.) if A is the domain of R (image of R resp.) according to $\mathcal{M}$. We also denote by $R|_1$ and $R|_2$ the role names attached to the domain and image classes of R. We denote the set of artifacts in $\mathcal{M}$ as ARTIFACTS($\mathcal{M}$) and, when convenient, we use ARTIFACTS($\mathcal{B}$) interchangeably. Each artifact is the top class of a hierarchy whose leaves are subclasses with a dynamic behavior (their instances change from one subclass to another). Each subclass represents a specific state in which an artifact instance can be at a certain moment in time. We denote by A-CLASSES($\mathcal{M}$) (A-CLASSES($\mathcal{B}$) resp.) the set of such subclasses, including the artifacts themselves. These subclasses must fulfill the disjointness constraint (i.e. they must have at most one of the subclasses type at a certain point in time). Given a class $\mathtt{S} \in$ A-CLASSES($\mathcal{M}$), we denote by $\text{ART}_\mathtt{S}$ the class $\mathtt{S}$ itself if $\mathtt{S}$ is an artifact, or the class $\mathtt{A}$ if $\mathtt{A}$ is an artifact and $\mathtt{S}$ is a possibly indirect subclass of $\mathtt{A}$. Given an artifact $\mathtt{A} \in$ ARTIFACTS($\mathcal{M}$), we denote by A-STATES($\mathtt{A}$) the set of leaves in the hierarchy with top class $\mathtt{A}$ if the hierarchy is complete (i.e. every superclass must have one of the subtypes). If the hierarchy is incomplete, A-STATES($\mathtt{A}$) will include the set of leaves and the superclass. We denote the classes in $\mathcal{M}$ as CLASSES($\mathcal{M}$), and the associations in $\mathcal{M}$ as ASSOCIATIONS($\mathcal{M}$). When convenient, we may refer to them as CLASSES($\mathcal{B}$) and ASSOCIATIONS($\mathcal{B}$).

A class diagram will also have a set of graphical and textual integrity constraints. The latter will be represented in OCL. We denote both graphical and textual constraints as $\mathcal{O}$.

4.2.2. State Machine Diagrams

$\mathcal{S}$ is a set of UML state machine diagrams, one per artifact in ARTIFACTS($\mathcal{M}$). More formally, for each artifact $\mathtt{A} \in$ ARTIFACTS($\mathcal{M}$), $\mathcal{S}$ contains a state transition diagram $S_\mathtt{A} = \langle V, v_o, v_f, E, X, T \rangle$, where V is a set of states, $v_o \in V$ is the initial state, $v_f \in V$ is the final state, E is a set of events (either *external* or *time* events), X is a set of effects, and $T \subseteq V \times OCL_{\mathcal{M}} \times E \times C \times X \times V$ is a set of transitions between pairs of states, where $OCL_{\mathcal{M}}$ is an OCL condition over $\mathcal{M}$ that must be true in order for the transition to take place and C is a tag on the result of the execution of the event in E. Note that v_o cannot have any incoming transition, and v_f cannot have any outgoing transition.

The states $V' \subset V$ of $S_{\mathtt{A}}$, such that $V' = V - \{v_o, v_f\}$, exactly mirror the classes in A-STATES($\mathtt{A}$), so that $S_{\mathtt{A}}$ encodes the allowed event-driven transitions of an artifact instance of type `A` from the current state to a new subclass (i.e. a new artifact state). Moreover, the initial transitions starting from v_o always result in the creation of an instance of the artifact being specified by $S_{\mathtt{A}}$.

We distinguish two different kinds of transitions (elements inside parenthesis are optional):

- `([`$OCL_{\mathcal{M}}$`])` `ExternalEvent(`$a_1, ..., a_n$`)` `([`C`])`, where $a_1, ..., a_n$ are the artifacts manipulated by `ExternalEvent`
- `[`$OCL_{\mathcal{M}}$`]` `(/`X`)`

In the first case, the transition will take place if $OCL_{\mathcal{M}}$ is true when the external event is received and the execution of the event results in tag C, if any (its possible values are `success` and `fail`). In the second case, it will take place if $OCL_{\mathcal{M}}$ is true and it will modify the contents of $\mathcal{M}$ as stated by the effect X.

$OCL_{\mathcal{M}}$ is an OCL boolean expression over $\mathcal{M}$, which begins with `self` or `Class.allInstances()->...`, where `Class` may be any $c \in$ CLASSES($\mathcal{M}$). A `TimeEvent` represents an instant of time defined by an expression. This expression may be relative with respect to another point in time or absolute. If it is relative it uses expression `after(time_expression)`; otherwise it uses `at(time_expression)`, as defined in [11]. `ExternalEvent(`$a_1, ..., a_n$`)` must appear at least in a transition of the state machine diagram of each artifact a_i. Given a state machine diagram $S \in \mathcal{S}$, we denote the set of external events in S as EXTEVENTS(S).

The execution of external events and the tags C resulting from this execution are driven by activity diagrams. Each effect X corresponds to an atomic task to be performed when making the transition, and whose parameters are exactly the artifacts involved in the transition. We assume that effects are part of an activity diagram with only one task, which will correspond to the effect itself. This activity diagram will have the same name as the effect plus "AD".

Given an artifact $\mathtt{A} \in$ ARTIFACTS($\mathcal{M}$), we denote by CONDITIONS($\mathtt{A}$) the set of conditions appearing in the state transition diagram $S_{\mathtt{A}}$, also considering all activity diagrams related to $S_{\mathtt{A}}$. We then define CONDITIONS($\mathcal{B}$) $= \bigcup_{\mathtt{A} \in \text{ARTIFACTS}(\mathcal{M})}$ CONDITIONS($\mathtt{A}$).

4.2.3. Activity Diagrams

$\mathcal{P}$ is a set of UML activity diagrams, such that for every state machine diagram $S = \langle V, v_o, v_f, E, X, T \rangle \in \mathcal{S}$, and for every event $\varepsilon \in$ EXTEVENTS(S) there exists exactly one activity diagram $P_\varepsilon \in \mathcal{P}$.

P_ε is a tuple $\langle N, n_o, n_f, F \rangle$, where N is a set of nodes, $n_o \in N$ is the initial node, $n_f \subset N$ is the set of final nodes and $F \subseteq N \times G \times C \times N$ is a set of transitions between pairs of nodes where C is a tag (`success` or `fail`) denoting the correct or incorrect execution of the transition, and G a guard condition.

There are four different types of nodes $n \in N$ in an activity diagram P_ε: initial nodes (denoted as INI(P_ε)), final nodes (FINAL(P_ε)), gateways (GATEWAYS(P_ε)) and activities (ACTIVITIES(P_ε)).

Initial and *final* nodes indicate the points where the activity diagram flow begins and ends. *Gateways* are used to control the sequence flow. We assume that they may only be a *decision node* or a *merge node*.

An *activity* may be an atomic *task* or a *material action*. The set of tasks of an activity diagram p is TASKS(p). Each task is associated to an *operation contract*, as explained in the next subsection. *Material actions* represent physical work done in the process but that does not change the system. The tasks that create artifacts should be the first ones in the activity diagram while those that delete them should be the last ones. This is necessary to ensure the proper tracking of the evolution through the tasks in the activity diagram in the translated model. We assume that all tasks have different names.

We make the following assumptions over each activity diagram P_ε: decision nodes and fork nodes have one incoming flow and more than one outgoing flow; merge nodes and join nodes have more than one incoming flow and exactly one outgoing flow; activities have exactly one incoming and one outgoing flow; initial nodes have no incoming flow and exactly one outgoing flow; and final nodes have one or several incoming flows but no outgoing flow.

We only allow guard conditions over a transition $f = \langle n_s, g, c, n_t \rangle \in F$ if n_s is a decision node. Then, g may correspond to an OCL condition over $\mathcal{M}$ or to a label representing a user-made decision. Similarly, we only allow c over $f \in F$ such that $f = \langle n_s, g, c, n_t \rangle$ and $n_t \in \text{FINAL}(P_\varepsilon)$.

With a slight abuse of notation, given a state machine diagram $S \in \mathcal{S}$, we denote by $\mathcal{P}_S \subseteq \mathcal{P}$ the set of activity diagrams referring to all external events appearing in S.

As we have explained previously, during the execution of an activity diagram the constraints in $\mathcal{O}$ may be violated, as we follow a strict interpretation of operation contracts [12]. However, these must be fulfilled at the end of the execution, otherwise the transition in the state machine diagram does not take place and all the changes made are "rolled back".

4.2.4. Tasks

$\mathcal{T}$ is a set of atomic tasks, each of which has an OCL operation contract. Its semantics is that the task can only be executed when the current information base satisfies its precondition, and that, once executed, the task brings the information base to a new state that satisfies its postcondition. If, during the execution of an activity diagram the precondition of one of the tasks is not met, then we assume that the corresponding transition does not take place and that no changes are made.

Given an artifact $\mathtt{A} \in \mathcal{M}$, we denote by $\text{TASKS}(\mathtt{A})$ the set of tasks appearing in the state machine diagram $S_\mathtt{A}$, also considering all activity diagrams related to $S_\mathtt{A}$. We then define $\text{TASKS}(\mathcal{B}) = \bigcup_{\mathtt{A} \in \text{ARTIFACTS}(\mathcal{M})} \text{TASKS}(\mathtt{A})$. Moreover, we assume that every task in $\text{TASKS}(\mathtt{A})$ that does not belong to the activity diagram of an initial transition has as input an instance of the artifact in S_a.

4.3. ***Encoding a BAUML model into logic***

We must first encode the BAUML model into logic so that satisfiability checking techniques can be used for reasoning. Roughly, this encoding will map the classes and associations to predicates. Those that are read-only, will be base predicates, whereas those that are read-write will be derived from the execution of the tasks that create and/or delete them. These derivation rules will also have to take into consideration the context in which the tasks execute, determined by the state machine diagram and the activity diagrams. Finally, the integrity constraints in the class diagram will be translated as logic formulas in the denial form, considering that they only need to be checked at the end of a transition.

We assume artifact-centric business process models with one artifact type since having two or more artifact types would require tracking simultaneously the evolution of two artifact types adding much more complexity. Moreover, we suppose that the initial BAUML model does not have any material actions in its activity diagrams since they have no impact by themselves on the system. This does not limit our approach since it is straightforward to transform an activity diagram with material actions into one without.

The work we present here clearly differs from [6], where only class diagrams and operation contracts were considered. Note that in this case no restrictions were imposed on the execution of the tasks nor on the checking of the constraints. Therefore, we have had to extend the translation to incorporate the execution order of operations given by activity and state machine diagrams. Moreover, we no longer assume that all classes and associations have to be created by the operations in the model.

Our translation process is divided into four steps, shown in Algorithm 1. To begin with, we focus on the generic steps: obtaining derivation rules for classes and associations, translating the integrity constraints, generating the derivation rules from the tasks, and adding the required conditions to ensure that tasks execute properly, in the context given by state transition and activity diagrams.

The first step creates the derivation rules for the read-write classes and associations. This distinction is made by examining the postcondition of all tasks as described in [6]. The predicate corresponding to each read-write class and association will have a time component t indicating that the element exists at time t, whereas read-only elements will be treated as base predicates without the time t.

The algorithm also takes into consideration if a class is *created* or *created and deleted* in the model. The general form of these rules is:

$$C(\overline{p}, t) \leftarrow addC(\overline{p}, t_1) \wedge \neg deletedC(\overline{p}_j, t_1, t) \wedge t \geq t_1 \wedge time(t),$$

Algorithm 1 TranslateToLogic($\mathcal{B} = \langle \mathcal{M}, \mathcal{O}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$)

▷ Step 1: Creating rules for read/write classes and associations

$r := \emptyset$
for all $c \in$ CLASSES($\mathcal{M}$) **do**
  if c is created in $\mathcal{P} \wedge c$ is not deleted in $\mathcal{P}$ **then**
    $r := r \cup \{C(\overline{p}, t) \leftarrow addC(\overline{p}, t_1) \wedge time(t) \wedge t \geq t_1\}$
  else if c is created in $\mathcal{P} \wedge c$ is deleted in $\mathcal{P}$ **then**
    $r := r \cup \{C(\overline{p}, t) \leftarrow addC(\overline{p}, t_1) \wedge \neg deletedC(\overline{p}_j, t_1, t) \wedge t \geq t_1 \wedge time(t)\}$
    $r := r \cup \{deletedC(\overline{p}_j, t_1, t_2) \leftarrow delC(\overline{p}_j, t) \wedge time(t_1) \wedge time(t_2) \wedge t \leq t_2 \wedge t > t_1\}$
  end if
end for
for all $a \in$ ASSOCIATIONS($\mathcal{M}$) **do**
  if a is created in $\mathcal{P} \wedge a$ is not deleted in $\mathcal{P}$ **then**
    $r := r \cup \{A(\overline{p}, t) \leftarrow addA(\overline{p}, t_1) \wedge time(t) \wedge t \geq t_1\}$
  else if a is created in $\mathcal{P} \wedge a$ is deleted in $\mathcal{P}$ **then**
    $r := r \cup \{A(\overline{p}, t) \leftarrow addA(\overline{p}, t_1) \wedge \neg deletedA(\overline{p}_j, t_1, t) \wedge t \geq t_1 \wedge time(t)\}$
    $r := r \cup \{deletedA(\overline{p}_j, t_1, t_2) \leftarrow delA(\overline{p}_j, t) \wedge time(t_1) \wedge time(t_2) \wedge t \leq t_2 \wedge t > t_1\}$
  end if
end for

▷ Step 2: Translate integrity constraints

$icSet := translateIC(\mathcal{O})$
for all condition $cond \in icSet$ **do**
  $cond := cond + \{\wedge validState(t)\}$
end for
$taskRules := \emptyset$

▷ Step 3: Generate rules for class and association creation and deletion for every task

for all $t \in \mathcal{T}$ **do**
  $resRules := translateTask(t)$
  $taskRules := taskRules \cup resRules$
end for

▷ Step 4: Generate necessary rules and conditions to ensure correct execution order

$taskRules := taskRules \cup generateConstraintsTaskExecution(\mathcal{B})$
return $\langle r, icSet, taskRules \rangle$

where $\overline{p}$ corresponds to the attributes in the class (including its OID [unique object identifier]) or the participants in the association, $\overline{p}_j$ represents the identifier of the class (its OID) or association (OID of the classes that participate and identify it) C, and thus $\overline{p}_j \subseteq \overline{p}$, and t and t_1 represent the time. We will see how $addC(...)$ and $deletedC(...)$ are obtained later on.

The rule basically states that a class or an association will exist at time t if it has been created previously, at t_1 ($t_1 \leq t$), and it has not been deleted in the meantime. For instance, `Submission` is encoded as:

$$Submission(s, i, title, t) \leftarrow addSubmission(s, i, title, t_1) \wedge time(t) \wedge t_1 \leq t \\ \wedge \neg deletedSubmission(s, t_1, t)$$

`Submission` is a derived predicate created and deleted by some of the tasks. On the other hand, `Author` is a base predicate as it is not created nor deleted by any task.

Step 2 of the algorithm translates the constraints $\mathcal{O}$ into a set of formulas in denial form, but we need to add an atom $\wedge validState(t)$ to each of them to ensure that they are only checked at the end of the execution of a state transition diagram transition, following the semantics of the framework.

For instance, the covering constraint in the hierarchy of *Submission* indicates that a *Submission* must have one of its subclasses' type. Then the condition:

$$\leftarrow Submission(s, i, title, t) \wedge \neg IsKindOfSubmission(s, t) \wedge validState(t)$$

states that there cannot be a bicycle which has not any of its subtypes (predicate $IsKindOfSubmission$), where $IsKindOfSubmission$ is a derived predicate from $WithdrawnSub$, $PendingReviewSub$, $ReviewedSub$, $AcceptedSub$, $RejectedSub$ (see below). This condition only applies when there are no transitions taking

place, indicated by predicate *validState*.

$$IsKindOfSubmission(s,t) \leftarrow WithdrawnSub(s,d,title,w,t)$$
$$IsKindOfSubmission(s,t) \leftarrow PendingRevSub(s,i,title,su,t)$$
$$IsKindOfSubmission(s,t) \leftarrow ReviewedSub(s,i,title,r,c,t)$$
$$IsKindOfSubmission(s,t) \leftarrow AcceptedSub(s,i,title,r,c,t)$$
$$IsKindOfSubmission(s,t) \leftarrow RejectedSub(s,i,title,r,c,re,t)$$

Step 3 is the most complex and it is decomposed into various algorithms. It generates the derivation rules that link the creation and deletion of the classes and associations with the tasks that perform these changes, and ensures that all tasks execute at the right time. This is done by calling Algorithms 2 and 3.

Finally, step 4 generates the remaining necessary constraints to ensure the correct execution of the tasks by calling Algorithm 4. For instance, if there is a sequence of tasks that execute in the activity diagram, it ensures that all of them execute and creates the derivation rules to generate predicate *validState* at the end of the execution of the activity diagram.

Algorithm 2 translateTask($task$)

```
rules := ∅
prevRules := getContextPreviousTasks(task, t)        ▷ t represents a time term
createList contains the classes and associations created by task
delList contains the classes and associations deleted by task
for all ruleFragment ∈ prevRules do
    for all el ∈ createList do
        r := addEl(p̄, t) ← task(p̄, x̄, t) ∧ pre_task(t − 1) ∧ time(t) ∧ ruleFragment
        rules := rules ∪ r
    end for
    for all el ∈ delList do
        r := delEl(p̄_j, t) ← task(p̄_j, ȳ, t) ∧ pre_task(t − 1) ∧ time(t) ∧ ruleFragment
        rules := rules ∪ r
    end for
    rules := rules ∪ {task′(p_a, t) ← task(p_a, z̄, t) ∧ pre_task(t − 1) ∧ time(t) ∧ ruleFragment}
end for
return rules
```

We will now analyze the details of the remaining algorithms. Algorithm 2 is aimed at translating the atomic tasks. As they make changes to the instances of the class diagram, this translation will result in the derivation rules that generate predicates *addEl* and *delEl*, where *el* is a class or an association. These rules are generated by analyzing the postcondition of each task and determining if the task creates or deletes some instance. If the task has a precondition, then its translation (following [9]) is also added to the body of the derivation rule to ensure that it is true at time $t-1$, where t represents the time the task executes.

However, this translation does not impose any restrictions over the order for task execution. In BAUML tasks execute following the restrictions and the order established by the state transition and activity diagrams. In particular, $task_k$ can only execute if pre_{task_k} is true and the previous task $task_{k-1}$ has executed at $t-1$.

Algorithm 2 generates the creation and deletion rules as described, invoking Algorithm 3 to obtain the part of the rule that refers to the successful execution of the previous tasks. At the end, Algorithm 2 generates a rule of the form:

$$task'(p_a,t) \leftarrow task(p_a,\overline{z},t) \wedge pre_{task}(t-1) \wedge time(t) \wedge ruleFragment,$$

where p_a corresponds to the OID of the business artifact, which we use to ensure the proper evolution of the system, and $\overline{z}$ corresponds to the remaining parameters or terms of *task*. The derived predicate of this rule, $task'(...)$, will be used as an indicator that *task* has executed properly by the next task.

Algorithm 3 is in charge of generating the part of the derivation rules that depends on the previous node(s) of a certain node. Its complexity lies in the fact that we consider not only linear activity diagrams, but that we also allow decision and merge nodes. We assume that control nodes do not add execution time to our diagrams and that they are traversed immediately. So, given a node n that belongs to an activity diagram P_ε and time t, the algorithm:

Algorithm 3 getContextPreviousTasks(n,t)

```
result := ∅
prevSet contains the previous nodes of n
for all n_p ∈ prevSet do
    if n_p is task then
        result := result ∪ n'_p(p_a, t − 1)
    else if n_p is decision node then
        guard := getGuard(n_p, n)
        res := getContextPreviousTasks(n_p, t)
        for all el ∈ res do
            result := result ∪ {el ∧ guard(t − 1)}
        end for
    else if n_p is merge node then
        res := getContextPreviousTasks(n_p, t)
        result := result ∪ res
    else if n_p is initial node then
        transitions contains the transitions in which the activity diagram appears
        for all t ∈ transitions do
            s_s is the source state of t
            cond is the translation of condition of t
            if s_s is not initial pseudostate ∧ cond is not empty then
                result := result ∪ {s_s(p̄, t − 1) ∧ cond(t − 1)}
            else if s_s is not initial pseudostate then
                result := result ∪ {s_s(p̄, t − 1)
            else if cond is not empty then
                result := result ∪ {cond(t − 1)}
            end if
        end for
    end if
    return result
end for
```

1. Obtains the previous nodes of n, stores them in *prevSet* and initializes *result* to the empty set.
2. For each $n_p \in prevSet$, it checks its type.
   (a) If n_p is a task, it then adds the $n'_p(...)$ predicate to the existing *result*, indicating that the task n_p will have executed successfully.
   (b) If n_p is a decision node, the algorithm needs to obtain the predicates corresponding to the tasks that may execute before n_p; therefore it invokes itself, but this time with n_p and t as input. As n_p is a decision node, there will be a guard condition in the edge between n_p and n. This guard will be translated as if it was a precondition and it will have to be true at $t-1$ in order for the task to execute. Then, it will add the guard condition to each rule-part obtained by the self-invocation.
   (c) If n_p is a merge node, it invokes itself with parameters n_p and t, and it adds the result of this invocation to variable *result*.
   (d) If, on the other hand, n_p is an initial node, it adds the source state of the state transition diagram of the transitions in which the activity diagram appears. If there is an OCL condition, it also adds the translation of the condition.
3. The algorithm returns variable *result*, containing a set of rule fragments.

For instance, for task *Add Rejection Reason*, we have the following rules, among others:

$$\begin{aligned} addRejectedSub(s,i,ti,r,c,st,t) \leftarrow\ & addRejectionReason(s,st,t) \\ & \wedge Submission(s,i,ti,t) \wedge precondAddRejReas(s,t-1) \\ & \wedge Submission(s,id,ti,t-1) \wedge addReviewInfo'(s,t-1) \\ addRejectionReason'(s,t) \leftarrow\ & addRejectionReason(s,st,t) \\ & \wedge Submission(s,i,ti,t) \wedge precondAddRejReas(s,t-1) \\ & \wedge Submission(s,i,ti,t-1) \wedge addReviewInfo'(s,t-1) \end{aligned}$$

The task creates an instance of the *RejectedSub* class. It has a precondition which must be true at $t-1$, and its translation appears in the derivation rule of *addRejectedSub*. In addition to this, the body of the rule includes the predicate $addReviewInfo'$, that guarantees that the previous operation (*Add Review Info*) has executed successfully.

Algorithm 4 generateConstraintsTaskExecution($\mathcal{B}$)

```
constr := ∅
for all task ∈ TASKS(B) do
    n_n is next node of task
    if n_n is task then
        constr := constr ∪ {← task(p_a, z̄, t) ∧ ¬n'_n(p_a, t + 1)}
    else if n_n is decision node ∨ n_n is merge node then
        r :=← task(p_a, z̄, t) ∧ ¬nextTask(p_a, t + 1)
        res := generateConstraintsNextTasks(n, task)
        constr := constr ∪ r ∪ res
    else if n_n is final node then
        constr := {validState(t) ← task'(p_a, t)}
    end if
end for
return constr
```

With the algorithms that we have seen so far we have restricted the order for the tasks execution in one direction, ensuring that task $task_k$ can only execute if $task_{k-1}$ has taken place. We also need to ensure that, once an activity diagram begins execution, it finishes. Algorithm 4 generates the necessary constraints to do so. For each task, it obtains its next node and, if the next node n_n is a task, it creates a rule of the form: $\leftarrow task(p_a, \overline{z}, t) \wedge \neg n'_n(p_a, t+1)$, where predicate n'_n corresponds to the derived predicate generated by Algorithm 2 to ensure that task n_n has executed properly. For instance, for the tasks *Register New Submission* and *Add Author to Submission* we have the following condition and derivation rule:

$$\leftarrow registerNewSubmission(s,t) \wedge \neg(addAuthorToSubmission(s,t+1))$$

On the other hand, if n_n is a decision node or a merge node, there is the possibility that there will be more than one task that can be executed. For this reason, the algorithm generates this rule: $\leftarrow task(p_a, \overline{z}, t) \wedge \neg nextTask(p_a, t+1)$, meaning that if $task$ has executed at t one of its next tasks must have executed at $t+1$. $nextTask$ is a derived predicate resulting from the execution of any of the next tasks. These derivation rules are created in Algorithm 5 and have the following form: $nextTask(p_a, t) \leftarrow task'_n(p_a, t)$. The algorithm iterates over the nodes until the next task(s) are found. Guard conditions are not considered because they have already been translated by the other algorithms.

Finally, if a task is followed by a final node, we need to generate rule: $validState(t) \leftarrow task'(p_a, t)$. This rule will ensure that the restrictions of the model are checked at the end of the execution. For instance, in our example the successful execution of task *Assign To AnchorPoint* generates predicate *validState* as it is the last task in the activity diagram:

$$validState(t) \leftarrow addRejectionReason'(s, st, t).$$

Algorithm 5 generateConstraintsNextTasks(n,task)

```
result := ∅
nextSet contains the set of next nodes of n
for all n_n ∈ nextSet do
    if n_n is task then
        nextTask(p_a, t) ← n'_n(p_a, t)
    else if n_n is decision node ∨ n_n is merge node then
        res := generateConstraintsNextTasks(n_n, task)
        result := result ∪ res
    else if n_n is final node ∧ n is decision node then
        guard contains the guard condition from n to n_n
        nextTask(p_a, t) ← task'(p_a, z̄, t) ∧ guard(ȳ, t)
        validState(t) ← task'(p_a, z̄, t) ∧ guard(ȳ, t)
    end if
end for
return result
```

There is a special case, however. If there is a decision node n and one of the next nodes $n_n \in \text{FINAL}(P_\varepsilon)$

is a final node, then these rules are needed:

$$nextTask(p_a, t) \leftarrow task'(p_a, \overline{z}, t) \wedge guard(\overline{y}, t)$$
$$validState(t) \leftarrow task'(p_a, \overline{z}, t) \wedge guard(\overline{y}, t),$$

which will ensure that after the execution of *task*, the diagram terminates if the guard condition is met.

4.4. ***Differences between UML/OCL and First-Order Logic that Impact the Translation Process***

The translation of UML and OCL into other formal languages can pose several challenges due to the differences in semantics [13, 14]. This subsection analyzes some of these issues.

Graphical Elements and Constraints. Our approach considers only the following graphical elements: classes, association classes, associations, attributes and association multiplicities. This does not include additional graphical restrictions that UML allows, such as the *subset* or *xor*, or the aggregation and composition associations. However, all of these can be expressed as OCL textual constraints which can be translated by our approach.

Redefinition of Operations. As the UML/OCL languages follow an object-oriented paradigm, one of its key characteristics is the ability to redefine operations in a subclass. However, we define operations at the specification level, where the system is considered to be a black box and hence operations are part of the system and not of a specific class [15]. Therefore, we do not consider operation redefinition in our approach.

Data Types. Although we assign a specific data type to each attribute, our approach to reasoning ignores the data types assigned to the attributes, and treats them as integers (an integer can be interpreted as a string). In contrast, however, our approach does make sure that the input parameters of the operations and the objects participating in the association do have the right types, as defined in the translation process.

Collections. A collection represents a group of elements of a certain type. OCL distinguishes between bags (collections with duplicates), sets (collections without), sequences (an ordered bag) and ordered sets [16]. We only consider sets since they are the ones handled by first order logic. Most OCL constraints and OCL operations will in practice result in sets, which is appropriate for our translation process.

Arithmetic Operations. We do not consider arithmetic operations such as addition, substraction, multiplication and division. Similarly, we do not support if/else conditions.

Model Evolution. First-order logic lacks the ability to represent the evolution of a domain over time, i.e. it does not inherently consider the notion of state. However, the evolution of a system over time is a key element in our approach. Therefore, to solve this limitation, we base our work on [12], which add the notion of a time predicate and represents the derivation rules in a way that considers it, as previously described.

4.5. ***Formalization of the Tests***

Our approach is aimed at providing the designer with different tests that allow him to assess the correctness of the BAUML model. Once we have the model encoded in logic, each validation test is reformulated into a query satisfiability problem over a derived predicate. So, for each test, a derived predicate that formalizes the test is defined. With this input, together with the translated schema, any satisfiability checking method able to deal with negation of derived predicates can be used to reason about BAUML models.

We formally define in this section the properties in Section 3 in terms of query satisfiability tests over a derived predicate. We illustrate this formalization through the translation of our running example.

4.5.1. Verification Tests

Liveliness of a Class or Association The liveliness test of a class or an association will ensure that an instance of it can be successfully created and that it persists in the system until the transition that has created it ends. The general form of the test is the following, where *el* is the name of the class or association:

$$livelinessTestEl() \leftarrow el(\overline{p}, t) \wedge validState(t).$$

If the class or association is not created in the business process, then the time component t would be omitted from the derivation rule shown above. As during the execution of activity diagrams integrity constraints can be violated, $validState(t)$ ensures that the integrity constraint is only checked when no activity diagrams are in the middle of an execution.

Minimum cardinality Given a n-ary association *asso*, with m participants, where $m > 0$ and a minimum cardinality of x in the end of class C, we would define the test in the following way:

$$\begin{aligned} minCardTest() \leftarrow\ & asso(p_1, ..., p_{m-1}, c_1, t) \wedge \ldots \wedge asso(p_1, \ldots, p_{m-1}, c_x, t) \\ & \wedge \neg extraAsso(p_1, \ldots, p_{m-1}, c_1, \ldots, c_x, t) \wedge c_1 \neq c_2 \wedge \ldots \wedge c_{x-1} \neq c_x \\ & \wedge validState(t) \\ extraAsso(p_1, \ldots, p_{m-1}, c_1, \ldots, c_x, t) \leftarrow\ & asso(p_1, ..., p_{m-1}, c_{x+1}, t) \wedge C(c_1) \\ & \wedge \ldots \wedge C(c_{x+1}) \wedge c_1 \neq c_2 \wedge \ldots \wedge c_x \neq c_{x+1} \end{aligned}$$

Maximum cardinality To avoid infinite loops, the maximum cardinality should be bounded before running this test. Given a n-ary association *asso*, with m participants, and a maximum cardinality of x in the end of class C, we would define the test in the following way:

$$\begin{aligned} maxCardTest() \leftarrow\ & asso(p_1, ..., p_{m-1}, c_1, t) \wedge \ldots \wedge asso(p_1, \ldots, p_{m-1}, c_x, t) \\ & \wedge c_1 \neq c_2 \wedge \ldots \wedge c_1 \neq c_x \wedge \ldots \wedge c_{x-1} \neq c_x \wedge \ldots \wedge validState(t) \end{aligned}$$

For both cardinality tests, a satisfactory answer means that the cardinalities are correct, whereas a negative answer means that the cardinality should be greater, for the minimum cardinality test, or lower, for the maximum.

Redundancy of integrity constraints An integrity constraint *ic* is redundant if other constraints subsume it. What we do is to remove the constraint from the schema and test if the model can fulfill it:

$$icRedundant() \leftarrow ic(t) \wedge validState(t)$$

If the result is positive, there is no other constraint restricting *ic*. Therefore, *ic* is not redundant. Otherwise, *ic* is redundant and can be deleted from the schema.

State reachability Checking the reachability of a state is equivalent to checking the liveliness of the corresponding class:

$$stateReachabilityTest() \leftarrow el(\overline{p}, t) \wedge validState(t).$$

Transition Applicability Given a transition $t = \langle v_s, o, e, c, x, v_t \rangle$, checking its applicability means ensuring that v_s is reachable and that o (if any) is true:

$$transApplTest() \leftarrow predV_s(\overline{p}, t)[\wedge ocl(t)] \wedge validState(t)$$

$predV_s$ corresponds to the predicate representing the subclass that corresponds to state v_s, *ocl* the ocl condition o in the transition (if there is one). We have to ensure that these conditions are met on a $validState(t)$, as transitions begin their execution in this state.

Transition Executability There are many factors that should be considered for the executability of a transition $t = \langle v_s, o, e, c, x, v_t \rangle$:

1. The source state v_s.
2. Any OCL conditions that may appear in the transition: o.
3. The target state v_t.
4. The event e or effect x which is part of t.
5. There cannot be any intermediate valid state ($validState(t)$) between the source state v_s and v_t.

Conditions 1 and 2 refer to the time, $t1$, before the execution of the transition begins. Condition 3 refers to the time, $t2$, at the end of the execution. Between $t1$ and $t2$ the tasks in e or x will execute.

Then, the form of this test will be the following:

$$transExecTest() \leftarrow predV_s(oid, \ldots, t_1)[\wedge ocl(t_1)] \wedge validState(t_1) \wedge predV_t(oid, \ldots, t_2) \wedge validState(t_2) \\ \wedge \neg validState(t_3) \wedge time(t_3) \wedge t1 < t2 \wedge t1 < t3 \wedge t3 < t2 \wedge execLastTask'(oid, \ldots, t_2)$$

$execLastTask'(oid, \ldots, t_2)$ corresponds to the last task in the activity diagram for event e or effect x. This ensures we are executing the right event or effect to perform the transition. $predV_s$ and $predV_t$ are the predicates representing the subclasses that correspond to the source and the target states, respectively.

Precondition Redundancy Test It checks if the precondition of the task is superfluous. The test is defined in the following way, for task $task_i$:

$$precRedTest() \leftarrow par_1(\overline{y_1}) \wedge \ldots \wedge par_n(\overline{y_n}) \wedge \neg pre_{task_i}(\overline{y}, t),$$

where par_i corresponds to the input parameters and pre_{task_i} the precondition of the task. If the test executes successfully, then the precondition is redundant.

Applicability Test It checks whether the necessary requirements for the excution of a certain task are met. It has the form, for task $task_i$:

$$applicabilityTask() \leftarrow pre_{task}(\overline{y}, t) \wedge task'_{i-1}(p_a, t).$$

$task'_{i-1}$ represents the predicate corresponding to the successful execution of the previous task to $task_i$.

Executability Test It checks if a certain task can be executed. Particularly useful to ensure that all paths from a decision node can be taken. The test has the form:

$$executabilityTask() \leftarrow task'(p_a, t).$$

Regarding the two last tests, and in particular regarding verification of activity diagrams and operation contracts, it is worth noting that we have defined the translation of the BAUML models into logic in a way in which activity diagrams cannot stop in the middle of an execution: they either execute successfully or they do not execute at all. Therefore, if we do not make some changes, the result of the executability and applicability tests of the tasks will depend, in the general case, on the executability of the event or effect they are part of.

What would be interesting to know is if a task is applicable or executable considering only the context required up to the point of its execution. To achieve this, we generate the logic schema without any of the predicates and rules produced by Algorithm 4 which force the execution to move forward. We need to keep the rest of rules for the remaining activity diagrams, to ensure that they execute until the end.

4.5.2. Validation Tests

Path Inclusion or Exclusion There are three tests that we can perform to check path inclusion and exclusion between two relationships, $asso1$ and $asso2$, that have the same participants, p_1 to p_n:

$$pathIncExcTest1() \leftarrow asso1(p_1, ..., p_n, t) \wedge asso2(p_1, ..., p_n, t) \wedge validState(t)$$

$$pathIncExcTest2() \leftarrow asso1(p_1, ..., p_n, t) \wedge \neg asso2(p_1, ..., p_n, t) \wedge validState(t)$$

$$pathIncExcTest3() \leftarrow asso2(p_1, ..., p_n, t) \wedge \neg asso1(p_1, ..., p_n, t) \wedge validState(t)$$

The first test checks if it possible for $asso1$ and $asso2$ to have instances with exactly the same participants. Tests 2 and 3 check the opposite: *is it possible to have instances of* $asso1$ *(*$asso2$*) if there is not the corresponding instance in* $asso2$ *(*$asso1$*)?*

Test	**Result**	
	True	*False*
pathIncExcTest1	Allows path inclusion	Does not allow path inclusion → **Path exclusion**
pathIncExcTest2	$asso1$ does not depend on $asso2$	$asso1$ requires $asso2$ → **Path inclusion**
pathIncExcTest3	$asso2$ does not depend on $asso1$	$asso2$ requires $asso1$ → **Path inclusion**

Table 1: Table showing the interpretation of the different results for the path inclusion and exclusion tests.

Table 1 summarizes the meaning of the results of the tests. A positive result in the tests may indicate that there is a constraint missing, whereas a negative result will indicate that the paths are mutually exclusive ($pathIncExcTest1$) or that they are inclusive ($pathIncExcTest2$ and $pathIncExcTest3$).

Missing irreflexive constraints Given an association $asso$ which relates the same class c to itself, we can check if it is reflexive by performing the following test:

$$reflexTest() \leftarrow asso(r_1, r_1, t) \wedge validState(t)$$

Full transition coverage The full transition coverage test checks if all potential combinations of transitions can actually take place.

$$transCovTest() \leftarrow lastTask'_{tr_1}(oid, ..., t_1) \wedge \ldots \wedge lastTask'_{tr_n}(oid, ..., t_n) \wedge t_1 < t_2 \wedge \ldots \wedge t_{n-1} < t_n \wedge \neg intValidState(t_1, t_2) \wedge \ldots \wedge \neg intValidState(t_{n-1}, t_n)$$

$$intValidState(t_a, t_b) \leftarrow time(t_a) \wedge time(t_b) \wedge time(t_c) \wedge t_a < t_c \wedge t_c < t_b \wedge validState(t_c)$$

Predicate $intValidState$ is necessary to ensure that the only transitions executed in the diagram are the ones stated in the first derivation rule. If the test is satisfiable, the sequence of transitions is valid. Otherwise, this may indicate an error in either the executability of the transitions themselves or in this particular combination of transitions. If the transitions have been checked satisfactorily for their executability, then the issue is in the combination of transitions.

5. Automatic Reasoning in our Running Example

With the purpose of showing the feasibility of our approach, we have developed a prototype tool which is able to automatically translate the models into the required logic for satisfiability checking, and then determine whether they fulfill certain semantic correctness properties. It is worth mentioning that reasoning

in a logic schema as expressive as the one we obtain is known to be a semidecidable problem and, therefore, it is not possible to always ensure termination. The same remark applies regarding (the lack of) efficiency of reasoning. However, our goal here is not to present a tool to solve the problem in general, but to show that we can already find satisfactory results for particular situations when using some of the existing technology. Then, our goal in this section is to show that our approach is feasible in practice since our implementation works for the example presented in the paper.

5.1. Implementation of the Approach

Given a BAUML model graphically defined using ArgoUML [17], we export it into an XMI file which is then provided to the prototype. The prototype applies the translation techniques explained previously and can generate the verification tests automatically. The user then selects the tests that he wishes to perform and the translation result obtained previously is transparently provided as input to another tool, SVTe. Given a certain property over the logic schema, SVTe can tell whether this property is fulfilled. Following this workflow, the user can answer several questions which deal with the semantic correctness of the model.

We use SVTe [18] as a satisfiability checking engine because it is developed by our research group. Moreover, it is one of the few satisfiability checkers able to deal with negative derived predicates. Not only does it indicate whether the tested query is satisfiable, but also provides a sample instance if the test is satisfied or, otherwise, the subset of constraints that prevent it from being satisfied.

SVTe handles each test as a satisfiability problem and it uses the CQC_E method [19], aimed at building a consistent state of a database schema that satisfies a given goal, represented as a set of one or more literals. The method starts with an empty solution, and given the goal, the database schema, the constraints and the derivation rules, it tries to obtain a set of base facts that satisfy the goal and all constraints. The CQC_E method is a semidecision procedure for finite satisfiability. This means that it does not terminate in the presence of solutions with infinite elements. However, termination is ensured if the model satisfies the conditions identified in [4].

To instantiate the variables during the inference process, the method uses Variable Instantiation Patterns (VIPs), which generate only the relevant facts that need to be added to the schema to satisfy the goal. If no instance that satisfies the database schema and the constraints is found, then the VIPs guarantee that the goal cannot be achieved with the given schema and constraints.

Logically, we consider all the elements in the BAUML framework (the class, state machine, activity diagrams and OCL operation contracts) when reasoning. It is important to point out that our reasoning approach works only with the specification of the model and does not need an initial instance of the model to obtain results.

5.2. Some Test Examples

Liveliness Tests. These tests show whether the classes in our model (both artifacts and objects) can have at least one instance. Figure 4 shows the result of the execution of the liveliness tests for all the classes in the model. As it can be seen, all the tests have executed successfully.

Let's look at the result for the liveliness test of *AcceptedSub*. By double-clicking on the result, the prototype opens a new window (Figure 5, left) stating the base predicates required to achieve the goal of having an instance of *AcceptedSub* at a valid state. They correspond to classes (*Author* and *Session*) and to tasks (e.g. *AssignToSession*, *RegisterNewSubmission*). The predicates representing classes correspond to those that are not created by the operations in the model, and therefore they do not include a time component. On the other hand, the predicates corresponding to the operations have a component representing time (the last term). By analyzing the time component, we can see the execution order which has to be followed by the operations. According to the result, the order is: *RegisterNewSubmission*, *AddAuthorToSubmission*, *AddReviewInfo* and *AssignToSession*, which is the one established by the transitions in the state machine diagram and the activity diagrams.

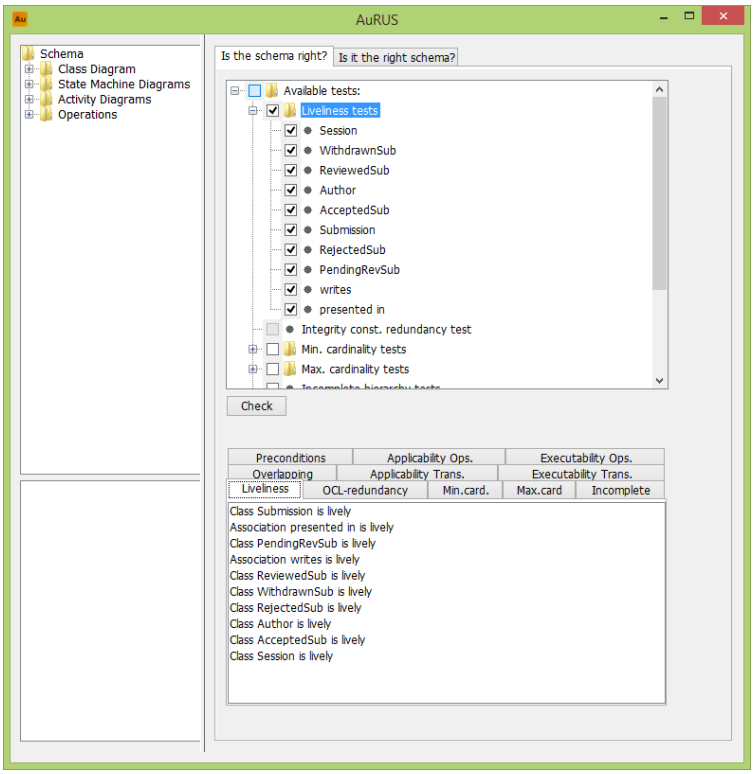

Figure 4: Selection and result of the execution for the liveliness tests of all the classes in the class diagram

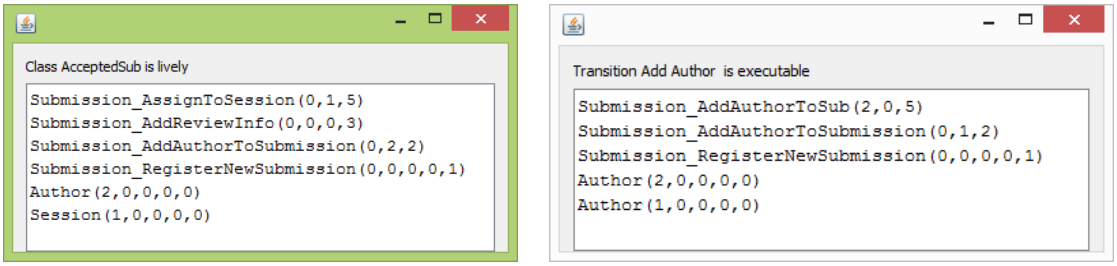

Figure 5: Base predicates to achieve *AcceptedSub* (left) and to ensure the successful execution of transition *AddAuthor* (right).

Transition Executability Tests. The transition executability tests prove that a certain transition in the state machine diagram is executable. If we focus on the transition containing event *Add Author*, its executability test would be defined in the following way:

$$execTransTest() \leftarrow PendingRevSub(s, id, t, d, t_0) \wedge validState(t_0) \wedge PendingRevSub(s, id, t, d, t_1) \\ \wedge validState(t_1) \wedge AddAuthorToSub(a, s, t_1) \wedge t_0 < t_1 \wedge \neg(intermediateValidState(t_0, t_1))$$

In our example, all transitions are executable. If we check the details for *Add Author* in Figure 5, right, we see the necessary base predicates that will eventually lead to the successful execution. In this case, we need to have two different *Author*s, and we will need to execute tasks *RegisterNewSubmission* and *AddAuthorToSubmission*, which correspond to transition *SubmitPaper*. Then, once the submission has been created, we need to execute task *AddAuthorToSub* which is in the transition that corresponds to *AddAuthor*.

5.3. Performance

Our reasoning approach is based on satisfiability checking, which is a well-known NP-complete problem and, as such, no algorithm has been found that can solve the problem efficiently. Therefore, it cannot be expected that the prototype we have created will provide fast execution times. It works and provides results for the tests related to our example; however, it suffers from efficiency issues and termination in reasonable time cannot be guaranteed for every example.

This section briefly analyses the performance of the tool for the running example. The resulting translation contains approximately[1] 74 constraints and 60 derivation rules. Table 2 shows the execution times for the verification tests, which have been generated automatically.

Table 2: Table showing execution times of the tests in our example

Tests	**Time (ms)**	**Time (min.)**
Liveliness		
Submission	20676	0,3446
WithdrawnSub	81192	1,3532
PendingRevSub	21678	0,3613
ReviewedSub	5327900	88,79833333
AcceptedSub	823072	13,71786667
RejectedSub	439559	7,325983333
writes	2497	0,041616667
presented in	806750	13,44583333
Min. Cardinality Constraints		
writes	2537	0,042283333
presented in	834450	13,9075
Max. Cardinality Constraints		
presented in	836167	13,93611667
Trans. Applicability Tests		
Add Author	21899	0,364983333
Review Submission	21938	0,365633333
Withdraw Submission	21928	0,365466667
Trans. Executability Tests		
Add Author	740174	12,33623333
Review Submission (fail)	2985656	49,76093333
Withdraw Submission	867448	14,45746667
Review Submission (success)	4739054	78,98423333
Submit Paper	5790	0,0965
Precondition Redundancy Tests		
Add Author to Submission	1750	0,029166667
Add Review Info	25682	0,428033333
Assign to Session	2854	0,047566667
Add Rejection Reason	2408	0,040133333
Withdraw	2434	0,040566667
Add Author to Sub	2733	0,04555
Op. Applicability Tests		
Add Author to Submission	1006	0,016766667
Add Review Info	2783	0,046383333
Assign to Session	3223	0,053716667
Add Rejection Reason	2774	0,046233333
Withdraw	2841	0,04735
Add Author to Sub	2123	0,035383333
Op. Executability Tests		
Register New Submission	575	0,009583333
Add Author to Submission	1743	0,02905
Add Review Info	7398	0,1233
Assign to Session	212789	3,546483333
Add Rejection Reason	115237	1,920616667
Withdraw	15113	0,251883333
Add Author to Sub	15782	0,263033333

[1]Numbers may vary a little depending on the test being performed.

As it can be seen, there is a wide range of execution times. The fastest test (executability of task *Register New Submission*) took about half a second, whereas the longest (liveliness of *ReviewedSub*) took almost 90 minutes. This variety in execution times can be mostly explained by their different complexity.

Operation executability and applicability tests are among the fastest. The reason is that some integrity constraints are *disabled* in order for these tests to provide meaningful results. Therefore, in these cases there is no need to force the execution to move forward, just ensure that up to that point the operations have executed in the right order. This is guaranteed by the derivation rules. Similarly, precondition redundancy tests are also quite fast, as they do not require the system to be in a valid state.

Transition applicability tests provide the results in a very reasonable time. Except for *SubmitPaper* - which has no actual requirements for being applied - the rest of tests only need an instance of *PendingRevSub*, which can only be obtained by executing transition *SubmitPaper*. Therefore, in this particular example they are very straightforward.

Transition executability tests are more complex since their definition needs ensuring that initial and final states are correct, that the right event is executed and that there are no valid intermediate states in-between. Moreover, all integrity constraints need to be considered. For this reason, they are costly, especially *Review Submission*, which requires a *PendingRevSub* at the start and whose activity diagram has three tasks.

Liveliness tests can also be quite costly, because their result depends on figuring out the right combination of operations or tasks that leads to obtaining an instance of the class or relationship. In general, the more tasks required (*AcceptedSub*, *RejectedSub*), the longer the test takes. It is noteworthy that the liveliness test of *ReviewedSub*, a superclass of both *AcceptedSub* and *RejectedSub*, took around 88 minutes instead of the 13 minutes or 7 minutes (respectively) for the subclasses. This is probably due to the fact that SVTe performs an iterative deepening depth-first search to find a solution and, in this particular example, there two possible paths to obtain a *ReviewedSub*. Minimum and maximum cardinality tests require instances of the classes that define the association. For this reason, they should take as long, or longer, than the corresponding liveliness tests, as shown here.

To summarize, as we have seen with our running example the tool can obtain results but in some cases the time required to do so is too high. The bottleneck is not in the translation process itself, but in the reasoner. A possible area for improvement would be to introduce parallelism or to tweak the algorithm in the reasoner that searches for a solution.

6. Related Work

We focus mainly on reasoning on business process models and reasoning on UML/OCL models. However, it is worth mentioning that there are similar approaches in other areas such as Multi-Agent Systems, which are out of the scope of this paper. For example, [20] formalizes and validates SEA_ML models, a domain-specific modeling language, by using the Alloy analyzer. We distinguish between proposals dealing with syntactical and structural reasoning, and approaches that focus on semantic reasoning and existing tools for reasoning on artifact-centric business process models.

6.1. Syntactical & Structural Reasoning

Most of this research follows a process-centric perspective and it has centered on detecting errors in the flow of activities. This is why many authors translate workflow models into Petri nets, which are formal.

Some works translate generic workflows into graphs to check certain properties such as soundness, existence of deadlocks, etc. For instance, Choi and Zhao [21] consider cyclic workflows, whereas Lin et al. [22] do not. A more informal approach is used by van Dongen, van der Aalst and Verbeek [23], where EPCs (Event-driven Process Chains) are first reduced and then translated into a Petri net. However, this approach may require user intervention to determine whether a potentially conflictive situation is erroneous or not.

Störrle [24] gives a formal definition of UML activity diagrams, by mapping the elements of activity diagrams, including the data flow, to colored Petri-nets. Then, standard properties of Petri nets like state reachability can be checked in the activity diagrams. However, data is only considered in terms of the flow of the diagram and no formal definition of the tasks and their impact on the data is given.

A seminal paper by van der Aalst et al. deals with structural reasoning [25]. It studies different variations of Workflow nets and how the different notions of soundness apply to them. Although focused on whether it is decidable to verify the workflows, the paper offers an overview of existing techniques to verify them.

All these works make an important contribution by providing methods for the verification of the syntactical or structural business process models and their results can be applied to our approach. However, being process-centric, they do not deal with artifacts nor the meaning of tasks, both of which are key elements in our proposal.

6.2. Semantic Reasoning

We examine semantic reasoning in data-aware approaches, artifact-centric business process models and UML diagrams, since these are the three main features covered by BAUML models.

6.2.1. Data-Aware Approaches

There are some approaches that, although not specifically artifact-centric, do consider the data [26, 27, 28]. Sidorova, Stahl and Trcka [27] deal with soundness in WFD-nets considering the read/write/delete operations in the process. Awad, Decker and Lohnmann [26] detect errors in the flow by considering the evolution of data from one state to the next. Given various Petri nets representing the evolution of artifacts, Lohmann and Wolf [28] create a valid choreography taking into consideration the policies that restrict the valid interaction and the goal states. However, in contrast to our work, none of these approaches deals with the detailed meaning of the tasks and they do not have an underlying conceptual schema representing the data and its relationships.

Similarly, Rinderle-Ma [29] studies the conditions which guarantee the correctness of the data flow when making changes to the control flow or the data flow of a business process. Nevertheless, it mainly focuses on read-write dependencies and there is no data model.

Knuplesch et al. [30] check if the business process model fulfills certain properties that may take the data into consideration. They create an abstraction of the model to avoid the state explosion caused by the data. However, the structure of the data is not fully represented.

To sum up, although these works follow approaches that could be considered similar to the one in this paper, they do not use models as complex as ours and tend to simplify the underlying data.

6.2.2. Semantic Reasoning on Artifact-centric BPM

Several approaches use data-centric dynamic systems (DCDSs), grounded on logic, as the basis for reasoning [31, 32, 33]. There are several ways to represent the data. Bagheri Hariri et al. use a relational database together with a set of condition-action rules and actions defined in logic [31]. In another work, Bagheri Hariri et al. [33] use a Knowledge and Action Base defined in a variant of Description Logics. Similarly, Calvanese et al. [32] map an ontology to a DCDS in order to verify certain temporal properties expressed in a variant of μ-calculus.

Damaggio et al. [34] represent artifacts using a set of variables, which are updated by services defined by pre and postconditions in first-order logic. This work is actually a summary of the results of Damaggio, Deutsch and Vianu [35], and Deutsch et al. [36]. The authors check whether the model fulfills a set of properties defined in LTL-FO , which is not as powerful as $\mu\mathcal{L}$. Despite this, they can represent integrity constraints, such as primary keys and foreign keys, and they allow arithmetic constraints in the services' definition, something not permitted in other works [31].

Similarly, Belardinelli, Lomuscio and Patrizi [37] check whether a deployed artifact system, defined in logic, fulfills a property defined in FO-CTL (a first-order extension of CTL). The fact that the artifact system has been deployed implies that it does not deal with infinite data, but there is rather an upper bound on the number of elements in each state.

Gerede and Su [38] define a specification language, ABSL, based on CTL, to specify the artifacts' lifecycle behavior and is able to check if the model satisfies a certain property defined in ABSL. However, like in the case of LTL-FO, CTL is not as powerful as μ-calculus.

Bhattacharya et al. [39] reason on reachability, dead-ends and redundancy. Artifacts are represented using a set of attributes, an identifier and a state. Services are defined by means of predicates *new*, *defined*

and assignations between variables. Business rules are defined by means of *if* rules. However, this proposal does not deal with actual data, but rather an abstraction of it.

Sumarizing, all these works represent, in contrast to our approach, artifact-centric business process models and the properties to be checked, in languages derived from logic. Consequently, the models under consideration are formal, but they are not intuitive nor practical from the point of view of the business. Moreover, desirable properties have to be defined manually. In addition to this, they have been proposed at a theoretical level and there is no tool yet to show its feasibility in practice.

From a different perspective, the Guard-Stage-Milestone (GSM) approach provides a more business-friendly representation of artifact-centric business processes, and several works study reasoning on these models. So, Heath et al. present a system to model and execute artifact systems [40] but it is limited to simulating the behavior of the model given certain data, instead of reasoning on generic properties. Other works use the GSMC tool to reason on GSM models [41, 42]. However, several restrictions are imposed on the data types and they only allows one instance per artifact [41]. A follow-up work by the same authors [42] perform model checking over GSM models from a multi-agent perspective; however the bound placed on the number of objects may sometimes lead to unreliable results when this bound is exceeded.

Hence, although GSM approaches follow a more user-friendly notation, the reasoning approaches have either been presented theoretically or the existing tools are limited, such as allowing only one instance per artifact.

Closer to our work, Weber, Hoffman and Mendling [43] perform verification over process models considering the meaning of the tasks. They use BPMN diagrams whose tasks may be annotated with preconditions and effects defined in logic, and use an optional ontology to define the underlying data. Time is not considered explicitly, but they have implemented a prototype tool that can perform some verification tests. However, as neither the ontology nor the details tasks are compulsory, the final results can only be partial or provide an intuitive idea of potential issues.

On the other hand, Borrego, Gasca and Gómez-López [44] use *artifact union graphs* (a Petri-net like notation) to check state reachability and weak termination. The artifact union graphs represent the states and the services that trigger the transitions between those states. Constraints and details of the services are defined in pre and postconditions following a certain grammar. They also have a tool which is able to perform the verification. Although their approach deals with several artifacts, the domain of the artifacts' attributes is constrained and the type of tests that can be performed is limited.

Finally, Lohmann [45] follows a completely different approach. Instead of checking the business process model's conformance to certain rules , the author uses a *compliance by design* approach: it generates business process models that already comply with the rules. However, it requires an initial business process model on top of which a new model is built which fulfills the rules. This means that there may be errors in the model which may not be detected. In addition, the notation used to represent the artifact-centric models is based on Petri nets to represent the lifecycle of the artifact, and no details of the tasks are given.

6.2.3. Semantic Reasoning on UML models

Most of the proposals for reasoning on UML models deal only with one diagram. For instance, several works [9, 46] focus on the class diagram, others handle state-machine diagrams [47] or focus on activity diagrams [48]. On the other hand, Lucas, Molina and Toval Álvarez [49] performed a systematic literature review of works dealing with various UML diagrams but only four of the analyzed papers perform reasoning on more than one of the diagrams in our approach: they can handle class and state machine diagrams.

Eshuis [48] checks the consistency between UML activity diagrams and class diagrams. However, instead of dealing with the activity diagram's actions specification, it considers that the object flow acts as a precondition and postcondition of the actions. These constraints are derived automatically from the diagram. Therefore, it only focuses on create, read, write and update dependencies among tasks.

Straeten, Simmonds and Mens [50] check the consistency between different UML diagrams using Description Logics, but target very basic properties and do not consider the definition of the operations or the additional constraints in the process, like we do.

Although not explicitly an artifact-centric approach, Reggio, Leotta and Ricca [51] study the quality of business processes represented using UML activity diagrams.

Summarizing, and despite the significant results on semantic reasoning on UML models, none of the current proposal is able to handle together all the elements of a BAUML model.

7. Conclusions and Further Work

Checking the correctness of BAUML models as early as possible is important to avoid the propagation of errors to the implementation of the processes. There are different tests that can be performed to determine the correctness. We focus on semantic tests, as they take into consideration the data and the meaning of the tasks when providing their results. Most of the available methods for performing semantic reasoning on artifact-centric business processes use as input models grounded on logic. These models are very formal and so they are not practical from the point of view of the business, as they are difficult to understand.

Bearing this in mind, in this paper we have proposed an approach to perform semantic reasoning on an artifact-centric business process model defined using the BAUML framework, which uses a combination of UML and OCL models. UML and OCL are standard, well-known languages, and UML is based on a graphical notation, which makes it easier to understand than languages such as logic. To be able to perform the reasoning with these models, we have used an existing method for satisfiability checking, which requires a set of derivation rules in first order logic and a goal.

We have shown how to automatically translate the BAUML models into a set of rules as required by the satisfiability methods, and we have detailed the algorithms that are able to do so. Moreover, we have formalized several tests that can be automatically generated and then applied to determine the semantic correctness of the BAUML model.

Hence, the advantage of this approach is that, starting from models which have a high-level of abstraction and are more intuitive, we show a way to check automatically check their correctness without requiring user intervention or knowledge of formal languages, such as first-order logic.

We have also shown the feasibility of our approach by a prototype that automatically translates a BAUML model into logic and can also generate the verification tests automatically.

Further work is required towards improving efficiency of the reasoner and identifying conditions that allow ensuring termination of reasoning for the input BAUML models.

Acknowledgments

This work has been partially supported by the Ministerio de Economía y Competitividad, under project TIN2014-52938-C2-2-R and by the Secretaria d'Universitats i Recerca de la Generalitat de Catalunya under project 2014 SGR 1534.

References

[1] M. Estañol, A. Queralt, M. R. Sancho, E. Teniente, Specifying artifact-centric business process models in UML, in: B. Shishkov (Ed.), BMSD 2014, Revised Selected Papers, volume 220 of *LNBIP*, Springer, 2015, pp. 62–81.

[2] M. Estañol, M. Sancho, E. Teniente, Verification and validation of UML artifact-centric business process models, in: J. Zdravkovic, M. Kirikova, P. Johannesson (Eds.), Advanced Information Systems Engineering - 27th International Conference, CAiSE 2015, Proceedings, volume 9097 of *LNCS*, Springer, 2015, pp. 434–449.

[3] R. Hull, Artifact-centric business process models: Brief survey of research results and challenges, in: R. Meersman, Z. Tari (Eds.), OTM 2008, volume 5332 of *LNCS*, Springer, 2008, pp. 1152–1163.

[4] D. Calvanese, M. Montali, M. Estañol, E. Teniente, Verifiable UML artifact-centric business process models, in: J. Li, X. S. Wang, M. N. Garofalakis, I. Soboroff, T. Suel, M. Wang (Eds.), CIKM 2014, ACM, 2014, pp. 1289–1298.

[5] M. Estañol, M.-R. Sancho, E. Teniente, Reasoning on UML data-centric business process models, in: [52], 2013, pp. 437–445.

[6] A. Queralt, E. Teniente, Reasoning on UML conceptual schemas with operations, in: P. van Eck, J. Gordijn, R. Wieringa (Eds.), CAiSE 2009, LNCS, Springer, 2009, pp. 47–62.

[7] G. Rull, C. Farré, A. Queralt, E. Teniente, T. Urpí, AuRUS: explaining the validation of UML/OCL conceptual schemas, Software & Systems Modeling 14 (2015) 953–980.

[8] M. Utting, B. Legeard, Practical Model-Based Testing: A Tools Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[9] A. Queralt, E. Teniente, Verification and validation of UML conceptual schemas with OCL constraints, ACM Trans. Softw. Eng. Methodol. 21 (2012) 13.
[10] P. Kardasis, P. Loucopoulos, Expressing and organising business rules, Information & Software Technology 46 (2004) 701–718.
[11] ISO, ISO/IEC 19505-2:2012 - OMG UML superstructure 2.4.1, 2012. Available at: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=52854.
[12] A. Queralt, E. Teniente, Specifying the semantics of operation contracts in conceptual modeling, in: Journal on Data Semantics VII, volume 4244 of *LNCS*, Springer, 2006, pp. 33–56.
[13] J. Cabot, R. Pau, R. Raventós, From UML/OCL to SBVR specifications: A challenging transformation, Inf. Syst. 35 (2010) 417–440.
[14] K. Anastasakis, B. Bordbar, G. Georg, I. Ray, On challenges of model transformation from UML to alloy, Software and System Modeling 9 (2010) 69–86.
[15] C. Larman, Applying UML and Patterns, 2nd edition ed., Prentice Hall, 2002.
[16] ISO, ISO/IEC 19507:2012 - OMG OCL version 2.3.1, 2012. Available at: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57306.
[17] ArgoUML, ArgoUML, 2017. URL: http://argouml.tigris.org/.
[18] C. Farré, G. Rull, E. Teniente, T. Urpí, SVTe: a tool to validate database schemas giving explanations, in: L. Giakoumakis, D. Kossmann (Eds.), DBTest 2008, ACM, 2008, pp. 1–6.
[19] G. Rull, C. Farré, E. Teniente, T. Urpí, Providing explanations for database schema validation, in: S. S. Bhowmick, J. Küng, R. Wagner (Eds.), DEXA, volume 5181 of *LNCS*, Springer, 2008, pp. 660–667.
[20] S. Getir, M. Challenger, G. Kardas, The formal semantics of a domain-specific modeling language for semantic web enabled multi-agent systems, Int. J. Cooperative Inf. Syst. 23 (2014).
[21] Y. Choi, J. L. Zhao, Decomposition-Based Verification of Cyclic Workflows, in: D. Peled, Y.-K. Tsay (Eds.), ATVA 2005, volume 3707 of *LNCS*, Springer, 2005, pp. 84–98.
[22] H. Lin, Z. Zhao, H. Li, Z. Chen, A novel graph reduction algorithm to identify structural conflicts, in: HICSS, IEEE Computer Society, 2002, p. 289.
[23] B. F. van Dongen, W. M. P. van der Aalst, H. M. W. Verbeek, Verification of EPCs: Using reduction rules and Petri nets, in: O. Pastor, J. Falcão e Cunha (Eds.), CAiSE 2005, volume 3520 of *LNCS*, Springer, 2005, pp. 372–386.
[24] H. Störrle, Semantics and verification of data flow in UML 2.0 activities, Electr. Notes Theor. Comput. Sci. 127 (2005) 35–52.
[25] W. M. P. van der Aalst, K. M. Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, M. T. Wynn, Soundness of workflow nets: classification, decidability, and analysis, Formal Aspects of Computing 23 (2011) 333–363.
[26] A. Awad, G. Decker, N. Lohmann, Diagnosing and repairing data anomalies in process models, in: S. Rinderle-Ma, S. W. Sadiq, F. Leymann (Eds.), Business Process Management Workshops, volume 43 of *LNBIP*, Springer, 2009, pp. 5–16.
[27] N. Sidorova, C. Stahl, N. Trcka, Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible, Inf. Syst. 36 (2011) 1026–1043.
[28] N. Lohmann, K. Wolf, Artifact-centric choreographies, in: P. P. Maglio, M. Weske, J. Yang, M. Fantinato (Eds.), ICSOC 2010, volume 6470 of *LNCS*, Springer, 2010, pp. 32–46.
[29] S. Rinderle-Ma, Data flow correctness in adaptive workflow systems, EMISA Forum 29 (2009) 25–35.
[30] D. Knuplesch, L. T. Ly, S. Rinderle-Ma, H. Pfeifer, P. Dadam, On enabling data-aware compliance checking of business process models, in: J. Parsons, M. Saeki, P. Shoval, C. C. Woo, Y. Wand (Eds.), ER 2010, volume 6412 of *LNCS*, Springer, 2010, pp. 332–346.
[31] B. Bagheri Hariri, D. Calvanese, G. D. Giacomo, A. Deutsch, M. Montali, Verification of relational data-centric dynamic systems with external services, in: R. Hull, W. Fan (Eds.), PODS, ACM, 2013, pp. 163–174.
[32] D. Calvanese, G. D. Giacomo, D. Lembo, M. Montali, A. Santoso, Ontology-based governance of data-aware processes, in: M. Krötzsch, U. Straccia (Eds.), RR, volume 7497 of *LNCS*, Springer, 2012, pp. 25–41.
[33] B. Bagheri Hariri, et al., Verification of description logic knowledge and action bases, in: L. D. Raedt, et al. (Eds.), ECAI, volume 242 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2012, pp. 103–108.
[34] E. Damaggio, A. Deutsch, R. Hull, V. Vianu, Automatic verification of data-centric business processes, in: S. Rinderle-Ma, F. Toumani, K. Wolf (Eds.), BPM 2011, volume 6896 of *LNCS*, Springer, 2011, pp. 3–16.
[35] E. Damaggio, A. Deutsch, V. Vianu, Artifact systems with data dependencies and arithmetic, ACM Trans. Database Syst. 37 (2012) 22.
[36] A. Deutsch, R. Hull, F. Patrizi, V. Vianu, Automatic verification of data-centric business processes, in: R. Fagin (Ed.), ICDT, volume 361 of *ACM International Conference Proceeding Series*, ACM, 2009, pp. 252–267.
[37] F. Belardinelli, A. Lomuscio, F. Patrizi, Verification of deployed artifact systems via data abstraction, in: G. Kappel, Z. Maamar, H. R. M. Nezhad (Eds.), ICSOC 2011, volume 7084 of *LNCS*, Springer, 2011, pp. 142–156.
[38] C. E. Gerede, J. Su, Specification and verification of artifact behaviors in business process models, in: B. J. Krämer, K.-J. Lin, P. Narasimhan (Eds.), ICSOC 2007, volume 4749 of *LNCS*, Springer, 2007, pp. 181–192.
[39] K. Bhattacharya, C. Gerede, R. Hull, R. Liu, J. Su, Towards formal analysis of artifact-centric business process models, in: G. Alonso, P. Dadam, M. Rosemann (Eds.), BPM 2007, volume 4714 of *LNCS*, Springer, 2007, pp. 288–304.
[40] F. T. Heath, et al., Barcelona: A design and runtime environment for declarative artifact-centric BPM, in: [52], 2013, pp. 705–709.
[41] P. Gonzalez, A. Griesmayer, A. Lomuscio, Verifying GSM-based business artifacts, in: C. A. Goble, P. P. Chen, J. Zhang (Eds.), 2012 IEEE 19th International Conference on Web Services, IEEE Computer Society, 2012, pp. 25–32.
[42] P. Gonzalez, A. Griesmayer, A. Lomuscio, Model checking GSM-based multi-agent systems, in: A. Lomuscio, S. Nepal,

F. Patrizi, B. Benatallah, I. Brandic (Eds.), ICSOC 2013 Workshops, volume 8377 of *LNCS*, Springer, 2013, pp. 54–68.
[43] I. Weber, J. Hoffmann, J. Mendling, Beyond soundness: on the verification of semantic business process models, Distributed and Parallel Databases 27 (2010) 271–343.
[44] D. Borrego, R. M. Gasca, M. T. Gómez-López, Automating correctness verification of artifact-centric business process models, Information & Software Technology 62 (2015) 187–197.
[45] N. Lohmann, Compliance by design for artifact-centric business processes, Inf. Syst. 38 (2013) 606–618.
[46] A. Queralt, A. Artale, D. Calvanese, E. Teniente, OCL-Lite: Finite reasoning on UML/OCL conceptual schemas, Data Knowl. Eng. 73 (2012) 1–22.
[47] C. Choppy, K. Klai, H. Zidani, Formal verification of UML state diagrams: a Petri net based approach, ACM SIGSOFT Soft. Eng. Notes 36 (2011) 1–8.
[48] R. Eshuis, Symbolic model checking of UML activity diagrams, ACM Trans. Softw. Eng. Methodol. 15 (2006) 1–38.
[49] F. J. Lucas, F. Molina, J. A. Toval Álvarez, A systematic review of UML model consistency management, Information & Software Technology 51 (2009) 1631–1645.
[50] R. V. D. Straeten, J. Simmonds, T. Mens, Detecting inconsistencies between UML models using Description Logic, in: D. Calvanese, et al. (Eds.), Description Logics, volume 81 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2003.
[51] G. Reggio, M. Leotta, F. Ricca, "Precise is better than light" a document analysis study about quality of business process models, in: EmpiRE 2011, IEEE, 2011, pp. 61–68.
[52] S. Basu, et al. (Eds.), Service-Oriented Computing - 11th International Conference, ICSOC 2013, volume 8274 of *LNCS*, Springer, 2013.