Verification and Validation of UML Artifact-centric Business Process Models

Montserrat Estañol[1], Maria-Ribera Sancho[1,2], and Ernest Teniente[1]

[1] Universitat Politècnica de Catalunya, Barcelona, Spain
{estanyol|ribera|teniente}@essi.upc.edu
[2] Barcelona Supercomputing Center, Barcelona, Spain

Abstract. This paper presents a way of checking the correctness of artifact-centric business process models defined using the BAUML framework. To ensure that these models are free of errors, we propose an approach to verify (i.e. there are no internal mistakes) and to validate them (i.e. the model complies with the business requirements). This approach is based on translating these models into logic and then encoding the desirable properties as satisfiability problems of derived predicates. In this way, we can then use a tool to check if these properties are fulfilled.

Keywords: artifact-centric BPM, UML, verification, validation

1 Introduction

Business process modeling (BPM) is a critical task in the business's definition, as these processes are directly involved in the achievement of an organization's goals. Business processes may be modeled following an artifact-centric approach which represents both the dynamic (i.e. the activities or tasks) and the structural (i.e. the data) dimensions of the process. Including the data in the model makes it possible to define precisely what each of the tasks does. This is why this approach has grown in importance in recent years.

It is essential to evaluate the correctness of these models as early as possible, to avoid the propagation of errors through the development of the business. Several research has been done on this topic [2, 6, 9, 17]. However, most of these specify the processes in different variants of logic, resulting in specifications that are complex and difficult to understand by the domain experts. They have also been proposed at a theoretical level: there is no tool that can perform the tests.

The correctness of an artifact-centric BPM can be assessed from two different perspectives. *Verification* ensures that the model is right, i.e. that it does not include contradictions or redundancies. *Validation* guarantees that we are building the right BPM, i.e., that the model fulfills the business requirements.

The main contribution of our work is to propose an approach to verify and validate an artifact-centric BPM specified in *BAUML* [4], 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. We also show the feasibility of our approach by using an implementation of an existing method that is able to carry out verification and validation tests.

To our knowledge, ours is the first approach able to check the correctness of artifact-centric BPMs in practice with reliable results since previous proposals always dealt with this problem at a theoretical level or bounded the number of objects considered. It is also the first one to handle together reasoning on class diagrams, state transition diagrams, activity diagrams and operation contracts.

This paper extends our previous work in several ways. In [8] we dealt with this problem at a theoretical level. In [14] we did not consider the notion of business artifact, nor state transition diagrams and activity diagrams during reasoning. In [4] we identified sufficient conditions over *BAUML* models which guarantee decidability of verification, and which can be applied to this work.

2 Motivation and Running Example

We base our work on the BALSA framework [12], which establishes four different dimensions that should be present in any artifact-centric business process model. They are the following. **Business Artifacts** represent the information required by the business, whose evolution we wish to track. **Lifecycles** are used to represent the evolution of an artifact during its life, from the moment it is created until it is destroyed. **Associations** establish the execution flow for services. **Services** (also known as tasks) are atomic units of work in the business process. As such, they make changes to artifacts by creating, updating and deleting them. Apart from artifacts, businesses keep data which may change but whose potential states are not relevant from the business's point of view. We will refer to it as *objects*.

In this paper we adopt the *BAUML* modeling approach [4], which represents the BALSA dimensions using UML and OCL: UML class diagrams for business artifacts; UML state transition diagrams for lifecycles; UML activity diagrams for associations, and OCL operation contracts for services.

As an example, consider the artifact-centric BPM of a city bicycle rental system. Figure 1 shows its UML class diagram. *Bicycle* is the only business artifact since we wish to track in the system the bicycle's evolution. A *Bicycle* may be in state *Available*, *InUse* or *Unusable* (we shortened the names for convenience; they should be called *AvailableBicycle*, etc.). The rest of the classes correspond to objects and specify the data required to rent a bicycle.

The textual constraints for Figure 1 are shown below.

1. *Bicycles* and *Users* are identified by their *id*. *AnchorPoints* by their *number*.
2. *inServiceSince* must be earlier or equal to *lastReturn*, *startTime*, and *date* in Unusable.
3. expectedReturn must be later or equal to *startTime*.

Figure 2 shows the lifecycle of the artifact *Bicycle*. When a *Bicycle* is registered it is *Available*. When a *User* picks it up to rent it, he may return it to its

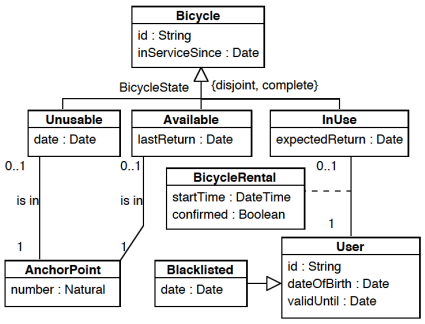

Fig. 1. Class diagram of our example

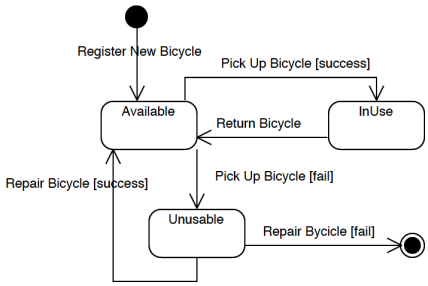

Fig. 2. State diagram of *Bicycle*.

anchor point if it is not in good shape and the bicycle is *Unusable*. Otherwise, it is *InUse*. When the user returns the bicycle, it is *Available* again. An *Unusuable* bicycle may be repaired, so that it is again *Available*. Otherwise, it is destroyed.

Figure 3 shows the activity diagram for transition *Register New Bicycle*. To do this, the bicycle has to be created first and then assigned to an anchor point.

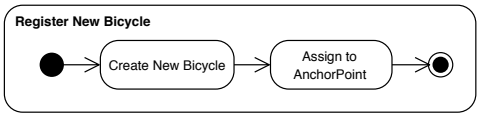

Fig. 3. Activity diagram of *Register New Bicycle*

The operation contracts for the tasks in Figure 3 are shown below. For the sake of simplicity, and without loss of generality, we leave out class attributes.

```
operation createNewBicycle(): Bicycle
post: Available.allInstances()->exists (b | b.oclIsNew() and
     result=b.oclAsType(Bicycle))
```

```
operation assignToAnchorPoint(b: Bicycle, ap: AnchorPoint)
pre: AnchorPoint.allInstances()->includes(ap) and ap.available->isEmpty()
     and ap.unusable->isEmpty()
post: ap.available = b.oclAsType(Available)
```

BAUML provides a high-level of abstraction that allows specifying artifact-centric BPMs from a technology-independent perspective, making these models understandable to model experts. However, it is very difficult to manually assess that the model is correct. For instance, is it possible to create an *unusable* bicycle? Will external event *Pick Up Bicycle* ever be executed? Are *Blacklisted* users forbidden from renting a bicycle? Automated reasoning techniques can aid the designer in this important task. This is the main goal of this paper.

3 Basic Concepts

This section formally presents the BALSA UML models that we use and the logic formalization in which they are translated in order to check their correctness.

The BAUML Model A BAUML model $\mathcal{B}$ is a tuple $\langle \mathcal{M}, \mathcal{S}, \mathcal{P}, \mathcal{T} \rangle$, describing the four dimensions of the BALSA framework:

Class Diagram: $\mathcal{M}$ is a UML class diagram, in which some classes represent (business) artifacts. 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. These subclasses must fulfill the covering and disjointness constraints (i.e. the artifact must exactly have one of the subclasses type at a certain point in time.) 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 (defined in OCL) integrity constraints, which we denote as $\mathcal{O}$.

State Transition Diagrams: $\mathcal{S}$ is a set of UML state transition 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 $S_{\mathtt{A}} = \langle V, V_0, E, T \rangle$, where V is a set of states, $V_0 \subseteq V$ is the set of initial states, E is a set of events, and $T \subseteq V \times OCL_{\mathcal{M}} \times E \times C \times V$ is a set of transitions between pairs of states, where $OCL_{\mathcal{M}}$ is an OCL condition over $\mathcal{M}$ and C is a tag on the result of the execution of the event in E. The states V of $S_{\mathtt{A}}$ exactly mirror the subclasses of $\mathtt{A}$.

Transitions have the following form (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`. The transition will take place if $OCL_{\mathcal{M}}$ is true when the external event is received. The execution of the event results in tag C (as we shall see, its possible values are `success` and `fail`).

$OCL_{\mathcal{M}}$ is an OCL boolean expression over $\mathcal{M}$. `ExternalEvent(`$a_1, ..., a_n$`)` must appear at least in a transition of the state transition diagram of each artifact a_i. The execution of external events and the tags C resulting from this execution are driven by activity diagrams.

Activity Diagrams: $\mathcal{P}$ is a set of UML activity diagrams, such that for every state transition diagram $S{=}\langle V, V_0, E, T \rangle \in \mathcal{S}$, 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 tasks (TASKS(P_ε)). *Initial* and *final* nodes indicate the points where the

activity diagram flow begins and ends, respectively. *Gateways* are used to control the sequence flow, they include *decision nodes* and *merge nodes*. Finally, each task is associated to an *operation contract*, which expresses a precondition on the executability of the task, and a postcondition describing its effect, both formalized in terms of OCL queries over $\mathcal{M}$.

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, and g corresponds to an OCL condition over $\mathcal{M}$. Similarly, we only allow c over $f \in F$ such that $f = \langle n_s, g, c, n_t \rangle$ and $n_t \in \text{FINAL}(P_\varepsilon)$.

We make the following assumptions: decision nodes have one incoming flow and more than one outgoing flow; merge nodes have more than one incoming flow and exactly one outgoing flow; tasks 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. In addition, the external event must, at the end of its execution, bring the artifact to the target state of the transition in the state machine diagram.

During the execution of an activity diagram the constraints may be violated, but they must be met at the end of the execution, otherwise the transition in the state transition diagram does not take place and the changes are rolled back.

Tasks: $\mathcal{T}$ is a set of atomic tasks, each of which has an OCL operation contract. A task can only be executed when the current information base satisfies its precondition and, once executed, it 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 transition diagram $S_{\mathtt{A}}$, also considering all activity diagrams related to $S_{\mathtt{A}}$. 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 $S_{\mathtt{A}}$.

Logic Formalization 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 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 Verification and Validation of BAUML Models

Given a BAUML model, our goal is to ensure that it is correct (verification) and that it satisfies the user requirements (validation). To do so, we need to transform the model into the logic described in section 3. After this, we will obtain a set of derivation rules and conditions (a schema) representing the BAUML model. A desirable property of the model will be then tested by checking the satisfiability of a derived predicate.

The work we present here clearly differs from [14], 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.

4.1 Translation Algorithms

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 set of classes and associations. To determine if a class or association is read-only or read-write, it is only necessary to examine the postcondition of all the tasks as described in [14]. 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 not include the time t and will be treated as base predicates.

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:

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

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, `Bicycle` is encoded as:

$$Bicycle(b, t) \leftarrow addBicycle(b, t_1) \wedge time(t) \wedge \neg deletedBicycle(b, t_1, t) \wedge t_1 \leq t,$$

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

```
// Step 1: Creating rules for read/write classes and associations
r := ∅
for all c ∈ CLASSES(M) do
    if c is created in P ∧ c is not deleted in P then
        r := r ∪ {C(p̄, t) ← addC(p̄, t1) ∧ time(t) ∧ t ≥ t1}
    else if c is created in P ∧ c is deleted in P then
        r := r ∪ {C(p̄, t) ← addC(p̄, t1) ∧ ¬deletedC(p̄j, t1, t) ∧ t ≥ t1 ∧ time(t)}
        r := r ∪ {deletedC(p̄j, t1, t2) ← delC(p̄j, t) ∧ time(t1) ∧ time(t2) ∧ t ≤ t2 ∧ t > t1}
    end if
end for
for all a ∈ ASSOCIATIONS(M) do
    if a is created in P ∧ a is not deleted in P then
        r := r ∪ {A(p̄, t) ← addA(p̄, t1) ∧ time(t) ∧ t ≥ t1}
    else if a is created in P ∧ a is deleted in P then
        r := r ∪ {A(p̄, t) ← addA(p̄, t1) ∧ ¬deletedA(p̄j, t1, t) ∧ t ≥ t1 ∧ time(t)}
        r := r ∪ {deletedA(p̄j, t1, t2) ← delA(p̄j, t) ∧ time(t1) ∧ time(t2) ∧ t ≤ t2 ∧ t > t1}
    end if
end for
// Step 2: Translate integrity constraints
icSet := translateIC(O)
for all condition cond ∈ icSet do
    cond := cond + {∧validState(t)}
end for
taskRules := ∅
// Step 3: Generate rules for class and association creation and deletion for every task
for all t ∈ T do
    resRules := translateTask(t)
    taskRules := taskRules ∪ resRules
end for
// Step 4: Generate necessary rules and conditions to ensure correct execution order
taskRules := taskRules ∪ generateConstraintsTaskExecution(B)
return ⟨r, icSet, taskRules⟩
```

whereas `User` is encoded as $User(u)$. `Bicycle` is a derived predicate created and deleted by some of the tasks. On the other hand, `User` 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, following [15], 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 `Bicycle` indicates that a Bicycle must have one of its subclasses' type. Then the condition: $\leftarrow Bicycle(b,t) \wedge \neg IsKindOfBicycle(b,t) \wedge validState(t)$ states that there cannot be a bicycle which has not any of its subtypes (predicate $IsKindOfBicycle$), where $IsKindOfBicycle$ is a derived predicate from $InUse$, $Available$ and $Unusuable$. This condition only applies when there are no transitions taking place, indicated by predicate $validState$.

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($task$)

$rules := \emptyset$
$prevRules := getContextPreviousTasks(task, t)$ // 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 $ruleFragment \in prevRules$ **do**
  for all $el \in createList$ **do**
    $r := addEl(\overline{p}, t) \leftarrow task(\overline{p}, \overline{x}, t) \wedge pre_{task}(t-1) \wedge time(t) \wedge ruleFragment$
    $rules := rules \cup r$
  end for
  for all $el \in delList$ **do**
    $r := delEl(\overline{p}_j, t) \leftarrow task(\overline{p}_j, \overline{y}, t) \wedge pre_{task}(t-1) \wedge time(t) \wedge ruleFragment$
    $rules := rules \cup r$
  end for
  $rules := rules \cup \{task'(p_a, t) \leftarrow task(p_a, \overline{z}, t) \wedge pre_{task}(t-1) \wedge time(t) \wedge ruleFragment\}$
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. In [14], 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 [15]) 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

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
```

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:

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 *Assign to Anchor Point*, we have the following rules:

$$addAvailableIsIn(b,a,t) \leftarrow assignToAnchPoint(a,b,t) \wedge AnchorPoint(a)$$
$$\wedge precondAssToAP(a,t-1) \wedge Bicycle(b,t) \wedge createNewBicycle'(b,t-1)$$
$$assignToAnchPoint'(b,t) \leftarrow assignToAnchPoint(a,b,t) \wedge AnchorPoint(a)$$
$$\wedge precondAssToAP(a,t-1) \wedge Bicycle(b,t) \wedge createNewBicycle'(b,t-1)$$

The task creates an instance of the *available is in* association. It has a precondition which must be true at $t-1$, and its translation appears in the derivation rule of $addAvailableIsIn$. In addition to this, the body of the rule includes the predicate $createNewBicycle'$, that guarantees that the previous operation (*Create New Bicycle*) 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 *Create New Bicycle* and *Assign to Anchor Point* we have the following condition and derivation rule: $\leftarrow createNewBicycle(b,t) \wedge \neg assignToAnchorPoint'(b,t+1)$.

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: $validState(t) \leftarrow assignToAnchorPoint'(b, t)$.

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 corresponding guard condition is met.

4.2 Verification and Validation Tests

After applying the translation described in the previous section, we are now interested in checking certain properties to guarantee the model's correctness. All tests are represented as checking the satisfiability of a derived predicate. Any satisfiability checking method that is able to deal with negation of derived predicates can be used to validate the schema. Note that we use the translation of our whole running example to perform the tests.

Verification Tests. The goal of verification tests is to ensure that there are no inherent contradictions or mistakes in the model. They can be generated and performed automatically without requiring intervention from the modeler.

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. Logically, it only makes sense to apply the tests to read-write classes and associations. The general form

of the test is the following, where *el* is the name of the class or association: $livelinessTestEl() \leftarrow el(\overline{p}, t) \wedge validState(t)$. Remember that *validState* is a derived predicate generated by the last task that executes in a transition. For instance, to test the liveliness of *Bicycle*, we would define the following derivation rule: $livelinessTestBicycle() \leftarrow Bicycle(b, t) \wedge validstate(t)$.

The **applicability test** will check whether a certain task can be executed, that is, if the necessary requirements for its execution are met. The test will have the following form, for task $task_i$: $applicabilityTask() \leftarrow pre_{task}(\overline{y}, t) \wedge task'_{i-1}(p_a, t)$.

The **executability test** will check if a certain task can be executed. It is particularly useful for those activity diagrams with decision nodes to ensure that all paths can be taken. The test will have the following form: $executabilityTask() \leftarrow task'(p_a, t)$. Notice that it is equivalent to checking if the predicate $task'$ can be generated, as $task'$ represents precisely the successful execution of *task*. For instance, to check the executability of task *Confirm Return*, we would run the following test: $execConfirmReturn() \leftarrow confirmReturn'(b, t)$.

Validation Tests. On the other hand, validation tests ensure that the model is aligned to the user requirements. In the general case, validation tests require the intervention from the user and thus cannot be generated automatically from the model. An interesting validation test in our example would be to check if a blacklisted user can rent a bicycle: $blacklistUserRent() \leftarrow Blacklisted(u) \wedge BicycleRental(b, u, i, t) \wedge validState(t)$. The *validState* predicate is needed to ensure that the BicycleRental is not deleted before the end of a transition.

5 Implementing our Approach within SVTe

We have studied the feasibility of our approach by using an existing tool, SVTe, that is able to perform the tests described previously. This tool uses the CQC_E method [16] which is 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, tries to obtain a set of base facts that satisfy the goal without violating any of the 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 assured 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.

Figure 4 shows the result of some of the previous tests: i.e. the outcomes of the bicycle liveliness test (top), the executability test for task *Confirm Return* (middle) and the validation test (bottom). Notice that all the tests execute

successfully, that is, there exists an instantiation of the database schema (i.e. the translation of our model) that fulfills the given goals. The tool shows the set of base predicates (corresponding to tasks and read-only classes and associations) that prove the satisfiability of the test. However, notice that although the last test also gives a positive result, it is not what should be: blacklisted users should *not* be allowed to rent bicycles. The reason for this is that an integrity constraint is missing in the class diagram, forbidding blacklisted users to have bicycle rentals.

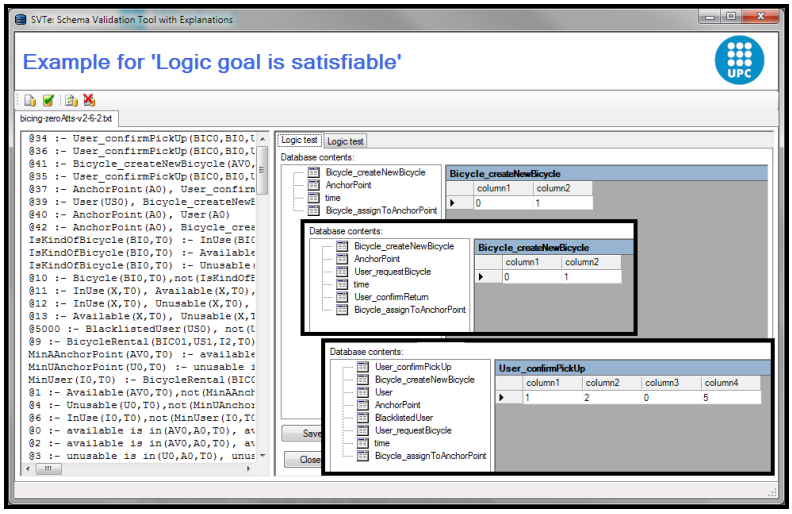

Fig. 4. Screenshots from SVTe showing the results of the test.

6 Related Work

We examine validation and verification in two different areas related to our work: artifact-centric business process models and UML diagrams.

Several approaches to reasoning on artifact-centric BPM use data-centric dynamic systems (DCDSs), grounded on logic, as the basis for reasoning [2, 3, 1]. [2] uses a relational database to represent the data, together with a set of condition-action rules and actions defined in logic. In contrast, [1] uses a Knowledge and Action Base defined in a variant of Description Logics to represent this data. Similarly, [3] maps an ontology to a DCDS in order to verify certain temporal properties expressed in a variant of μ-calculus.

Similarly, [6] represents artifacts using a set of variables, which are updated by services defined by pre and postconditions in first-order logic. They check whether the resulting model fulfills a set of properties defined in LTL-FO, which is not as powerful as μ-calculus.

All these works represent artifact-centric business process models in languages derived from logic. Consequently, the models under consideration are formal, but they are not practical for business people. Moreover, they have been proposed at a theoretical level and do not have a tool that implements them.

In contrast, the Guard-Stage-Milestone (GSM) approach provides a business-friendly representation of artifact-centric business processes. [17] studies the decidability of verification over GSM models by translating them into a DCDSs. However, the presented results are theoretical, as there is no tool that can actually perform the reasoning. [11] presents a system to model and execute artifact systems. However, to our knowledge, the system is limited to simulating the behavior of the model given certain data and this is different to our work in this paper. [10] performs 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.

Similarly to our work, [18] performs verification over process models considering the meaning of the tasks. These are annotated with preconditions and effects defined in logic, and use an ontology to define the underlying data. Time is not considered explicitly, which only allows for analysis of the current state of the system, whereas in our case we can analyze the system's evolution.

On the other hand, most of the proposals for reasoning on UML models deal with only one diagram. For instance, [15] focuses on the class diagram, [5] handles state-machine diagrams, and [7] focuses on activity diagrams. As far as approaches examining various UML diagrams, [13] offers a systematic literature review 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. [14] handles both the class diagram and the operation contracts, but it does not consider state transition nor activity diagrams.

7 Conclusions

We have presented a way of validating and verifying artifact-centric business process models defined using the *BAUML* framework. This framework provides us with a set of models which can be defined and are understandable by the modelers. Checking the correctness of these models as early as possible is important to avoid the propagation of errors to the execution stage of the process. These errors can result from mistakes in the models themselves (e.g. contradictions) or errors in the sense that the models do not fulfill the business requirements.

To ensure that they are free of errors, we translate the BAUML models into logic and encode the desirable properties as derived predicates. We can then use an existing tool to check if the properties are fulfilled. To the best of our knowledge, there is no other proposal that is able to check the correctness of artifact-centric BPMs with reliable results.

We are aware that, in some instances, the tool may not perform efficiently, and even not terminate for some tests, due to the temporal cost of the search for a solution and its potential infinity. Improving the efficiency of the tool and the translation of parallelism is left as further work.

Acknowledgments This work has been partially supported by the Ministerio de Ciencia e Innovación under project TIN2011-24747 and by UPC.

References

1. Bagheri Hariri, B., et al.: Verification of description logic knowledge and action bases. In: Raedt, L.D., et al. (eds.) ECAI. Frontiers in Artificial Intelligence and Applications, vol. 242, pp. 103–108. IOS Press (2012)
2. Bagheri Hariri, B., et al.: Verification of relational data-centric dynamic systems with external services. In: PODS. pp. 163–174. ACM (2013)
3. Calvanese, D., Giacomo, G.D., Lembo, D., Montali, M., Santoso, A.: Ontology-based governance of data-aware processes. In: Krötzsch, M., Straccia, U. (eds.) RR. LNCS, vol. 7497, pp. 25–41. Springer (2012)
4. Calvanese, D., Montali, M., Estañol, M., Teniente, E.: Verifiable UML artifact-centric business process models. In: Li, J., Wang, X.S., Garofalakis, M.N., Soboroff, I., Suel, T., Wang, M. (eds.) CIKM 2014. pp. 1289–1298. ACM (2014)
5. Choppy, C., Klai, K., Zidani, H.: Formal verification of UML state diagrams: a Petri net based approach. ACM SIGSOFT Soft. Eng. Notes 36(1), 1–8 (2011)
6. Damaggio, E., Deutsch, A., Vianu, V.: Artifact systems with data dependencies and arithmetic. ACM Transactions on Database Systems 37(3), 1–36 (Aug 2012)
7. Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Trans. Softw. Eng. Methodol. 15(1), 1–38 (2006)
8. Estañol, M., Sancho, M.R., Teniente, E.: Reasoning on UML data-centric business process models. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) Service-Oriented Computing. LNCS, vol. 8274, pp. 437–445. Springer (2013)
9. Gerede, C.E., Su, J.: Specification and verification of artifact behaviors in business process models. In: Krämer, B.J., Lin, K.J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 181–192. Springer (2007)
10. Gonzalez, P., Griesmayer, A., Lomuscio, A.: Model checking gsm-based multi-agent systems. In: Lomuscio, A., Nepal, S., Patrizi, F., Benatallah, B., Brandic, I. (eds.) ICSOC 2013 Workshops. LNCS, vol. 8377, pp. 54–68. Springer (2013)
11. Heath, F.T., et al.: Barcelona: A design and runtime environment for declarative artifact-centric BPM. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 705–709. Springer (2013)
12. Hull, R.: Artifact-centric business process models: Brief survey of research results and challenges. In: Meersman, R., Tari, Z. (eds.) OTM 2008. LNCS, vol. 5332, pp. 1152–1163. Springer (2008)
13. Lucas, F.J., Molina, F., Álvarez, J.A.T.: A systematic review of UML model consistency management. Information & Software Technology 51(12), 1631–1645 (2009)
14. Queralt, A., Teniente, E.: Reasoning on UML conceptual schemas with operations. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. pp. 47–62. LNCS, Springer (2009)
15. Queralt, A., Teniente, E.: Verification and validation of UML conceptual schemas with OCL constraints. ACM Trans. Softw. Eng. Methodol. 21(2), 13 (2012)
16. Rull, G., Farré, C., Teniente, E., Urpí, T.: Providing explanations for database schema validation. In: Bhowmick, S.S., Küng, J., Wagner, R. (eds.) DEXA. LNCS, vol. 5181, pp. 660–667. Springer (2008)
17. Solomakhin, D., et al.: Verification of artifact-centric systems: Decidability and modeling issues. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) Service-Oriented Computing. LNCS, vol. 8274, pp. 252–266. Springer (2013)
18. Weber, I., Hoffmann, J., Mendling, J.: Beyond soundness: on the verification of semantic business process models. Distributed and Parallel Databases 27(3), 271–343 (2010)