Formalization of fUML: An Application to Process Verification

Yoann Laurent, Reda Bendraou, Souheib Baarir, Marie-Pierre Gervais

Formalization of fUML: an Application to Process Verification

Yoann Laurent[1], Reda Bendraou[1], Souheib Baarir[1,2], and Marie-Pierre Gervais[1,2]

[1] Sorbonne Universites, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France
[2] Universite Paris Ouest Nanterre La Défense, F-92001, Nanterre, France
{first.last}@lip6.fr

Abstract. Much research work has been done on formalizing UML Activity Diagrams for process modeling to verify different kinds of soundness properties (deadlock, unreachable activities and so on) on process models. However, these works focus mainly on the control-flow aspects of the process and have done some assumptions on the precise execution semantics defined in natural language in the UML specification. In this paper, we define a first-order logic formalization of fUML (Foundational Subset of Executable UML), the official and precise operational semantics of UML, in order to apply model checking techniques and therefore verify the correctness of fUML-based process models. Our formalization covers the control-flow, data-flow, resources, and timing dimensions of processes in a unified way. A working implementation based on the Alloy language has been developed. The implementation showed us that many kinds of behavioral properties not commonly supported by other approaches and implying multiple dimensions of the process can be efficiently checked.

Keywords: Formalization, Model-checking, fUML, Alloy

1 Introduction

With the increasing complexity of processes, whatever their kind (i.e. business, software, medical, military), process modelers need adequate tooling support to simulate and to ensure their correctness before to use them in a real context. Recent studies reported a significant rate of errors in industrial process models [1,2]. Typical errors are deadlocks, unreachable activities, inefficient use of resources and timing problems.

UML Activity Diagrams (AD) are well-known for describing dynamic behavior and have been extensively used as a process modeling language (PML) [3,4,5]. UML is a standard with a good tooling support and AD allow the expression of most of the workflow patterns as identified by [6]. In order to verify UML-based process models, current state-of-the-art has already proposed some formalizations of the way an AD operates [7,8,9]. These formalizations are mandatory to apply model-checking techniques enabling an exhaustive and an automatic verification of their models. However, in the current UML specification [10], the operational semantics remains unclear, imprecise and ambiguous. This semantics is explained in *natural language* and dispersed through the specification. Due to

this fact, the authors of these formalizations have done some assumptions on the precise operational semantics. As a consequence, the same process might be executed and verified differently from one tool into another, implying a *gap* between the semantics adopted respectively by each tool.

Recently, the OMG released fUML (Semantics of a Foundational Subset for Executable UML Models) [11], a new standard that precisely defines the execution semantics for a subset of UML 2.3 in a form of an *Execution Model* implemented in a virtual machine. However, even if the semantics is now clear and not subject to human interpretation, the semantics is not given in a formal way but in the form of pseudo Java-code. Therefore, it is not possible to straightforwardly apply model checking techniques.

In this paper, we define a formal model of fUML using first-order logic (FOL). The formalization addresses a subset of fUML encompassing only the concepts required for process modeling as identified in [3]. Current formalizations proposed in the literature focus mainly on the control-flow aspects of the process preventing to verify many kinds of properties related to data-flow, resources and timing constraints [12]. Therefore, our formalization covers both control and data-flow of the process through the use of the AD notations, and takes into account the associated organizational data such as resources and timing constraints. Then, we implement our formalization by using the Alloy modeling language [13] and we build a graphical tool on top of the implementation. The result of the verification is then graphically displayed on the process.

The rest of this paper is organized as follows. Section 2 presents the fUML standard and its execution semantics. Section 3 presents our FOL formalization of fUML. Section 4 gives an overview of interesting properties supported by our formalism. The implementation of the formalization and a case study are presented in Section 5. Finally, related work is addressed in Section 6 and Section 7 concludes by sketching some future perspectives of this work.

2 fUML

fUML is an OMG standard that precisely defines the execution semantics of a subset of UML 2.3. The standard defines a virtual machine in the form of pseudo Java-code, enabling compliant fUML models (i.e., UML models using only elements comprised in the fUML subset) to be executed. It can be decomposed in three main parts: (i) the abstract syntax represented by a subset of UML, mainly composed by the `Class Diagram` and most of the `Activity Diagram`; (ii) the Execution Model which defines the execution semantics of the abstract syntax and (iii) the model library which defines primitive types and behaviors (e.g. integer type and addition between two integers). In this section we give an overview of the Execution Model.

2.1 Execution model

The Execution Model is itself a model, written in fUML, that specifies how fUML models are executed. The execution semantics adopted by fUML is quite similar to Coloured Petri Nets (CPN) and is based on the principle of offering and consuming object or control tokens between the different activity constituents.

To illustrate this concept, Figure 1 shows a simple process represented with an AD composed of one `InitialNode`, two `Action` nodes and an

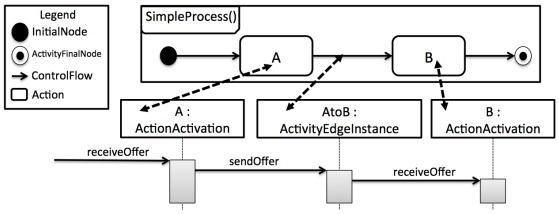

Fig. 1: Call between the UML elements within the Execution Model

`ActivityFinalNode`. Each of these nodes are connected with a `ControlFlow` edge. The sequence diagram shows the corresponding calls between the nodes in the Execution Model. The diagram is a simplified version of what really happens during the execution and focuses on the interaction between elements. `ActionActivation` and `ActivityEdgeInstance` are the instantiation of the corresponding abstract syntax.

When the fUML virtual machine invokes this activity, it starts by inserting a token in each `InitialNode`. Then, the nodes with a token (i.e., the `InitialNode` in our example) *fire* (i.e., execute their own behavior) and *sendOffer* on each of their outputs `ControlFlow`. The `ControlFlow` is then able to call on its target node A to *receiveOffer*. When the node "`A`" receives an offer, it first checks if the prerequisites for its execution are satisfied, if yes, takes the offered tokens from the input control flows and fires. At the end of the firing operation, the node directly *sendOffer* on its outputs `ControlFlow`. The execution of an activity is then an extended chain of *sendOffer-receiveOffer-fire-sendOffer* calls between the activity constituents. When an `ActivityFinalNode` is reached or if there are no nodes still able to execute, the activity is terminated. Each abstract syntax element of an activity diagram has its own semantics. For example, a `DecisionNode` will offer a token only on one of its output edges determined during its fire execution.

Similarly to CPN, tokens positions and contents on the system represent the actual execution state. Since the goal of this paper is mainly on the verification of fUML-based process models, we focus on the formalization of the tokens game between the semantics elements of an UML AD. Note that we call "tokens game" the rules and conditions on which a token may pass through an edge to another node to form a complete execution.

3 Formalization of the fUML tokens game

In the following, we present our formalization of the fUML tokens game by defining the syntax of the langage and its semantics.

3.1 Syntax

Figure 2 shows an excerpt of the UML class diagram handled by our formalization. Here we concentrate only on those elements that are part of the fUML standard and useful for the definition of a process as identified in [3].

An `Activity` is a graph with three kinds of `ActivityNodes`: `ObjectNode`, `ControlNode` and `ExecutableNode`. An `ObjectNode` represents the data in a process, a `ControlNode` coordinates the execution flow and an `ExecutableNode` represents a node that can be executed, i.e. process action. There are two kinds

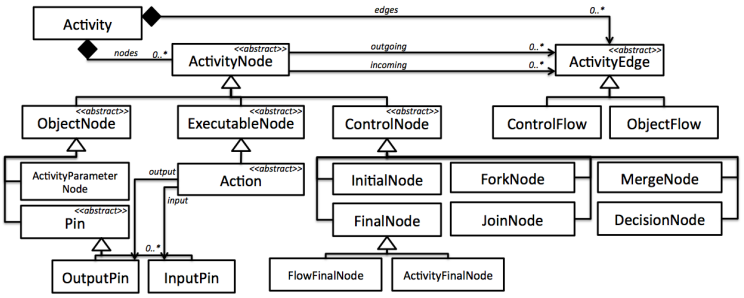

Fig. 2: Excerpt of the fUML Activity Diagram meta-model handled by our formalization

of `ActivityEdge` to link the nodes: `ObjectFlow` and `ControlFlow`. `ObjectFlow` edges connect `ObjectNode`s and can have data passing along it. `ControlFlow` edges constrains the desired order of execution of the `ActivityNode`s. `ControlNode` can be used for parallel routing (`ForkJoin`), conditional routing (`DecisionNode`), synchronization (`JoinNode`) and merging multiple alternate flows (`MergeNode`). `InitialNode` and `AcitivityFinalNode` represent respectively the beginning and the end of an `Activity` while `FlowFinal` terminates a flow. `InputPin` and `OutputPin` are anchored to `Action`s to represent the required *input* data and the *output* data produced by the action. Similarly, an `Activity` can have multiple `ActivityParameterNode` to represent its data input and output. Thus, an `Activity` can represent a process by defining a coordinated sequencing set of actions using both control- and data-flow.

Formally, we consider three basic elements: $Control$, $Executable$, and $Object$.

- $Control = \{fork, join, decision, merge, initial, activityFinal, flowFinal\}$ represents the different `ControlNode` types,
- $Executable = \{action\}$ represents the `ExecutableNode` type,
- $Object = \{activityParameter, outputPin, inputPin\}$ represents the `ObjectNode` type,
- $Types = Control \cup Executable \cup Object$ represents the set of all types.

Thus, we introduce the notion of diagram as a vertex-labeled graph:

Definition 1. *A **Diagram** is a tuple $D = (V, E, Types, lab, lower, upper)$ such that:*

- *V is the set of vertices.*
- *$E \subseteq V \times V$ is the set of edges.*
- *$lab : V \mapsto Types$ is the labeling function associating to each vertice $v \in V$ a $Types$.*
- *$lower/upper : V \mapsto \mathbb{N} \cup \{\epsilon\}$ are functions that return, respectively, the lower and upper multiplicity of an object node.*

$$lower(v) \stackrel{def}{=} \begin{cases} n \in \mathbb{N} & \text{if } lab(v) \in Object \\ \epsilon & \text{otherwise} \end{cases}$$

The function upper *has the same definition.*

For a Diagram $D = (V, E, Types, lab, lower, upper)$, we introduce the following auxiliary functions that will help us to define formally an AD.

- $Vlab : Types \mapsto 2^V$ is the function that returns all the vertices of a type:

$$Vlab(t) \stackrel{def}{=} \{v \in V \mid lab(v) = t\}$$

- $incoming/outgoing : V \mapsto 2^E$ are functions that return, respectively, the incoming and outgoing edges of a node:

$$incoming(v) \stackrel{def}{=} \{(a, v) \in E \mid \forall a \in V\}$$

$$outgoing(v) \stackrel{def}{=} \{(v,a) \in E \mid \forall a \in V\}$$

- *source/target* : $E \mapsto V$ are functions that return, respectively, the source and target of an edge.

$$source(e) \stackrel{def}{=} \{s \in V \mid e = (s,t), \forall t \in V\}$$
$$target(e) \stackrel{def}{=} \{t \in V \mid e = (s,t), \forall s \in V\}$$

- *input/output* : $V \mapsto 2^V$ are functions that return, respectively, the input and output pins of an action node.

$$input(v) \stackrel{def}{=} \begin{cases} in \subseteq V & \text{if } lab(v) = action \wedge \forall v' \in in, lab(v') = inPin \wedge \\ & ((v,v') \in E \vee (v',v) \in E) \\ \emptyset & \text{otherwise} \end{cases}$$

$$output(v) \stackrel{def}{=} \begin{cases} out \subseteq V & \text{if } lab(v) = action \wedge \forall v' \in out, lab(v') = outPin \wedge \\ & ((v,v') \in E \vee (v',v) \in E) \\ \emptyset & \text{otherwise} \end{cases}$$

Now, we can define the notion of **Activity Diagram**. Actually, it is a **Diagram** with some additional structural constraints.

Definition 2. *An **Activity Diagram** is a Diagram, AD = (V, E, Types, lab, lower, upper), with the following additional constraints:*

- *No node is disconnected:* $\forall v \in V, incoming(v) \neq \emptyset \vee outgoing(v) \neq \emptyset$.
- *The source and target of an edge are different:* $\forall e \in E, source(e) \neq target(e)$.
- *Initial nodes have no incoming edge:*
$$\forall v \in Vlab(initial) : incoming(v) = \emptyset$$
- *All activity final and flow final nodes have no outgoing edge:*
$$\forall v \in (Vlab(flowFinal) \cup Vlab(activityFinal)) : outgoing(v) = \emptyset$$
- *Pin nodes are connected to a unique pin node:*
$$\forall v \in Vlab(inPin) : |incoming(v) = \{(a,v) \in E \mid \forall a \in V\}| = 1 \wedge$$
$$\forall v \in Vlab(inPin), \forall e \in incoming(v), \forall a \in source(e) : lab(a) = outPin \wedge$$
$$\forall v \in Vlab(outPin) : |outgoing(v) = \{(v,a) \in E \mid \forall a \in V\}| = 1 \wedge$$
$$\forall v \in Vlab(outPin), \forall e \in outgoing(v), \forall a \in target(e) : lab(a) = inPin$$
- *The lower bound of an Object node is not greater than its upper bound:*
$$\forall v \in Vlab(Object) : lower(v) \leq upper(v)$$
- *The upper bound of an Object node is at least equal to one:*
$$\forall v \in Vlab(Object) : upper(v) \geq 1$$

Generally, a process is characterized by two main parts: the workflow and the associated organizational information. Here, the workflow is represented using UML AD. The organizational information is attached directly to the actions to give insight about the execution. This information is domain dependent. For instance, *software* processes might focus on the number of agents and their skills, while *medical* processes require instrumentation and drugs. Therefore, we define a process as an AD extended with most commonly used organizational information: resources and time. Note that the definition can be easily extended to take into account other domain dependent information.

Definition 3. *A **Process** is a tuple* $P = (V, E, Types, lower, upper, lab, Resource, Use, Timing)$ *where:*

- $(V, E, Types, lab, lower, upper)$ *forms an **Activity Diagram** s.t.:*
  - V *contains at least one initial node:* $\exists n \in V, lab(v) = initial$.
  - V *contains at least one activity final node:* $\exists n \in V, lab(v) = activityfinal$.
- *Resource is a finite set of resources,*

– $Use : V \mapsto 2^{Resource}$ *is the function that maps each action to a set of resources:*

$$Use(v) \stackrel{def}{=} \begin{cases} r \subseteq Resource & \text{if } lab(v) = action \\ \emptyset & \text{otherwise} \end{cases}$$

– $Timing : V \mapsto \mathbb{N} \cup \{\epsilon\}$ *is the function that associates to each action a time to perform it:*

$$Timing(v) \stackrel{def}{=} \begin{cases} v \in \mathbb{N} & \text{if } lab(v) = action \\ \epsilon & \text{otherwise} \end{cases}$$

3.2 Semantics

The semantics of our model follows the newly defined fUML standard [11]. We formalize the way the tokens transit between the nodes and edges that compose an fUML AD model. Moreover, to be able to reason about the timing constraints of the process, we extend the formalization with discrete clocks representing the time spent during the process execution.

The semantics of our formalism is based on the notions of *states*, *enabling* and *firing* of transitions (similar to those used in CPN).

State. A state formalizes the configuration on which the process is at any time of its execution.

Definition 4 (State). *A state of a process* $P = (V, E, Types, lower, upper, lab, Resource, Use, Timing)$ *is a tuple* $s = (m, gc, lc)$ *such that:*

– $m : V \cup E \mapsto \mathbb{N}$ *is the function, called marking, that associates to each node and edge a natural number.*
  - *for* $v \in V$, $m(v)$ *is the number of tokens,*
  - *for* $e \in E$, $m(e)$ *is the number of offers.*

– $gc \in \mathbb{N}$ *is the global discrete clock representing the current time spent on the process,*

– $lc : V \mapsto \mathbb{N} \cup \{\epsilon\}$ *is the local discrete clock representing the current time spend on a given action:*

$$lc(v) \stackrel{def}{=} \begin{cases} n \in \mathbb{N} & \text{if } lab(v) = action \\ \epsilon & \text{otherwise} \end{cases}$$

The set of all states of a process P is noted ***States****.*

Definition 5 (Initial State). *An initial state* $s_0 = (m_0, gc_0, lc_0)$ *of the system is always defined as follows:*

– *All nodes own 0 token, except (i) the initial nodes which start with 1 token and (ii) the input activity parameter node which start with a number of tokens that varies between its lower and upper bounds:*

$$m_0(v) = \begin{cases} 1 & \text{if } lab(v) = initial \\ n \in \{lower(v), ..., upper(v)\} & \text{if } lab(v) = activityParameter \wedge incoming(v) = \emptyset \\ 0 & \text{otherwise} \end{cases}$$

– *The global clock is initialized to zero:* $gc_0 = 0$.

– *Local clocks are initialized to zero:*

$$lc_0(v) = \begin{cases} 0 & \text{if } lab(v) = action \\ \epsilon & \text{otherwise} \end{cases}$$

Transition. The dynamic of a process, i.e. its execution, is defined through the notion of *transition*. To move from a state to another one, a transition is first *enabled* then *fired*. Therefore, the *enabling* notion corresponds to a pre-condition while the *firing* notion corresponds to a post-condition. We first define the enabling notion, and then formalize the firing concept.

Transition enabling. A transition is said to be *enabled* when some preconditions are met (to allow the firing of the transition). By abstracting the way the fUML Execution Model executes an AD, two cases can be distinguished: (i) a node is ready to execute; (ii) a node is ready to terminate. In our framework, these are represented by predicates **eStart** and **eFinish**, respectively. Also, note that the system can progress through time elapsing using the **eTime** predicate.
Let us consider a process $P = (V, E, Types, lower, upper, lab, Resource, Use, Timing)$ and a state $s = (m, gc, lc)$. To simplify our notation, we assume that s is implicitly available in the following enabling predicates.

1. **eStart** is the predicate that determines if a node v is ready to be executed. Its formal definition relies on the following auxiliary predicates.
   - The first condition corresponds to check if the node is not already executing (not owning tokens) and have incoming edges:
   $$pAll(v) \stackrel{def}{=} (m(v) = 0) \wedge incoming(v) \neq \emptyset$$
   - An *activity* node needs an offer on all of its incoming edges:
   $$pNode(v) \stackrel{def}{=} pAll(v) \wedge \bigwedge_{e \in incoming(v)} (m(e) > 0)$$
   - An *action* node extends the behavior with input and output pins, so the number of offers on its incoming pins are also checked:
   $$pAction(v) \stackrel{def}{=} pNode(v) \wedge \bigwedge_{e \in incoming(input(v))} (m(e) \geq lower(v))$$
   - Unlike other activity nodes, a *merge* node needs at least one of its incoming edge to have an offer:
   $$pMerge(v) \stackrel{def}{=} pAll(v) \wedge \bigvee_{e \in incoming(v)} (m(e) > 0)$$

   Then, the enabling test corresponds to:
   $$eStart(v) \stackrel{def}{=} \begin{cases} pAction(v) & \text{if } lab(v) = action \\ pMerge(v) & \text{if } lab(v) = merge \\ pNode(v) & \text{otherwise} \end{cases}$$
2. **eFinish** is the predicate that determines if a node is ready to terminate and relies on the following auxiliary predicates.
   - The node must own tokens:
   $$haveTokens(v) \stackrel{def}{=} (m(v) > 0)$$
   - An *action* must have its local clock incremented at least until its defined timing:
   $$pTiming(v) \stackrel{def}{=} (lc(v) \geq Timing(v))$$

   Then, the enabling test to terminate a node corresponds to:
   $$eFinish(v) \stackrel{def}{=} \begin{cases} haveTokens(v) \wedge pTiming(v) & \text{if } lab(v) = action \\ haveTokens(v) & \text{otherwise} \end{cases}$$
3. **eTime** determinates if the clocks can be increased. The clocks can be increased only during working time, i.e. when there is at least one action that is executing:
   $$eTime() \stackrel{def}{=} \bigvee_{v \in V, lab(v) = action} ((m(v) > 0) \wedge (lc(v) < Timing(v)))$$

Transition firing. The firing of a transition and the effect it has on a state can be defined as follows. Also here, two cases have to be distinguished: (i) firing a transition on a node that can start; (ii) firing a transition on a node that can terminate and (iii) firing a transition to represent the elapsing time.

Let us consider a second state $s' = (m', gc', lc')$. **fStart**, **fFinish** and **fTime** express the constraints that must be satisfied to ensure that s' is a successor of s. **fStart** is a constraint related to a staring node (a node that satisfies the enabling predicate **eStart**), **fFinish** is a constraint related to a finishing node (a node that satisfies the enabling predicate **eFinish**), and **fTime** is a constraint related to the increasing of the clocks (if the current state satisfies the enabling predicate **eTime**). For simplification, we assume that s and s' are implicitly available in the following firing predicates.
We first introduce the predicate fz that constrains to equality the marking of all the vertices and edges of s and s', except the one given as parameter p:

$$fz(p \in (V \cup E)) \stackrel{def}{=} \bigvee_{v \in \{(V \cup E) \setminus p\}} (m'(v) = m(v))$$

1. **fStart** is based on the following auxiliary predicates.
   - An *activity* node is executed by adding a token on it and removing the offers from its incoming edges:

$$sNode(v) \stackrel{def}{=} (m'(v) = m(v) + 1) \wedge \bigwedge_{e \in incoming(v)} (m'(e) = m(e) - m'(v)) \wedge fz(v \cup incoming(v))$$

   - An *action* node requires some additional conditions due to the presence of input and output pins. Offers from the incoming edge of its input pin are consumed up to the maximum bound allowed by the multiplicity. Then, tokens are produced on the output pin between the lower and upper multiplicity bound:

$$sActionIPin(v) \stackrel{def}{=} \bigwedge_{\substack{i \in input(v),\\ inc \in incoming(i)}} ((\neg(upper(i) \geq m(inc)) \wedge m'(i) = m(inc)) \vee ((upper(i) \geq m(inc)) \wedge m'(i) = upper(i)))$$

$$sActionOPin(v) \stackrel{def}{=} \bigwedge_{o \in output(v)} (m'(o) \geq lower(o) \wedge m'(o) < upper(o))$$

$$sActionEdge(v) \stackrel{def}{=} \bigwedge_{\substack{e \in incoming(v)\\ \cup incoming(input(v))}} (m'(e) = m(e) - m'(v))$$

$$sAction(v) \stackrel{def}{=} (m'(v) = m(v) + 1) \wedge sActionIPin(v) \wedge sActionOpin(v) \wedge sActionEdge(v) \wedge fz(v \cup input(v) \cup output(v) \cup incoming(v) \cup incoming(input(v)))$$

   - Unlike the other nodes, a *merge* node is executed by removing offers from only *one* of its incoming edges:

$$sMerge(v) \stackrel{def}{=} (m'(v) = m(v) + 1) \wedge \Big(\bigvee_{\substack{e \in incoming(v),\\ m(e) > 0}} (m'(e) = m(e) - m'(v)) \wedge fz(e \cup v) \Big)$$

Then, the transition firing, for a starting node, is characterized by:

$$fStart(v) \stackrel{def}{=} \begin{cases} sAction(v) & \text{if } lab(v) = action \\ sMerge(v) & \text{if } lab(v) = merge \\ sNode(v) & \text{otherwise} \end{cases}$$

2. **fFinish** is based of the following auxiliary predicates.
   - An *activity* node remove its owning tokens and offers it on all its outgoing edges:

$$fNode(v) \stackrel{def}{=} (m'(v) = m(v) - 1) \wedge \bigwedge_{e \in outgoing(v)} (m'(e) = m(e) + m(v)) \wedge fz(v \cup outgoing(v))$$

   - A $flowFinal$ node removes its token but do not offer it:

$$fFlowFinal(v) \stackrel{def}{=} (m'(v) = m(v) - 1) \wedge fz(v)$$

   - A *decision* node only offers its token only on one of its outgoing edges:

$$fDecision(v) \stackrel{def}{=} (m'(v) = m(v) - 1) \wedge \bigvee_{e \in outgoing(v)} (m'(e) = m(e) + m(v) \wedge fz(v \cup e))$$

   - An *action* node requires to reset its tokens on both its input and output pins, and offers its tokens on its outgoing edges and outgoing edges of its output pins. Moreover, its local clock is reinitialized to 0:

$$fActionPin(v) \stackrel{def}{=} \bigwedge_{p \in (output(v) \cup input(v))} (m'(p) = 0)$$

$$fActionEdge(v) \stackrel{def}{=} \bigwedge_{\substack{e \in (outgoing(v) \\ \cup outgoing(output(v)))}} (m'(e) = m(e) + m'(v))$$

$$fAction(v) \stackrel{def}{=} (m'(v) = m(v) - 1) \wedge fActionPin(v) \wedge fActionEdge(v) \wedge (lc'(v) = 0) \wedge fz(v \cup input(v) \cup output(v) \cup outgoing(v) \cup outgoing(output(v)))$$

Then, the transition firing, for a finishing node, is defined by:

$$fFinish(v) \stackrel{def}{=} \begin{cases} fFlowFinal(v) & \text{if } lab(v) = flowFinal \\ fDecision(v) & \text{if } lab(v) = decision \\ fAction(v) & \text{if } lab(v) = action \\ fNode(v) & \text{otherwise} \end{cases}$$

3. **fTime** increases the local clock of each action currently executing and increases the global clock as well:

$$fTime() \stackrel{def}{=} (gc' = gc + 1) \wedge \bigwedge_{\substack{v \in Vlab(action) \\ \wedge (m(v) > 0)}} (lc' = lc + 1) \wedge fz(\emptyset)$$

At this point we are able to define the complete transition (successor relation between states). Basically, when an activity final node is executed, or when there is no other node that can either start or terminate, the execution is over.

Definition 6 (Successor Relation). *Let $P = (V, E, Types, lower, upper, lab, Resource, Use, Timing)$ be a process. Let $s = (m, gc, lc)$ and $s' = (m', gc', lc')$ be two states of P. s' is a successor of s, iff the predicate $transition(s, s')$ holds:*

$$step(v) \stackrel{def}{=} (eStart(v) \wedge fStart(v)) \vee (eFinish(v) \wedge fFinish(v)) \vee (eTime() \wedge fTime())$$

$$transition(s, s') \stackrel{def}{=} \bigwedge_{v \in Vlab(activityFinal)} (m(v) = 0) \wedge \bigvee_{v \in V} (step(v))$$

Thus, to represent a process execution, we define the notion of trace:

Definition 7 (Trace). *Let* $P = (V, E, Types, lower, upper, lab, Resource, Use, Timing)$ *be a process. A* ***Trace*** *is an ordered set of states denoted* $\sigma = \langle s_0, s_1, ..., s_n \rangle \in States^*$ *s.t.:* $\forall i \in \mathbb{N}, transition(\sigma[i], \sigma[i+1])$ *holds, where* $\sigma[i]$ *denotes the i-th state of the trace and* s_0 *is the initial state. The set of all trace is noted* ***Traces****.*

4 Properties for Process Verification

To study the properties of the modelled process using our formalization we need a formal logic. Many logics exist and can express different kind of properties: Computation Tree Logic (CTL), Linear Temporal Logic (LTL), etc. In our case, almost all our properties can be handled using LTL.

LTL formulae are constructed from atomic propositions, logical operators $\vee$, $\wedge$, $\neg$, and temporal operators X (meaning "next"), G ("globally"), U ("until") and F ("eventually") [14]. In our formalism, atomic propositions are statical (related to the structure of the process) or dynamical, of the form $m(n)$ *op* v or gc *op* v where $n \in V \cup E$, $op \in \{=, \neq, <, \leq, >, \geq\}$ and $v \in \mathbb{N}$.

Given a process $P = (V, E, Types, lower, upper, lab, Resource, Use, Timing)$ and an LTL property ϕ, we say that $P \models \phi$, iff $\forall \sigma \in \mathbf{Traces}, \sigma \models \phi$. It is worth noting that LTL semantics is defined over infinite traces. To treat the case of finite traces, we just used the so-called stuttering principle to extend a trace to an infinite one.

In the following, we give an overview of interesting properties that can be verified on a process and give some examples. Due to space restriction, we choose only some relevant constraints from each aspect of the process dimension. The goal here is to show the ability of our formalism to deal with a wide variety of process constraints rather than presenting them exhaustively.

Control-flow. Control-flow analysis deals with questions like "does the process terminate?", "Is there any deadlock?", "Does `TaskA` ever happen?", etc. These properties are often referred as *soundness* properties [12] in the literature. Soundness tends to check some desirable properties such that a started process can always complete (*option to complete*).

- *Option to complete* can be checked by verifying that at least one `ActivityFinalNode` of the process is always executed:

$$\mathsf{F}\left(\bigvee_{v \in Vlab(activityFinal)} (m(v) > 0)\right) \tag{1}$$

Data-flow. The goal of data flow analysis [15] is to validate the process against different data problems such as *missing data*, i.e. when a data element needs to be accessed, but either it has never been created or it has been deleted without having been created again.

- *Missing data* can be checked by ensuring that when a node has offers on its control edges, it will finally have offers on its input pin:

$$\mathsf{G}\left(\bigwedge_{v \in Vlab(action)} (pNode(v) \implies \mathsf{F}(pAction(v)))\right) \tag{2}$$

Resources. Resources properties deal with resource problems like *missing resource*, i.e., when an activity requires a resource which may not be available.

- *Missing resource* can be checked by verifying that when an action is ready to start, there is no other action currently executing utilizing the same resource:

$$\bigwedge_{\substack{v \in Vlab(action),\\ r \in Resource,\\ r \in Use(v)}} \mathtt{G}\ (eStart(v) \implies (\bigwedge_{\substack{o \in Vlab(action),\\ o \neq v}} (r \in Use(o) \implies m(o) = 0))) \quad (3)$$

Time. The goal of timing properties is to answer questions like "Is it possible to finish the process *on time* whatever the path taken?".

- To check that the process can terminate before max time unit can be expressed by the following LTL property:

$$\mathtt{F}(\bigvee_{v \in Vlab(activityFinal)} (m(v) > 0) \wedge (gc > max)) \quad (4)$$

  A counter-example to this formula means that at least one execution can terminate before max time unit. This answers the original property.

Business. While the other categories specify properties that must hold for all processes, business properties represent specific properties tailored to a given process. They play an important role since a process could be syntactically correct and valid against the precedent properties but still violates some business constraints. Business properties deal with questions like "does the `ImportantAction` is executed whatever the choice made during the execution?" or "Is `ImportantArtefact` (i.e., the goal of the process) always available at the end of the process?".

- Let $P = (V, E, Types, lower, upper, lab, Resource, Use, Timing)$ be a process. Let $\mathtt{actionA}, \mathtt{actionB} \in V, lab(\mathtt{ationA}) = lab(\mathtt{actionB}) = action$ be two action nodes and let $max \in \mathbb{N}$ be a natural representing the maximum time between the execution of two actions. To verify that when `actionA` is executed, `actionB` is always executed afterwards before max time units the property is expressed as follows:

$$\begin{aligned} \forall i \in \mathbb{N} : \mathtt{G}((m(\mathtt{actionA}) > 0 \wedge (gc = i)) \implies \\ \mathtt{F}((m(\mathtt{actionB}) > 0) \wedge (gc \leq max + i))) \end{aligned} \quad (5)$$

  Note that this constraint use infinite domain of integer which makes it not a standard LTL formula. However, we can turn it back to classical LTL formula by a simple modification of the treated model. For a sake of clarity, we do not burden the model and stick to our expression.

5 Implementation

We implement our formalization using the Alloy language [13]. It is is a declarative modeling language based on FOL and relational calculus for expressing complex structural and behavioral constraints. It is associated to a tool, called `Alloy Analyzer`: a constraint solver that provides fully automatic simulation and checking based on model-finding through SAT-solving (Satisfiability-solving).

On top of the formal framework implemented using Alloy, we have developed a prototype currently provided as an Eclipse EMF plugin. The main intent of this prototype is to assist the modeler by automatically verifying fUML-based

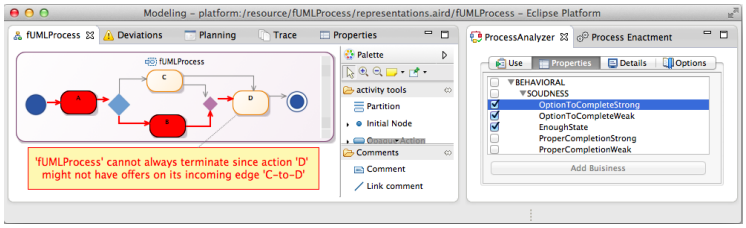

Fig. 3: Process Analyzer integrated inside our process environment, displaying a counter-example for the *option-to-complete* property.

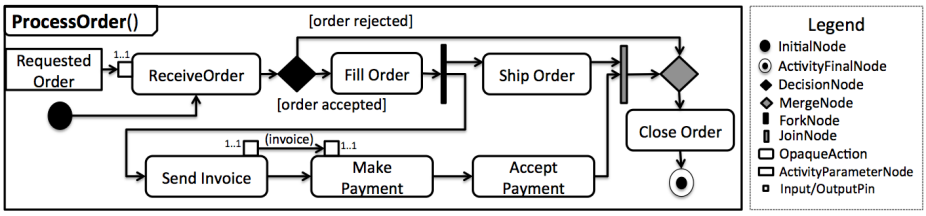

Fig. 4: `ProcessOrder` process represented as an fUML Activity Diagram

processes in the form of XMI Instances. It comes with a library of predefined properties ready to be checked, but also allows to add some common *business properties* through a graphical interface. The user only has to check in the interface the desired properties, and fill the parameter if required (e.g., maximum time to terminate the process). The *business properties* can be added through pre-defined templates, e.g. select the `ActionA` which must always be executed before `ActionB`. Figure 3 shows a screenshot of our tooling for process modeling and enactment emphasizing the process view and its analyzer. When the verification is performed, the path leading to the counter-example (if any) is highlighted in green for reachability properties, and in red otherwise.

5.1 Case Study

This section presents a case study on the Alloy implementation by checking the properties presented in section 4 on the `ProcessOrder` process from the UML specification [10]. This process simply proceeds the arrival of a new order and is visible on Figure 4.

To perform the verification, the process and the properties are translated into an Alloy specification. Then, this specification is given as input to the `Alloy Analyzer` which reduces the verification to a SAT problem. It is presented to a SAT solver (MiniSat among others) in a Conjunctive Normal Form (CNF) format. A CNF is a conjunction of clauses. Each clause represents a disjunction of variables. A satisfying assignement to a SAT problem consists of a boolean affectation to the variables such that all clauses are satisfied. Usually, the complexity of a SAT problem is measured by the number of clauses and variables.

Let $P = (V, E, Types, lower, upper, lab, Resource, Use, Timing)$ be the process from Figure 4 where $(V, E, Types, lab)$ are displayed on the figure, $Resource = \{\texttt{BankConnector}\}$, $Use = \{\texttt{AcceptPayment} \mapsto \{\texttt{BankConnector}\}\}$ and $Timing = \{\texttt{ReceiveOrder} \mapsto 1, \texttt{FillOrder} \mapsto 2, \texttt{SendInvoice} \mapsto 1, \texttt{MakePayment} \mapsto 1, \texttt{AcceptPayment} \mapsto 2, \texttt{ShipOrder} \mapsto 3, \texttt{CloseOrder} \mapsto 1$

Property	T. Semantics	Vars	Clauses	CNF Gen.	SAT Solving	C.E.
(1) Control-flow		410k	1239k	31s	5s	no
(2) Data-flow		427k	1286k	27s	5s	no
(3) Resources		419k	1261k	33s	0.3s	no
(4) Time	✓	2361k	8716k	185s	3s	yes
(5) Business	✓	2385k	8716k	201s	199s	no

Table 1: Metrics from the `Alloy Analyzer` executed on the `ProcessOrder` from Figure 4

}. Generally, these pieces of information are available with the model through UML Profiling [10] or as direct extension of the UML AD metamodel [3]. For sake of clarity, we do not propose some graphical representation of these data (*Resource*, *Use* and *Timing*) but directly give their formal representation.

Table 1 summarizes the obtained results. Column 1 represents the analyzed property from Section 4. For the "(4) time" property, we are using $max = 4$. Concerning the "(5) business" property, we choose the two actions `FillOrder` and `SendInvoice`, and $max = 6$. Columns 2 specifies if we are using the timed semantics for the verification. Due to the presence of the global and local clock ticks, a lot of extra states are introduced to support the properties related to the time. For efficiency reason, we also implemented a version of the semantics without these clocks on each state for the properties which are not relying on it. Columns 3 and 4 represent, respectively, the number of generated variables and clauses. Columns 5 and 6 represent, respectively, the time to generate the CNF and to solve the SAT problem. Columns 7 specifies the result of the verification, if there is any counter-example. All analyses were performed on a MacBook Air 2011 with Intel Core i5 processor and 4GB of RAM with Mavericks as OS.

These results highlight the effectiveness of our tool w.r.t. a concrete example. Actually, even if the whole generated SAT problems present a relatively high complexity (over 1 million clauses and over 410 thousand variables), the solving time is less than 1 minute for the untimed properties. The timed-related properties have a similar ratio in terms of clauses and variables but require few minutes due to the presence of extra states. Interested readers can download the complete Alloy formalization with the case study from our website[3].

6 Related Work

There is an extensive literature on verifying process models. A complete overview of the related work would be beyond the scope of this paper (see [12]). Therefore, we focus on the work directly relevant to this paper, namely formal verification approaches of UML AD.

Generally, the verification is based on mapping the process model into mathematical formalisms used to model systems such as automata, Petri Nets or process algebra. All of these formalisms have been investigated for the verification of UML AD. Jung et al. [16] propose a transformation from UML AD to Colored Petri Nets. Dong et al. [7] presents an approach for formalizing UML AD using π-calculus, a kind of computing models for representing concurrent systems and express the interactions between evolving processes. Eshuis et al.

[3] http://pagesperso-systeme.lip6.fr/Yoann.Laurent/

[8] check UML AD in the context of workflow modeling by translating the activity into the input language of NuSMV, a symbolic model checker. Guelfi et al. [9] propose a translation of UML AD extended with timing constraints into Promela (Process or Protocol Meta Language) in order to check behavioral properties with the model-checker SPIN. However, these works are not based on the new fUML standard and have done some assumptions on the precise operational semantics which creates tool-interoperability problems. Moreover, the semantics richness of these approaches are less complete than fUML, many simplifications have been carried out. While all of these approaches propose to check control-flow related properties, data-flow are not always considered and only [9] supports the timing constraints. Properties related to the resources are never supported.

Montogna et al. [17] propose an approach allowing the definition of a virtual machine for fUML in the $\mathbb{K}$-Framework, enabling the execution of models on a more formal definition than the current Java-based implementation. To the best of our knowledge, there is no temporal logic verification proposed.

Abdelhalim et al. [18] present an approach to manually map an fUML models into the process algebraic specification language CSP (Communicating Sequential Processes) and use the FDR (Failures-Divergences Refinement) model-checker to check if the model is deadlock free. When a deadlock is found, a counter-example trace which led to the deadlock is generated. Their formalization focuses only on the asynchronous communication between objects within fUML which has been guided by their case study.

To the best of our knowledge, our work is the first attempt to formalize the tokens game of the fUML standard to verify process models. If a comparison is made between the above-mentioned work, our approach is not relying on the semantics and concepts of the targeted formal language in terms of expressiveness, e.g. Petri Nets, instead of the modeling language. In these approaches, the assumption is made that the semantics choices made in these formal techniques are valid as well for UML AD.

7 Conclusion and Future Work

This paper proposes a first-order logic formalization of the newly defined fUML specification to verify fUML-based process models. The formalization is able to deal with the control- and data-flow, resources, and timing aspects of the process in a unified way. A tool implementation based on the Alloy modeling language has been successfully integrated in an Eclipse-based process environment. The tool is able to verify automatically a wide range of properties without the user's intervention and allows one to verify some *business* properties. Currently, the tool is under evaluation within the European MERgE project, which main goal is to develop and demonstrate innovative concepts and design tools addressing both "safety" and "security" concerns in development processes.

The case study and the tool proved the feasibility of our approach, however some improvements are already under realization. The first one consists in covering the formalization of more UML AD concepts that can be of interest for the modeling of more complex processes. Examples of such concepts are `DataStoreNode` (a buffer for non-transient data), `AcceptEventAction` and `SendSignalAction` (for dealing with events) and `StructuredActivity` (expansion, loop, conditional nodes). Moreover, we are working on extending the formalization to be *data-aware*. Currently, the contents of the tokens within the

`ObjectNodes` are not taken into account. This prevents, for example, to express *guard* on edge to determine if the edge can be traversed. Some formalizations have taken some of these concepts into accounts [8]. However, much simplification has been done in comparison of the way fUML operates and only integers are considered. In fUML, each tokens can have a simple value type (integer, string, natural, boolean) or more complex `Classifier` type defined in a Class Diagram. Then, tokens are manipulated using the action nodes from the `IntermediateActions` package. This package defines the classical actions to create, read, suppress and modify tokens at runtime within the AD and formalizing such concepts is a non-trivial task. Finally, we are exploring optimizations techniques to treat the properties related to time in a more efficient way based on the expertise of well-known approaches such as timed automata.

Acknowledgments. This work was funded by the MERgE project (ITEA 2 Call 6 11011).

References

1. Mendling, J.: Empirical studies in process model verification. In: Transactions on Petri Nets and Other Models of Concurrency II. Springer (2009) 208–224
2. Mendling, J., Verbeek, H., van Dongen, B.F., van der Aalst, W.M., Neumann, G.: Detection and prediction of errors in epcs of the sap reference model. Data & Knowledge Engineering **64**(1) (2008) 312–329
3. Bendraou, R.: Uml4spm: A uml2. 0-based metamodel for software process modelling. MoDELS (2005) 17–38
4. Bendraou, R., Jézéquel, J., Gervais, M., Blanc, X.: A comparison of six uml-based languages for software process modeling. Software Engineering, IEEE Transactions on **36**(5) (2010) 662–675
5. Russell, N., van der Aalst, W.M., Ter Hofstede, A.H., Wohed, P.: On the suitability of uml 2.0 activity diagrams for business process modelling. In: Proceedings of the 3rd Asia-Pacific conference on Conceptual modelling-Volume 53
6. van Der Aalst, W.M., Ter Hofstede, A.H., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and parallel databases **14**(1) (2003) 5–51
7. Dong, Y., ShenSheng, Z.: Using π-calculus to formalize uml activity diagram for business process modeling. In: ECBS, IEEE (2003) 47–54
8. Eshuis, R.: Symbolic model checking of uml activity diagrams. TOSEM (2006)
9. Guelfi, N., Mammar, A.: A formal semantics of timed activity diagrams and its promela translation. In: APSEC, IEEE (2005)
10. OMG: Uml version 2.4.1. http://www.omg.org/spec/UML/ (2011)
11. OMG: Fuml version 1.1. http://www.omg.org/spec/FUML/ (2013)
12. van der Aalst, W., Van Hee, K., ter Hofstede, A., Sidorova, N., Verbeek, H., Voorhoeve, M., Wynn, M.: Soundness of workflow nets: classification, decidability, and analysis. Formal Aspects of Computing **23**(3) (2011) 333–363
13. Jackson, D.: Software Abstractions: logic, language and analysis. MIT Press (2011)
14. Pnueli, A.: The temporal logic of programs. In: Foundations of Computer Science, 1977., 18th Annual Symposium on, IEEE (1977) 46–57
15. Trčka, N., van der Aalst, W., Sidorova, N.: Data-flow anti-patterns: Discovering data-flow errors in workflows. In: AISE, Springer (2009) 425–439
16. Jung, H.T., Joo, S.H.: Transformation of an activity model into a colored petri net model. In: TISC, IEEE (2010) 32–37
17. Motogna, S., Cr Ciun, F., Lazar, I., Pârv: Formal definition of fuml in k-framework. Studia Universitatis Babes-Bolyai, Informatica **58**(3) (2013)
18. Abdelhalim, I., Sharp, J., Schneider, S., Treharne, H.: Formal verification of tokeneer behaviours modelled in fuml using csp. In: FMSE. Springer (2010) 371–387