RZ 3673 (# 99683) 10/17/06
Computer Science 27 pages

Research Report

A Classification of UML2 Activity Diagrams

Jana Koehler, Jochen M. Küster, John Novatnack and Ksenia Ryndina

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

{koe, jku, ryn}@zurich.ibm.com

Research
Almaden · Austin · Beijing · Delhi · Haifa · T.J. Watson · Tokyo · Zurich

A Classification of UML2 Activity Diagrams

Jana Koehler **Jochen M. Küster** **John Novatnack** **Ksenia Ryndina**
IBM Zurich Research Laboratory
CH-8803 Rueschlikon
Switzerland
email: {koe,jku,ryn}@zurich.ibm.com

Technical Report RZ 3673

Abstract

We present the results of a case study where we investigated a semantic mapping of UML2 activity diagrams to the π-calculus. Our study was initiated by recent discussions on the role of the π-calculus for future business-process management systems as well as our interest in developing formal analysis techniques for activity diagrams.

The study revealed interesting insights into the semantic expressivity of activity diagrams and the semantic nature of the different modeling elements, in particular of object nodes and activity final nodes. We show that for certain types of diagrams, a semantic mapping of object nodes, in particular of pins, to message reading and receiving operations is insufficient and propose an encoding of pins as π-processes. Our results motivated us to present a novel classification of activity diagrams based on their semantic expressivity.

1 Introduction

The release of Version 2 of the Unified Modeling Language (UML2) [17] has seen a considerable extension to model functional behavior. In particular, activity diagrams have significantly changed and many new interesting features have been added. The semantics of UML2 activity diagrams is informally described and many examples are given throughout the specification to illustrate their behavior. Furthermore, the series of articles by C. Bock in the Journal of Object Technology [3, 4, 5, 6] contributed to a better understanding of UML2 activity diagrams and their difference to activity diagrams as defined in UML 1.5 [24].

With the increasing interest in service and business process modeling, the role of UML2 and in particular of activity diagrams also seems to grow. Many approaches to business process modeling use modeling constructs which are very close to those introduced for UML2 activity diagrams or rely directly on variants of them. Examples are a proposal for a Business Process Definition Metamodel as described in [1], the Business Process Modeling Notation [23], or the language used in the IBM WebSphere Business Modeler [34]. Using UML2 activity diagrams for modeling functional behavior is also appealing because of their integration with UML2 state machines and class diagrams. Besides this, UML2 has been found to support many of the well-known workflow patterns [41, 25].

In this paper, we take a closer look at some of the semantic subtleties of activity diagrams that provide significant challenges when used in a practical environment. The challenges come from two sides. First, the user has to fully understand the semantics in order to correctly capture the intended behavior. In our own work, we found that users very often produce erroneous models as soon as the behavior requires to capture a combination of cyclic with sequential or parallel data/control flows, or if sequential and parallel flows must be mixed. In these examples, we often see deadlocks, livelocks

and incorrect multiple instantiations of activities. Secondly, the semantics has to be precisely built into simulation and analysis tools to enable automatic analysis and verification. We are therefore interested in formalizing the semantic expressivity of various classes of activity diagrams and investigate how analytical methods can be tailored to these classes.

So far, formalizations of activity diagrams based on abstract state machines, Petri nets, finite-state processes, stream-processing functions, the software specification formalism B, and finally the π-calculus have been proposed.[1] For our case study, we decided to use the π-calculus by Milner, Parrow, and Walker [14, 16] as the formalism of choice. Our motivation came from a controversial discussion on the suitability of the π-calculus as the formal foundation for future business-process software over the last few years. Our choice does not imply that other formalisms (notably Petri nets) would not be suitable for a formalization. The goal of our formalization is to gain a better understanding of the semantic subtleties of the diagrams. Relevant results have been published for statecharts [40] based on a comparison of semantic formalizations.

Our insights into the formalization of the various modeling elements allows us to come up with a classification of the diagrams based on their semantic expressivity. We believe that the identification of semantic challenges, their formal discussion and the resulting classification help in further strengthening the role of activity diagrams in any modeling scenario. A semantic classification also provides an interesting foundation for further theoretical investigations of activity diagrams. From a practical point of view, a restricted class of activity diagrams, which can be adequately captured in a less fine-grained formalization is of great interest for reasons of early quality assessment and assurance, based on models at a high abstraction level. Our insights also provide the basis for optimization techniques that can be applied to a semantic encoding in order to reduce the size of the resulting transition system. Such optimizations are very important to scale analytical algorithms to the size of models occurring in practice and to reduce their runtime to practically acceptable answering times.

The paper is organized as follows: We briefly recapture the essentials of UML2 activity diagrams in the next section, set the scope of our formalization and present an example, which we will use throughout this paper. In Section 3, we give a short introduction into the π-calculus. Section 4 takes a sang-froid approach and presents an initial formalization by mapping each activity to a π-calculus process. In Section 5, we encode the example according to this formalization. In Section 6, we review this initial formalization and identify its shortcomings, which we locate in particular in the semantic mapping of object nodes. We identify semantic challenges that the initial formalization does not correctly capture and show how to extend the formalization. Section 7 presents a more adequate semantic formalization of object nodes. In Section 8, we summarize our findings and derive the desired classification of activity diagrams, but also formulate a set of open questions for future research. After a discussion of related work in Section 9, we conclude with an outlook on future work in Section 10.

2 UML Activity Diagrams with Parameter Sets

This section briefly recaptures the main features of UML2 activity diagrams as described in [17] with a specific focus on parameter sets, which were newly introduced in UML2. UML2 activity diagrams follow traditional control and data flow modeling approaches and use the token-flow semantics from Petri nets to informally describe the semantics. An *activity* describes the functional behavior under consideration. Depending on the degree of abstraction, an activity can be refined by an activity model showing (sub)-activities at a more fine grained level, possibly combined with three types of (not further refinable) nodes: action nodes, control nodes, and object nodes. *Action* nodes operate on control and data values that they receive, and provide control and data to other actions [3]. *Control* nodes route control and data values

[1] See the Related Work section for a detailed discussion.

through the graph. Among the control nodes, UML2 defines *decision* and *merge* for sequential branching and joining, and *fork* and *join* for parallel branching and joining [5]. Furthermore, the start node and two kind of end nodes belong to the control nodes.

Figure 1 shows an example of an activity diagram that we will use throughout this paper. Action nodes are depicted with round-cornered rectangles. In our example, we have actions named A, B, C, D, E, F. As for control nodes, we have the *initial* node S, which is depicted with a black dot, the *flow final* node X, which is depicted with a circle containing a cross, and the *activity final* node T, which is depicted with a circle containing a black dot. Other control nodes, such as fork, join, decision, or merge do not occur in our example, but will be formalized.

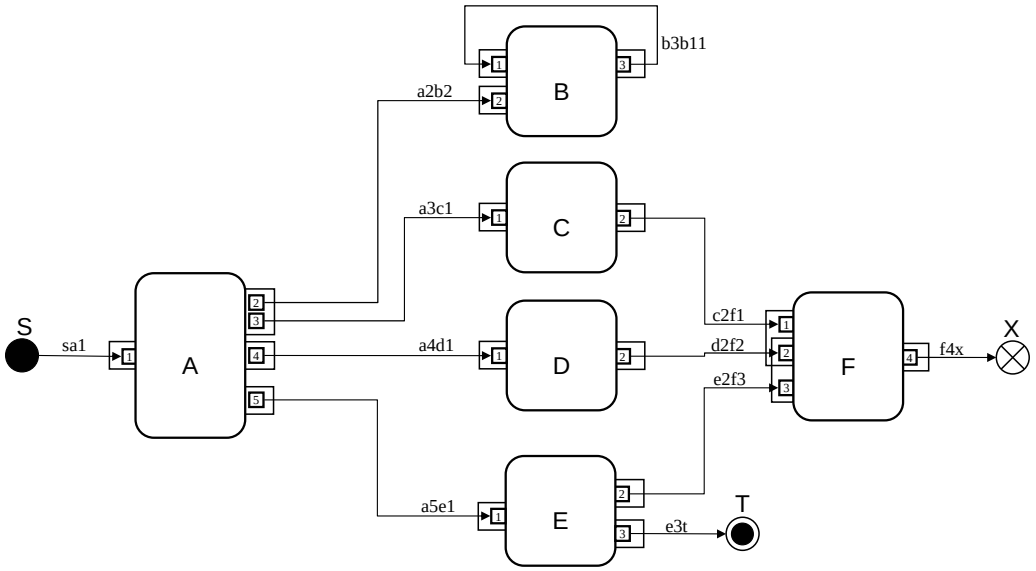

Figure 1: Example activity diagram with named pins, edges, action and control nodes.

The third type of nodes are *object* nodes, which hold data tokens temporarily as they wait to move through the graph [6]. Our example contains *pins*, which hold the input and output of actions, and which we simply enumerated in Figure 1. UML2 further defines *activity parameter nodes*, which we do not consider separately in our formalization due to their apparently strong semantic alignment with pins. Unfortunately, the UML2 metamodel does not completely define the precise relationship between the pins of an action node and the parameter nodes of the associated activity. Parameters can be grouped into so-called *parameter sets*, such that exactly one of the groups can accept or provide values for the action [4]. No association is defined between parameters and pins and there is also no notion of a pin set, however, Figure 12.117 in [17, page 387] shows a grouping of pins using parameter set notation. Thus, in our formalization, we assume that parameter sets lead to a corresponding grouping of pins and we will speak of the pins belonging to a specific parameter set — the same view is also adopted in [32]. In Figure 1, Parameter sets are graphically depicted by rectangles that group pins. We do not name them explicitly in the diagram. Parameter sets must have at least one pin in them, but the same pin can be part of more than one parameter set, i.e., we will speak of so-called overlapping parameter sets. Let us consider action F in the example. It contains two overlapping input parameter sets, which share the input pin 2. F only has a single output parameter set containing the single output pin 4. Furthermore, UML2 defines *centralbuffers* and *datastores*, which we exclude from our formalization.

Control and data flow is modeled by edges, usually beginning in output pins and ending in input pins. UML2 takes a unified approach to control and data. A pin is a control pin having a control type if its *isControl* attribute is set to *true*, otherwise it is a data pin having a data type. In our example, we

named edges by combining the action and pin names that participate in the connection, e.g., $a4d1$ is the edge that connects the ouput pin 4 of action A with the input pin 1 of action D. In UML2, only action nodes have pins, while control nodes do not have pins, but route the control and data flow directly. In our formalization of control nodes, we deviate from this difference and also define pins for control nodes (except for initial, flow final and activity final nodes). This allows us to simplify the syntax definition of activity diagrams and to treat all activity nodes uniformly in the semantics. For example, we will define edges as connections betweens pins, while otherwise, we had to define edges between pins for action nodes, but for the nodes without pins the edge would begin or end directly in the node. We summarize our short review of UML2 activity diagrams in the following syntax definition, in which we focus on the main modeling elements as described above. Compared to the original metamodel definition, this definition is a strong simplification. Note that we also do not consider hierarchical activity diagrams.

Definition 1 (Syntax) *An activity diagram $M = (N, P, \mathcal{I}, \mathcal{O}, E)$ consists of*

- *a finite set N of nodes partitioned into subsets i.e., $N = N_A \cup N_C$ where N_A denotes the set of action nodes, while the set of control nodes N_C is further partitioned into subsets:*
  - *N_D of decision nodes, N_M of merge nodes, N_F of fork nodes and N_J of join nodes,*
  - *N_I of initial nodes, N_{FF} of flow final nodes, and N_{AF} of activity final nodes.*
- *a finite set P of pins partitioned into input and output pins, i.e., $P = P_{in} \cup P_{out}$,*
- *a finite set of input parameter sets $\mathcal{I}$ where each set $I \in \mathcal{I}$ is a subset of the set of input pins, i.e., $I \subseteq P_{in}$,*
- *a finite set of output parameter sets $\mathcal{O}$ where each set $O \in \mathcal{O}$ is a subset of the set of output pins, i.e., $O \subseteq P_{out}$,*
- *a finite set E of edges each connecting either two pins with each other or a pin with a control node, i.e., $E \subseteq (P_{out} \cup N_I) \times (P_{in} \cup N_{FF} \cup N_{AF})$.*

UML2 formulates several syntactic constraints such as:

- each action node has at least one input and one output parameter set, and
- there are no empty parameter sets, i.e., each set contains at least one pin and a set is associated with exactly one action node,
- no two parameter sets have exactly the same pins.

Furthermore, the specification states that nodes with unconnected pins cannot execute. Thus, from a practical point of view, one could require that all pins and control nodes are connected by edges—however, this is not relevant for our formalization. UML2 allows that pins may be arbitrarily connected to even more than one other pin, i.e., they cannot only be used to model the routing behavior of control nodes, but additional nondeterminism occurs when, e.g., one output pin is connected to more than one input pin. However, we believe that it would be a better modeling practice to have only 1-1 edges between pins and control nodes and thus base our formalization on this assumption. A discussion of the semantic problems caused by 1-m edges between pins can be found in [33] and we will also come back to this issue later in the paper.

3 A short π-calculus Glossary

The π-calculus is a process algebra developed by Milner, Parrow and Walker [14, 16], which reuses concepts from the Calculus of Communicating Systems (CCS) [14]. It introduces the action prefixes of receiving any message y along a channel x (denoted as $x(y)$), sending a message z along a channel x (denoted as $\overline{x}\langle z\rangle$), or making a silent transition (denoted as τ). If the message is not relevant, the shorthand notations $\overline{x}$ for sending over channel x and x for receiving over channel x can be used.[2] Action prefixes of a process are composed by the . operator. A simple example illustrating these concepts is the process $A = \overline{x}\langle y\rangle.A'$, which sends a message y over the channel x and then continues to behave like process A'.
Given a set of action prefixes π_i, the basic syntax of process expressions is defined by

$$\begin{array}{lll} P \quad ::= \quad \sum_{i=1}^{n} \pi_i.P_i \mid & & \textit{alternative composition} \\ \prod_{i=1}^{n} P_i \mid & & \textit{parallel composition} \\ \text{new a } P \mid & & \textit{restriction of name scope} \\ !P & & \textit{infinite replication} \end{array}$$

A process $\sum_{i=1}^{n} \pi_i.P_i = \pi_1.P_1 + \cdots + \pi_n.P_n$ is a process that performs one of the actions prefixes π_i and then behaves like process P_i. The empty sum results in the null process, 0, which is often omitted. For example, we write $\overline{x}\langle y\rangle$ instead of $\overline{x}\langle y\rangle.0$. A process $\prod_{i=1}^{n} P_i = P_1 \mid \cdots \mid P_n$ is the concurrent composition of processes P_1 to P_n. The restriction *new a* P restricts the scope of the name a to process P. The replication operator ! allows a process P to be replicated an infinite number of times.

Note that $x(z).P$ and *new z* P has the effect of binding the name z to scope P. The binding of z in $x(z).P$ will lead to a substitution of the free occurrences of z in P by a when receiving a via x. For example, $(x(z).\overline{z}\langle a\rangle.0 \mid \overline{x}\langle w\rangle.0)$ evolves to $(\overline{w}\langle a\rangle.0|0)$ by an interaction over the channel x. The π-calculus offers the ability to transform processes according to a set of structural congruences. For example, $P \mid 0 \equiv P$ states that every process P composed in parallel with the null process is structurally congruent to itself. $P + 0 \equiv P$ shows that adding a null process to a process P does not increase its capabilities.

We introduce a notational shorthand to capture the possible permutations of action prefixes, i.e., instead of writing $a.b.A + b.a.A$, we write $(a \mid b).A$ and define:

Definition 2 (Notational convention for prefix permutations) *Given action prefixes $a_1, \ldots, a_n$ and a process A, we write*

$$\begin{aligned} \Big(\prod_{i=1}^{n} a_i \Big).A &= (a_1 \mid \cdots \mid a_n).A \\ &= a_1.a_2.\ldots a_n.A + a_2.a_1.\ldots a_n.A + \cdots + a_1.\ldots.a_n.a_{n-1}.A \end{aligned}$$

A second notational shorthand is used to capture the alternative composition of prefixes with the same process in a more compact way:

Definition 3 (Notational convention for alternative prefix composition) *Given action prefixes $a_1, \ldots, a_n$ and a process A, we write*

$$\begin{aligned} \Big(\sum_{i=1}^{n} a_i \Big).A &= (a_1 + \cdots + a_n).A \\ &= a_1.A + \cdots + a_n.A \end{aligned}$$

[2] Note that one often speaks of names being sent or received via some other name, however, we will continue to speak of channels and messages, because we hope that this makes the formalization more intuitive.

The semantics of the π-calculus is defined using *reaction rules*, cf. [15, page 91]. These reaction rules (also called reduction rules in [26]) describe how a process P can evolve to a process P' owing to some action within P. In addition, the π-calculus provides *transition rules*, which define how a system may interact with its environment. As we will focus on processes without any environmental interaction, we restrict ourselves to reduction rules as shown in Table 1.

TAU	$\overline{\tau.P + M \rightarrow P}$
INTER	$\overline{(x(y).P + M) \mid (\overline{x}\langle z\rangle.Q + N) \rightarrow \{z/y\}P \mid Q}$
PAR	$\dfrac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$
RES	$\dfrac{P \rightarrow P'}{\text{new z } P \rightarrow \text{ new z } P'}$
STRUCT	$\dfrac{P \rightarrow P'}{Q \rightarrow Q'}$ if $P \equiv Q$ and $P' \equiv Q'$

Table 1: The π-calculus reduction rules.

The TAU-rule defines that a process $\tau.P + M$ evolves to the process P when performing an internal τ-transition. The rule INTER describes the interaction of two concurrently running processes, which synchronize on the exchange of a message over a channel x. The process $x(y).P$ expects any message y, and the process $\overline{x}\langle z\rangle.Q$ sends a message z over x. The rule describes that the result of this interaction will be $P|Q$, where in P all free occurrences of y have been replaced by z. The alternative behaviors M and N are not preserved, i.e., if a "sum exercises one of its capabilities, the others are rendered void", cf. [26, page 39]. The rule PAR states that if a process P evolves to P' then process $P \mid Q$ evolves to $P' \mid Q$. The rule RES shows that "restriction of a name does not inhibit a reduction" [26]. The last rule, STRUCT, establishes a link between reduction and structural congruence: If P reduces to P' and P is structurally congruent to Q and P' is structurally congruent to Q', then also Q reduces to Q'. This link is important because it allows one to exploit structural congruence and then apply the reduction rules.

4 An Initial Formalization of Activity Diagrams

Parameter sets in UML2 introduce an enormous expressivity for modeling communication behavior between actions. The UML2 specification gives the following informal characterization:

> *"A behavior with input parameter sets can only accept inputs from parameters in one of the sets per execution. A behavior with output parameter sets can only post outputs to the parameters in one of the sets per execution. Multiple object flows entering or leaving a behavior invocation are typically treated as 'and' conditions. However, sometimes one group of flows are permitted to the exclusion of another. ... The notation ... expresses a disjunctive normal form where one group of 'and' flows are separated by 'or' groupings. For input, when one group or another has a complete set of input flows, the activity may begin. For output, based on the internal processing of the behavior, one group or other of output flows may occur."* [17, page 386]

Similar to Dong and ShenSheng [9] and Puhlman and Weske [20], our semantic mapping maps each activity and action node to a separate π-calculus *process*. Edges between pins and control nodes are mapped to *channels*, allowing the token flow in activity diagrams be mapped to the message-based

communication of π-calculus processes. The semantics of parameter sets is captured by defining the composition and communication of the π-processes accordingly. The following definitions introduce a function $\pi : M \rightarrow \mathcal{P}$, which maps the diagram M to a process definition $\mathcal{P}$ in the π-calculus. Let $M = (N, P, \mathcal{I}, \mathcal{O}, E)$ be an activity diagram. We define the function π as follows:

Definition 4 (Mapping of an Edge) *An edge $e \in E$ is mapped to a channel name e in the π-calculus, i.e., we define $\pi(e) = e$.*

The semantics of input and output pins is captured by a message sending or receiving action prefix:

Definition 5 (Mapping of an Input/Output Pin) *Let p be an output pin, q be an input pin, and $e = (p, q)$ be the edge that connects both pins. Let t be some, not further specified message in the π-calculus. We define $\pi(p) = \overline{\pi(e)}\langle t \rangle$ and $\pi(q) = \pi(e)(t)$.*

Definition 6 (Mapping of a Parameter set) *The semantics of a parameter set $S = \{p_1, \ldots, p_k\}$ containing pins p_1 to p_k is the parallel composition of the mapping of the pins:*

$$\pi(S) = \prod_{i=1}^{k} \pi(p_i) = \pi(p_1) \mid \cdots \mid \pi(p_k) \ .$$

Note that this definition makes use of our notational shorthand as defined in Definition 2. The alternative behaviors of an action, which are encoded in several input and output parameter sets, can now be captured as follows:

Definition 7 (Mapping of an Action Node) *Let A be an action node with input parameter sets $I_1, \ldots, I_k$ and output parameter sets $O_1, \ldots, O_n$:*

$$\pi(A) = A \stackrel{def}{=} \Big(\sum_{i=1}^{k} \pi(I_i) \Big).\mathit{performA}.\Big(\sum_{j=1}^{n} \pi(O_j) \Big).0 \ .$$

This means, a process A executes message-reading operations corresponding to one of its input parameter sets, then executes a *performA*-transition denoting the execution of the process followed by the message-sending operations corresponding to one of the output parameter sets. Note that this definition makes use of our second notational shorthand as defined in Definition 3.

Definition 8 (Mapping of Control Nodes) *Let D be a decision node with input parameter set I and output parameter sets $O_1, \ldots O_n$ and let M be a merge node with input parameter sets $I_1, \ldots, I_n$ and output parameter set O:*

$$\pi(D) = D \stackrel{def}{=} \pi(I).\Big(\sum_{j=1}^{n} \pi(O_j) \Big).0$$

$$\pi(M) = M \stackrel{def}{=} \Big(\sum_{i=1}^{k} \pi(I_i) \Big).\pi(O).0$$

Let F be a fork node with input parameter set I and output parameter set O and let J be a join node with input parameter set I and output parameter set O:

$$\pi(F) = F \stackrel{def}{=} \pi(I).\pi(O).0$$
$$\pi(J) = J \stackrel{def}{=} \pi(I).\pi(O).0$$

A decision only activates one of its output parameter sets whereas a merge only requires input from one of its input parameter sets. A fork produces output on all pins in its single output parameter set and a join requires input from all pins in its single input parameter set. Note that all outgoing branches in a fork occur in a single output parameter set because they need to be activated concurrently. Similarly, in a join all incoming branches occur in a single input parameter set. As already mentioned, we deviate here from the UML2 specification, which makes a very subtle distinction between the semantic behavior of pins of an action node and the behavior of a control node. In our semantics, we do not make this distinction and treat action and control nodes uniformly.

Initial and flow final nodes are mapped in a similar way:

Definition 9 (Mapping of Initial and Flow Final nodes)

$$\begin{array}{lcll} \pi(n \in N_I) & = & \overline{e}\langle t \rangle.0 & \textit{if e is the edge starting in start node n} \\ \pi(n \in N_{FF}) & = & e(t).0 & \textit{if e is the edge ending in the flow final node n} \ . \end{array}$$

We immediately encounter a problem when trying to adequately map the activity final node, which has a very complex semantics:

> *"A token reaching an activity final node terminates the activity. In particular, it stops all executing actions in the activity, and destroys all tokens in object nodes, except in the output activity parameter nodes. Terminating the execution of synchronous invocation actions also terminates whatever behaviors they are waiting on for return. Any behaviors invoked asynchronously by the activity are not affected. All tokens offered on the incoming edges are accepted. Any object nodes declared as outputs are passed out of the containing activity, using the null token for object nodes that have nothing in them. If there is more than one final node in an activity, the first one reached terminates the activity, including the flow going towards the other activity final."* [17, page 320]

We notice that it is impossible to adequately capture the semantics of the activity final node in the initial semantics. We address this problem in more detail in Section 6. As a temporary "solution", we treat activity final nodes in the same way as flow final nodes.

So far, we have encoded the various modeling elements that can occur in an activity diagram M. However, this is not sufficient. The semantics of the diagram can only be captured by defining how the individual π-processes are composed:

Definition 10 (Mapping of an activity diagram) *Let $M = (N, P, \mathcal{I}, \mathcal{O}, E)$ be an activity diagram with $E = \{e_1, \ldots, e_n\}$. Let $\pi(n_i)$ be the π-calculus mapping of node $n_i \in N = \{n_1, \ldots, n_k\}$. The mapping of M is defined as*

$$\pi(M) = M \stackrel{def}{=} \textit{new}\, e_1, \ldots, e_n \ \prod_{i=1}^{k} \pi(n_i) \ .$$

The definition restricts the scope of channel names to the process that encodes the diagram M.

5 Encoding of the Example

In this section, we discuss the behavior of the example activity diagram based on the initial formalization and discuss where the semantic mapping is not adequately capturing the behavior formulated in the UML2 specification. We begin by deriving the semantic mapping for our example from Figure 1. As we

mentioned before, we do not have a semantic mapping for the activity final node in the initial formalization and therefore temporarily encode it as a flow final node. The encoding makes use of our shorthand notations, which are used in the definition of processes A, B, E, F.

$$
\begin{array}{lcl}
S & \stackrel{def}{=} & \overline{sa1}\langle t\rangle.0 \\
A & \stackrel{def}{=} & sa1.(t).\mathit{performA}.\Big(\big(\overline{a2b2}\langle t\rangle \mid \overline{a3c1}\langle t\rangle\big) + \overline{a4d1}\langle t\rangle + \overline{a5e1}\langle t\rangle\Big).0 \\
B & \stackrel{def}{=} & \big(a2b2(t) + b3b1(t)\big).\mathit{performB}.\overline{b3b1}\langle t\rangle.0 \\
C & \stackrel{def}{=} & a3c1(t).\mathit{performC}.\overline{c2f1}\langle t\rangle.0 \\
D & \stackrel{def}{=} & a4d1(t).\mathit{performD}.\overline{d2f2}\langle t\rangle.0 \\
E & \stackrel{def}{=} & a5e1(t).\mathit{performE}.\big(\overline{e2f3}\langle t\rangle + \overline{e3t}\langle t\rangle\big).0 \\
F & \stackrel{def}{=} & \big(c2f1(t) \mid d2f2(t)\big) + \big(d2f2(t) \mid e2f3(t)\big).\mathit{performF}.\overline{f4x}\langle t\rangle.0 \\
T & \stackrel{def}{=} & e3t(t).0 \\
X & \stackrel{def}{=} & f4x(t).0 \\
M & \stackrel{def}{=} & \text{new } sa1, \ldots, f4x\big(S \mid A \mid B \mid C \mid D \mid E \mid F \mid T \mid X\big)
\end{array}
$$

The reduction rules can now be used to validate whether the process M exhibits the desired behavior following the informal UML2 semantics. We do not show any reductions here in detail, but informally summarize selected behavioral aspects of the process. The initial node, S, sends some message t via channel sa1. Process A reads a message from channel sa1, performs the process A and sends messages either over channels $a2b2$ and $a3c1$, or $a4d1$ or $a5e1$ depending on whether a process is available with which it can engage in a conversation. In this semantic mapping, we cannot say which of the channels will be chosen, because we leave this choice nondeterministic. This represents one abstraction that we encoded in the semantics and that can also be found in the UML2 specification [17, page 386]. Processes B, C, D, E behave similarly: they read from one of their input channels, perform their internal action, and then send a message on one of their output channels. Process F can read messages from three possible input channels where one of them is shared between the two possible behaviors ($d2f2$). As soon as a message can be read, one of the behaviors is enabled while the other is no longer possible. For example, if a message can be received via $c2f1$, the process transits to $\big(0 \mid d2f2(t)\big).\mathit{performF}.\overline{f4x}\langle t\rangle.0$, because only one of the behaviors composed by "+" is exercised, while the other is rendered void, cf. [26, page 39] and the reduction rule INTER. If afterwards a process tries to send a message via $e2f3$, F cannot read this message anymore because the required behavior is no longer available. After reading a message via $d2f2$, the process transits to $\mathit{performF}.\overline{f4x}\langle t\rangle.0$ and is not able to read any further messages.

The messages are not further specified, i.e., the ts are in fact completely unrelated to each other and could also be left off in the formalization above. We decided to leave them in only for reasons of readability.

Summarizing, we notice several other problems in addition to the incorrectly mapped activity final node T. Each π-process terminates after execution and is not ready to receive further inputs. In particular, B does not loop as expected from the informal semantics and only executes once. Besides this, processes such as F are loosing tokens. The following sections discuss these problems in more detail and provide possible solutions. As we will see, the main reason for the inadequateness of this semantics can be identified in the semantic mapping of object nodes.

6 Restrictions of the Initial Semantics and Semantic Challenges

In this section, we summarize the semantic features of UML2 activity diagrams and discuss to which extent they are addressed by the initial semantics. For each major semantic feature, a challenge is formulated that has to be addressed by any complete formalization.

Semantic Challenge 1 (Multiple Action Instances) *Each action has the potential of being instantiated multiple times within the same process instance depending on the actual token flow at execution time.*

In the example from Figure 1, B contains a "self-loop", i.e., an edge that leads from one of B's output pins back to one of its input pins. In order to execute the prefix $\overline{b3b1}\langle t\rangle$, a new instance of B must be ready to receive t. As already observed, the semantic mapping as defined in Definition 7 does not conform to the informal semantics of UML2, in particular for models with cycles. The mapping B' below shows a possible correction by adding the replication operator to produce an infinite number of instances of B in parallel before tokens have even been received by B:

$$B' \stackrel{def}{=} !B$$

Although this correction allows instances of B to communicate with each other, one may argue whether it correctly captures the semantics of activity instantiation in UML2, which assumes action instantiation to be triggered by token arrival. The replication operator, however, leads to infinitely many instances of an action executing in parallel based on the structural law $!P \equiv P|!P$.

> *"If a behavior is not reentrant, then no more than one execution of it will exist at any given time. An invocation of a nonreentrant behavior does not start the behavior when the behavior is already executing. In this case, control tokens are discarded, and data tokens collect at the input pins of the invocation action, if their upper bound is greater than one, or upstream otherwise. An invocation of a reentrant behavior will start a new execution of the behavior with newly arrived tokens, even if the behavior is already executing from tokens arriving at the invocation earlier."* [17, page 302]

The semantic mapping B'' captures reentrant behavior. The currently running instance of B'' instantiates in parallel another instance of action B'' while performing the action B''. This enables to instantiate a new action instance as soon as tokens arrive, even if another instance of this action is still executing. This means, in a reentrant behavior, we see multiple instances of one action executing simultaneously.

$$B'' \stackrel{def}{=} \big(a2b2(t) + b3b1(t)\big).\big(\mathit{performB''}.\overline{b3b1}\langle t\rangle \mid B''\big)$$

The above encoding is sufficient to capture the behavior of the process B in our example. For a general mapping based on this idea, one needs to know in advance how many instances of a process need to run in parallel. Unfortunately, this is not at all obvious in case of an arbitrary activity diagram. One may therefore return to the previous solution based on the replication operator, which works for an unknown number of instances. We implemented a simple example process of the form $A \stackrel{def}{=} (x(t).\mathit{performA}.0) \mid A$ causing infinite replication by exploiting the above mentioned structural law in the Mobility Workbench [38] as well as the Concurrency Workbench [8]. Both tools cannot simulate such a process and seem to generate an infinite transition system. However, if the number of required parallel instances is known, i.e., for example a process of the form $A \stackrel{def}{=} x(t).(\mathit{performA}.0 \mid A \mid A)$ is encoded, the process can be simulated, but the tools run out of resources when an analysis of the process is tried.

In case of *non-reentrant* behavior, only a single instance of an action is executing. For non-reentrant behavior, the following semantic mapping would be sufficient. In this mapping, a new instance of B''' can only be instantiated after a currently running instance of B''' has sent its output token and terminated.

$$B''' \stackrel{def}{=} \big(a2b2(t) + b3b1(t)\big).\mathit{performB'''}.\overline{b3b1}\langle t\rangle.B'''$$

However, in this specific example, B wants to communicate with itself, which is not possible in this encoding because of the synchronous communication. See later in this section for a more detailed discussion and Section 7 for an object-node semantics enabling asynchronous communication.

Semantic Challenge 2 (Multiple Activity Instances) *An activity model has the potential of being instantiated multiple times in parallel, once for each arriving set of input tokens. Each instance defines its own scope containing new instances of actions.*

The mapping of the activity diagram enables only a single instantiation of the activity M. This does not fully conform to the UML2 specification, which states that each activity diagram has the potential of being instantiated multiple times in parallel, one for each control token introduced to the system [17, page 259]. For our example, this requires that a new instance of process M is created when it receives the triggering control token from the environment. One potential solution to this problem is to define a wrapping process, $Wrap$, which creates a single instance of process M for each control token received from the outside source:

$$Wrap \stackrel{def}{=} input(start).(M|Wrap)$$

For each control token received on the channel $input$, process $Wrap$ creates an instance of process M in parallel with another instance of the wrapping process. Therefore, $Wrap$ is able to generate an unknown number of parallel instantiations of process M. As we pointed out earlier, using the replication operator ! may not be a possible alternative as it does not reflect the semantics of activity instantiation correctly and seems to prohibit any analysis in existing tools.

The two challenges that we discussed so far could be easily addressed by minor modifications of the initial semantic mappings for action nodes and activity diagrams. The following challenges will require significant changes of the semantic encoding of object nodes. The UML2 semantics assumes that tokens are offered on an output pin to the receiving action and consumed *all-at-once* when this action is ready. Our formalization leads to a simplified semantics where tokens immediately flow to the end of an edge and wait there until all other tokens of the same parameter set have arrived. Since we assume that edges connect pins 1-1, this simplification does not affect the execution of the activity diagram as there is no nondeterminism in the flow of a single token, which can only flow to one designated pin. In case of multiple edges leaving an output pin, there would be a difference as the action that is first ready to receive, determines where the tokens flow. We argue that the original UML2 semantics may lead to many unexpected behaviors that are hard to detect, in particular because parts of the diagram may not be executable as actions always 'come too late'.

Semantic Challenge 3 (Token Preservation) *Control and object tokens not consumed by an action must remain in the input channel for further instantiations of this action.*

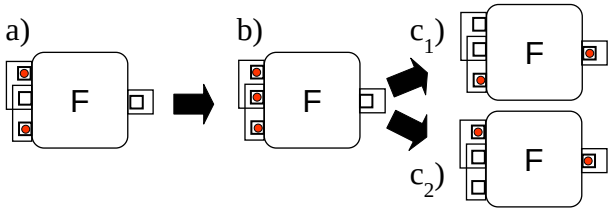

Figure 2: Token flow behavior of action F.

Figure 2 illustrates the anticipated semantics of action F in the case that one token has been received by each of the input parameter sets and only the shared pin still waits for a token as shown in Situation a). When this token arrives as shown in Situation b), each of the parameter sets is enabled and exactly one of them can trigger the execution of action F. After finishing the execution, F emits a token on its

output pin, but preserves the token in the non-triggering input parameter set, i.e., either Situation c_1 or c_2 should occur.

However, this behavior is not possible in the π-process F. The capability of F to read again from $d2f2(t)$ is no longer available once a token on the shared pin $f2$ has been received as we have already pointed out in the previous section. The semantics we have discussed so far provides no solution to the Token Preservation challenge. UML2 defines even more advanced features of object nodes that we want to briefly discuss in the next challenges.

Semantic Challenge 4 (Streaming, buffering, upper bounds, and multiplicities) *Pins can exhibit the buffering behavior of queues and stacks of limited and unlimited capacity. In particular, when they are streaming, they can receive and send tokens combined as bundles of defined multiplicity while the action is executing.*

Streaming allows an action to take inputs and provide outputs while it is executing. During one execution, the action may consume multiple tokens on each streaming input and produce multiple tokens on each streaming output [17, page 302]. Upper bounds on pins define the maximum number of tokens it can hold. Multiplicities on the pins are used to define the number of tokens that are necessary for an action to execute. For example, an action A may require three tokens before it can start executing. If an output pin of an action C sends three tokens, then A would execute once, however, an action B requiring only a single token would execute three times. Similarly to the token preservation behavior, this behavior requires a more precise semantics for object nodes. We immediately see that encoding pins with message reading and sending operations is not enough, but they have to be encoded as π-processes themselves. This is the basic idea for the semantics in the next section. Milner has shown how buffers are encoded with π-processes in [15].

Semantic Challenge 5 (Global Termination and Exceptions) *Execution of an activity final node terminates all actions and nested activities in the scope of the structured activity node containing the activity final node. All tokens within this scope have to be discarded.*

Global termination behavior also occurs through exception actions and in interruptible activity regions.

> *"Interruptible activity regions are groups of nodes within which all execution can be terminated if an interruptible activity edge is traversed leaving the region. Raising the exception terminates the immediately containing structured node or activity and begins a search of enclosing nested scopes for an exception handler that matches the type of the exception object. ... If an exception occurs during the execution of an action, the execution of the action is abandoned and no regular output is generated by this action. If the action has an exception handler, it receives the exception object as a token. If the action has no exception handler, the exception propagates to the enclosing node and so on until it is caught by one of them. If an exception propagates out of a nested node (action, structured activity node, or activity), all tokens in the nested node are terminated."* [17, pages 259, 310]

In order to capture the semantics of activity final nodes, exceptions and interruptible regions, the semantic mapping has to be further extended. We can immediately see that these behaviors require not only to encode pins as processes, but in addition also to encode an additional "communication infrastructure" between the actions requiring additional processes and thus further increasing the size of the resulting semantic mapping. We found two examples in the literature, where such a communication infrastructure has been formalized recently. A solution for services based on the π-calculus is presented in [18], which allows a service to react to an intermediate abort event. A formalization of so-called cancellation regions in workflows based on Reset Workflow nets is presented in [42].

Finally, we want to discuss the underlying scheme of communication in activity diagrams, which can be a very flexible mix of asynchronous and synchronous behavior.

Semantic Challenge 6 (Asynchronous vs. Synchronous Communication) *Any form of asynchronous and synchronous communication between actions is encodable in activity diagrams.*

Communication in the π-calculus is synchronous, i.e., a message can only be sent from the sender when the receiver is ready to receive it. The simplified semantics reflects such a synchronous behavior. In order to encode asynchronous communication, pins must be again encoded a π-calculus processes, which can buffer tokens.

The UML2 specification talks about the assumed communication behavior in different places and somehow leaves "all options open". The semantics of edges is informally defined with the following token flow rules:

> *"Edges have rules about when a token may be taken from the source node and moved to the target node. A token traverses an edge when it satisfies the rules for target node, edge, and source node all at once. This means a source node can only offer tokens to the outgoing edges, rather than force them along the edge, because the tokens may be rejected by the edge or the target node on the other side. Multiple tokens offered to an edge at once is the same as if they were offered one at a time. Since multiple edges can leave the same node, token flow semantics is highly distributed and subject to timing issues and race conditions, as is any distributed system. There is no specification of the order in which rules are applied on the various nodes and edges in an activity. It is the responsibility of the modeler to ensure that timing issues do not affect system goals, or that they are eliminated from the model."* [17, page 309]

The specification also mentions two further features of the communication within activity diagrams, which are referred to as *traverse-to-completion* and consumption of tokens *all-at-once*, which we already briefly mentioned when discussing the token-preservation challenge.

> *"Traverse-to-completion means that tokens move along the path of least resistance by going to the first available object node. ...Another behavior that falls under traverse-to-completion is the transformation of tokens as they move across an object flow edge. ...Another aspect of traverse-to-completion is that an input pin of an action cannot accept tokens until all the input pins of the action can accept them. This is to prevent deadlock, where the input pins of two actions each have some of the tokens required for the other to start."* *[6, pages 35-37].*
>
> *"The flow prerequisite is satisfied when all of the input pins are offered tokens and accept them all at once, precluding them from being consumed by any other actions. This ensures that multiple action executions competing for tokens do not accept only some of the tokens they need to begin, causing deadlock as each execution waits for tokens that are already taken by others."* [17, page 302].

Obviously, these features add further complexity to any semantics and one is wondering to which extent they should or can be captured, see also the discussion in [33]. Given our insights so far, one may even advocate for a revision of the UML2 specification. We also argue that objects should not change type while flowing along edges, in particular for reasons of clarity and readability of diagrams in practical applications. Consumption of tokens *all-at-once* seems to be encodable by adopting a synchronous communication behavior that ensures that the hand-shake takes place not only between single pins, but

for entire parameter sets. We have no proposal for how the notion of "*least resistance*" should be adequately encoded in order to achieve a traverse-to-completion behavior that rules out many interleavings of process executions that would occur in the corresponding π-process. When implementing our encoded model in the Mobility Workbench we obtained a trace showing that process A interacted with process C and subsequently process C interacted with process F before any interaction between processes A and B happened. This differs from the expected UML2 behavior where A streams out both tokens to B and C simultaneously, i.e., A interacts with B and C before these can start any other interactions. The role of these semantic subtleties in the context of model verification needs to be further investigated.

7 A Semantics of Object Nodes

The semantic challenges from the preceding section demonstrated that a mapping of pins and parameter sets to simple action prefixes is not sufficient. Instead, they must be mapped to rather complex processes that can buffer messages and participate in a complex communication structure to correctly reflect the activation of UML2 activities. Figure 3 illustrates the communication structure that must be represented in the π-calc

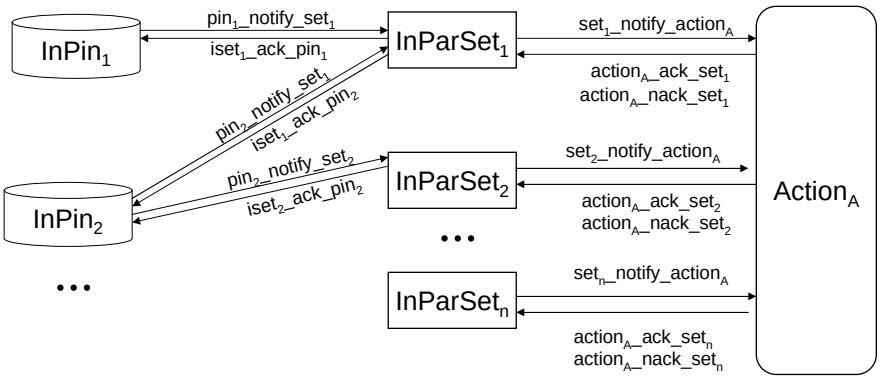

Figure 3: Communication structure of processes in the object-node semantics.

The main idea is as follows: Each pin is mapped to a pin process, which buffers the token messages in a queue. An input parameter set is itself a process that communicates with all of its pin processes and with the action. We introduce a separate communication channel for each direction of communication, which allows us to abstract from the messages that are exchanged. This means, in the following definitions we use the short-hand notations for the sending and receiving over a channel, i.e., do not show message names. The input parameter set process records the status of all its pins. If all pins have received at least one token, it is completely notified. Each "completely-notified" input parameter set process then notifies the action process. The action process selects one of the parameter sets to complete. The selected parameter set causes all its pin processes to remove one token. Now, the action process can execute and communicate with one of the output parameter set processes.

In the following, we provide formal definitions for all required processes encoding the pins, parameter sets and the action, and then extend our example.

Definition 11 (Mapping of an Input Pin) *Let p be an input pin contained in input parameter sets $\mathcal{I}(p)$. Let e be the incoming edge of p. The mapping $\pi(p)$ is defined by the following processes:*

$$
\begin{aligned}
InPin_p(0) &\stackrel{def}{=} e.\Big(\prod_{I_k \in \mathcal{I}(p)} \overline{pin_p\_notify\_set_k}\Big).InPin_p(1) \\
InPin_p(n) &\stackrel{def}{=} \Big(\sum_{I_k \in \mathcal{I}(p)} iset_k\_ack\_pin_p\Big).InPin_p(n-1) \\
&\quad + e.InPin_p(n+1) + \Big(\sum_{I_k \in \mathcal{I}(p)} \overline{pin_p\_notify\_set_k}\Big).InPin_p(n) \qquad \text{for } n \geq 1\ .
\end{aligned}
$$

The process for an input pin behaves as follows: A token is received, all owning parameter sets are notified via the $pin_p\_notify\_set_k$ channel, and the $InPin_p$ process transitions to the state where one token is stored. The process is then ready for an acknowledgment by any owning parameter set, expressed by $iset_k\_ack\_pin_p$ or for receiving another token. In the latter case, the process stores another token, in the former case, the process transitions to a lower state (meaning that one token is removed). Alternatively, it can also send further notifications to the owning parameter sets, expressed by $pin_p\_notify\_set_k$, for renotification purposes.

The pin process for an output pin waits for a notification by one of its parameter sets (modeled by $oset_l\_notify\_pin_q$) and then sends out the token:

Definition 12 (Mapping of an Output Pin) *Let q be an output pin contained in output parameter sets $\mathcal{O}(q)$. Let e be the outgoing edge of q. The mapping $\pi(q)$ is defined by the following process:*

$$
OutPin_q \stackrel{def}{=} \Big(\sum_{O_l \in \mathcal{O}(q)} oset_l\_notify\_pin_q\Big).\overline{e}.OutPin_q
$$

The next definitions capture the communication between the parameter sets and their pins and the action. An input parameter set can be in different states: *initialized* if it has not yet received a notification from any pin in the parameter set, *partially notified* if some of its pins have already notified, but not all, or *completely notified* when all pins have received at least one token and notified the paramater set. In state *initialized*, modeled by the process $InParSet_k(0)$, it is ready to receive a notification from one of its pins and transitions to the *partially notified* state where this pin has sent a notification, modeled by the process $InParSet_{k,Notified=\{p\}}$. In state *partially notified*, modeled by processes $InParSet_{k,Notified \neq I_k}$, the parameter set waits for a notification from one of those pins that are not included in the set of pins that have already notified. If it is notified by a pin p via $pin_p\_notify\_set_k$, it transitions to a process $InParSet_{k,Notified+\{p\}}$, meaning that also pin p has received a token and therefore, has sent a notification. If all pins have notified, the parameter set transitions to *completely notified*, modeled by the $InParSet_{k,Notified=I_k}$ process. In this state, the parameter set notifies the action that it is ready to complete and then calls itself. After the notification of the action, the parameter set process can receive an $action_A\_ack\_set_k$, denoting that this parameter set has been selected to complete. Then it notifies its pins via $\overline{iset_k\_ack\_pin_p}$.

In all states, the parameter set process can receive some message over $action_A\_nack\_set_k$ meaning that the action has chosen another parameter set to complete. In such a case, the parameter set needs new notifications from all its pins because some of them could have been notified and removed tokens by the alternative parameter set completion. Remember that a pin can belong to more than one parameter set.

Definition 13 (Mapping of an Input Parameter Set) *Let I_k be an input parameter set belonging to some action A. The mapping $\pi(I_k)$ is defined as*

$$
\begin{aligned}
InParSet_k(0) \stackrel{def}{=} & \Big(\sum_{p \in I_k} pin_p\_notify\_set_k.InParSet_{k,Notified=\{p\}} \Big) \\
& + action_A\_nack\_set_k.InParSet_k(0) \\
InParSet_{k,Notified \neq I_k} \stackrel{def}{=} & \Big(\sum_{p \in I_k, p \notin Notified} pin_p\_notify\_set_k.InParSet_{k,Notified+\{p\}} \Big) \\
& + action_A\_nack\_set_k.InParSet_k(0) \\
InParSet_{k,Notified=I_k} \stackrel{def}{=} & \ \overline{set_k\_notify\_action_A}.InParSet_{k,Notified=I_k} \\
& + action_A\_ack\_set_k.\Big(\prod_{p \in I_k} \overline{iset_k\_ack\_pin_p} \Big).InParSet_k(0) \\
& + action_A\_nack\_set_k.InParSet_k(0)
\end{aligned}
$$

The process for an output parameter set waits for a notification by the action and then notifies all its pins.

Definition 14 (Mapping of an Output Parameter Set) *Let O_k be an output parameter set of an action A. The mapping $\pi(O_k)$ is defined as*

$$
OutParSet_k \stackrel{def}{=} action_A\_notify\_set_k.\Big(\prod_{q \in O_k} \overline{oset_k\_notify\_pin_q} \Big).OutParSet_k
$$

The mapping $\pi(A)$ of an action A introduces parallel processes for each of its input and output parameter sets as well as its input pins and output pins and initiates an *InternalAction* process, which waits for a notification from one of its input parameter sets and then notifies this set of having been selected. It also notifies all other parameter sets that they have not been chosen. Afterwards it performs the $performA$-transition and sends a notification to the output parameter set that is correlated to the input parameter set that was activated. Note that we use a correlation mechanism in order to be able to model that a certain input parameter set always gives rise to the activation of a specific output parameter set. Alternatively, we could nondeterministically choose the output parameter set.

Waiting for a notification by an input parameter set is modeled as a nondeterministic choice over all input parameter sets. It is possible that more than one input parameter set can complete. If this is the case, one of them is selected nondeterministically. Alternatively, a *leader election* [2] algorithm can be encoded in the semantic mapping to ensure fairness of the parameter set selection.

Definition 15 (Mapping of an Action) *Let A be an action. Let $\mathcal{I}(A)$ be the input parameter sets of A, let $\mathcal{O}(A)$ be the output parameter sets of A, let $ipins(A)$ be all the input pins of A and let $opins(A)$ be all the output pins of A. Let $corr(I_k)$ be a function that returns the index of the output parameter set that is correlated to input parameter set I_k. The mapping $\pi(A)$ is defined as*

$$
\begin{aligned}
Action_A \stackrel{def}{=} & \Big(\prod_{p \in ipins(A)} InPin_p(0) \Big) \mid \Big(\prod_{I_k \in \mathcal{I}(A)} InParSet_k(0) \Big) \mid InternalAction_A \mid \\
& \Big(\prod_{O_l \in \mathcal{O}(A)} OutParSet_l \Big) \mid \Big(\prod_{q \in opins(A)} OutPin_q \Big)
\end{aligned}
$$

with

$$InternalAction_A \stackrel{def}{=} \sum_{I_k \in \mathcal{I}(A)} \Big(set_k\_notify\_action_A.\overline{action_A\_ack\_set_k}.\big(\prod_{I_i \in \mathcal{I}(A), i \neq k} action_A\_nack\_set_i \big). performA.\overline{action_A\_notify\_set_{corr(I_k)}}.InternalAction_A \Big)$$

We partially sketch the encoding of action F below, using again the short-hand notations of Section 3:

$$\begin{aligned}
InPin_1(0) &\stackrel{def}{=} c2f1.\overline{pin_1_notify_set_1}.InPin_1(1)\\
InPin_2(0) &\stackrel{def}{=} d2f2.\big(\overline{pin_2_notify_set_1} \mid \overline{pin_2_notify_set_2}\big).InPin_2(1)\\
InPin_3(0) &\stackrel{def}{=} e2f3.\overline{pin_3_notify_set_2}.InPin_3(1)\\
InPin_1(n) &\stackrel{def}{=} iset_1_ack_pin_1.InPin_1(n-1) + c2f1.InPin_1(n+1)\\
&\quad + \overline{pin_1_notify_set_1}.InPin_1(n)\\
&\vdots\\
InParSet_1(0) &\stackrel{def}{=} pin_1_notify_set_1.InParSet_{1,\{p_1\}} + pin_2_notify_set_1.InParSet_{1,\{p_2\}}\\
&\quad + action_F_nack_set_1.InParSet_1(0)\\
InParSet_{1,\{p_1\}} &\stackrel{def}{=} pin_2_notify_set_1.InParSet_{1,\{p_1,p_2\}} + action_F_nack_set_1.InParSet_1(0)\\
InParSet_{1,\{p_2\}} &\stackrel{def}{=} pin_1_notify_set_1.InParSet_{1,\{p_1,p_2\}} + action_F_nack_set_1.InParSet_1(0)\\
InParSet_{1,\{p_1,p_2\}} &\stackrel{def}{=} \overline{set_1_notify_action_F}.InParSet_{1,\{p_1,p_2\}} + action_F_nack_set_1.InParSet_1(0)\\
&\quad + action_F_ack_set_1.(iset_1_ack_pin_1 \mid iset_1_ack_pin_2).InParSet_1(0)\\
&\vdots\\
F &\stackrel{def}{=} InPin_1(0) \mid InPin_2(0) \mid InPin_3(0) \mid InParSet_1(0) \mid\\
&\quad InParSet_2(0) \mid InternalAction \mid OutParSet_1 \mid OutPin_4
\end{aligned}$$

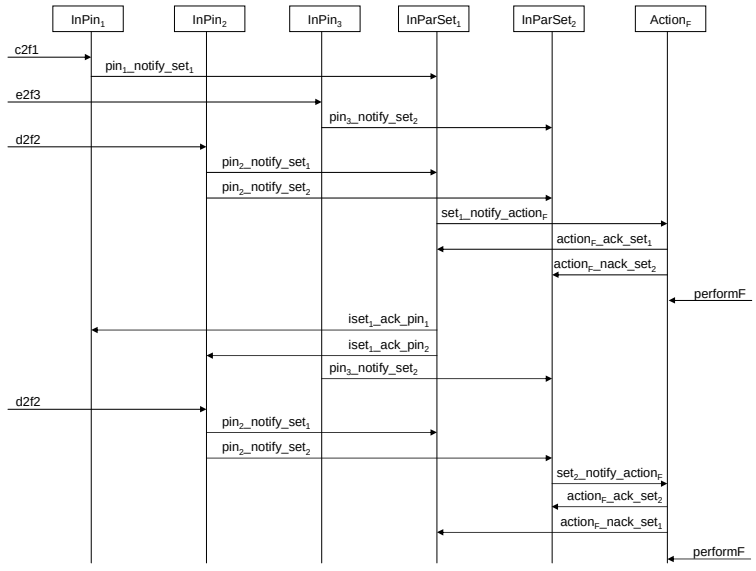

Figure 4: Communication of processes related to action F in the object-node semantics.

Figure 4 shows one possible exchange of messages between the action F, its three input pin processes, and their parameter set processes. Upon receiving over $c2f1$, the process $InPin_1$ sends a notification to its parameter set $InParSet_1$. $InPin_2$ and $InPin_3$ also send such a notification to $InParSet_1$ set once they receive over $e2f3$ and $d2f2$, respectively. $InPin_2$ also notifies its other parameter set

$InParSet_2$. As $InParSet_1$ is the first to notify $Action_F$, it receives an $action_F\_ack\_set_1$ to denote that it has been chosen to complete. $InParSet_2$ receives an $action_F\_nack\_set_2$ to denote that it has not been chosen. $InParSet_1$ sends acknowledgements to its contained pins. Afterwards, both $InParSet_1$ as well as $InParSet_2$ are in the initialized state again and are ready to receive notifications from their pins. $InPin_3$ still has a token and therefore notifies its parameter set again. Upon receiving over $d2f2$, $InPin_2$ will again notify both its parameter sets. This time, $InParSet_2$ is completely notified and notifies $Action_F$, which will send an acknowledgement to $InParSet_2$ and a negative acknowledgement to $InParSet_1$.

The previous definitions have shown that the semantics of object nodes can be captured in the π-calculus, but at the price of adding many additional processes to the semantic encoding, which need to engage in a quite complex communication. So far, we have not addressed advanced features such as multiplicities and streaming. Multiplicities could be encoded by changing an $InPin$ process in such a way that it only sends the notification if it has received enough tokens. Streaming probably requires a different communication infrastructure because tokens must be released immediately. We used the Concurrency Workbench [8] to validate the semantic encoding of object nodes and applied it to the action F in our example. This enabled us to study the resulting transitions and validate whether the intended communication between pins, parameter sets and the action indeed occurs. The validation also confirmed that a communication between the action and the parameter sets that have *not* been selected to complete is indeed necessary. For the case of two overlapping parameter sets, the completion of one parameter set also affects the number of tokens available to the other parameter set through their shared pins. This means, all pins that still have a token after one of the parameter sets has completed, need to renotify all their owning sets in order to allow the set to update its status of notifications. The results of our practical experiments are briefly summarized at the end of the next section where we show encouraging evidence that the theoretically possible explosion on the number of transitions does not necessarily occur in practice.

Given the obvious complexity of the presented encoding, the question remains whether other semantic encodings would lead to smaller transition systems. Answering this question would go significantly beyond the scope of this paper, which focuses on clarifying the semantic nature of the various modeling elements. We believe that searching for more compact encodings is an interesting question for future research and that contrasting π-calculus encodings of object nodes with Petri net encodings could yield further interesting insights. An initial treatment of object nodes in Petri nets has been presented in [31], which requires to use Colored Petri nets, i.e., a higher-order Petri net formalism, although properties such as overlapping parameter sets sharing a pin are not yet discussed.

8 A Classification of Activity Diagrams

The discussion in Section 6 identified three dimensions of semantic complexity in the activity diagrams distinguishing the communication behavior, the concurrency of actions, and the number of possible action instances. In the following, we will define semantic classes of activity diagrams based on these three dimensions, then give an initial syntactic characterization, and finally discuss practical aspects of their underlying transition system.

8.1 Semantic Classes of Activity Diagrams

We observed that several modeling elements may lead to multiple instantiations of actions. Secondly, action instances can execute sequentially or concurrently. By sequential execution we understand that only one single action instance is active at any point in time. Thirdly, we observed different communication schemes between action instances. In the simplest case, communication is synchronous and pins can be mapped to message sending and receiving actions. A more complex semantics is required when

actions communicate asynchronously, because this requires to buffer tokens in object nodes (in particular, pins), which led to the semantic mapping of pins to π-processes. Besides this, modeling elements like the activity final node require to encode additional processes that enable a controlled broadcasting between nodes in the activity diagram, which can occur in a synchronous or asynchronous form. These three dimensions are summarized in Figure 5 where they provide the basis for a semantic classification of activity diagrams.

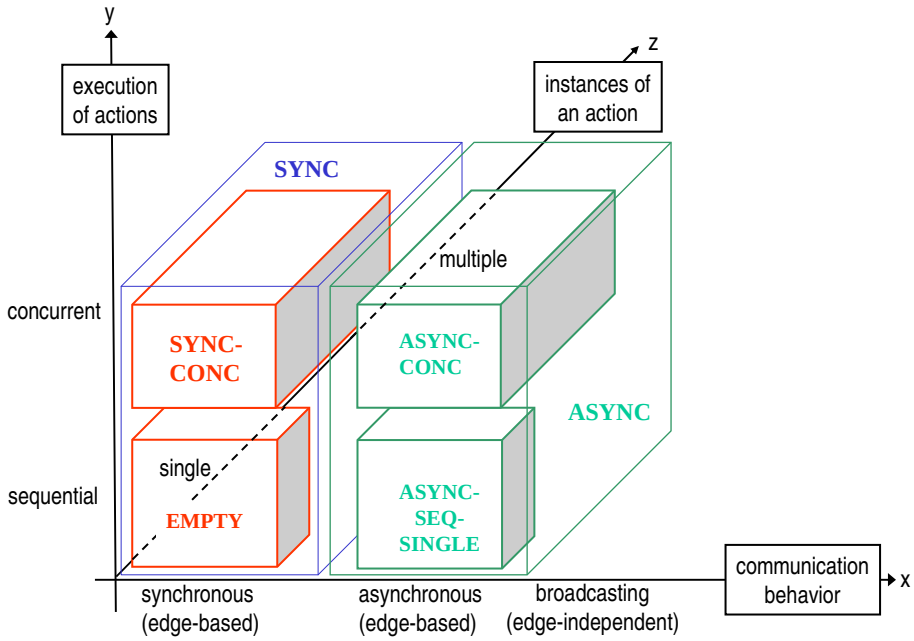

Figure 5: A semantic classification of activity diagrams.

Overall, twelve different semantic classes can be built based on the three dimensions. The figure shows five explicit classes. We consider the last four of them as being of particular interest for future research:

- SYNC-SEQ-SINGLE(=EMPTY): This class combines *synchronous* communication between the actions with sequential execution of action instances such that only a single instance of each action is created over the lifetime of the process.
- ASYNC-SEQ-SINGLE: This class combines *asynchronous* communication between the actions with sequential execution of action instances such that only a single instance of each action is created over the lifetime of the process.
- SYNC-CONC: This class covers synchronous communication between action instances executing concurrently.
- ASYNC-CONC: This class permits asynchronous communication between concurrently executing action instances.
- BROADCAST-CONC: This class combines broadcasting with the concurrent execution of action instances. This class includes asynchronous or synchronous edge-based communication, which can be combined with asynchronous or synchronous broadcasting communication, i.e., it could also be refined into further subclasses.

The class EMPTY contains all activity diagrams with single instances, sequential communication and synchronous communication. We have chosen this name as this class seems to contain no executable diagrams. A single action instance, which wants to engage in a synchronous (hand-shaking) communication, but has no communication partner, cannot execute. From a syntactical point of view, however, this class could contain non-executable, rather pathological diagrams, e.g., those containing only disconnected nodes.

Our initial semantic mapping is sufficient for the class SYNC-CONC, because the π-calculus is inherently concurrent. In case that we only support single instantiation of actions, the initial semantic mapping falls into the class SYNC-CONC-SINGLE, in case that we support multiple instantiations it falls into the class SYNC-CONC-MULT. For the class ASYNC-SEQ-SINGLE, a different process-algebraic formalization can be defined where—different than in our initial semantic mapping—actions are not executed concurrently but sequentially, see [33] for a draft formalization. Our semantics supporting object nodes falls into the class ASYNC-CONC: the buffering allows for asynchronous communication between the actions and we have chosen concurrent execution of all actions. As we also support multiple instantiations, the semantics actually falls into the class ASYNC-CONC-MULT.

Our own practical experience in the area of business process modeling in the insurance and banking domain showed that approximately 50 % of all activity diagrams fall into the class ASYNC-SEQ-SINGLE. The majority of the remaining diagrams contains cycles and can therefore lead to multiple instantiations of certain actions. Broadcasting occurs rather seldomly in our domains. Furthermore, encoding global termination is not necessary for many models. Often, an activity final node can be replaced by a flow final node and still captures the intention of the designer. We argue that activity final nodes only need to be encoded explicitly if concurrent executions can occur in a model.

The example in Figure 1 falls into the class BROADCAST-CONC because it contains a termination node which requires a broadcasting mechanism. If the termination node is replaced by a flow final node, it falls into class ASYNC-CONC. A closer look at the execution traces of this model reveals that asynchronous communication is not required because no execution trace contains the action F (there is actually a deadlock here). As a consequence, in this particular case, the model can even be correctly analysed in class SYNC-CONC. Below, we elaborate on this idea and formulate several hypotheses stating when an activity diagram can be correctly encoded in one or the other dimension.

8.2 Syntactic Characterizations

According to the UML2 semantics, concurrency of actions, asynchronous and synchronous communication, as well as multiple instances of actions can all occur within a single diagram. In principle, any formalization that can capture these properties will be suitable. Nevertheless, it is desirable to identify those models where not all semantic features are needed, because the execution of the activity diagram does not lead to multiple instantiations or does not require asynchronous communication. This means that these activity diagrams could also be adequately encoded *without* the possibility of multiple instantiations and asynchronous communication which (in our case) has led to smaller sizes of the underlying transition systems as we will show at the end of this section. In the following, we formulate several hypotheses that help to decide which type of semantic encoding is necessary given the absence or presence of syntactic constructs.

Hypothesis 1 (Single instances/Multiple Instances) *An activity diagram can be encoded semantically correct using* single instances *of an action if it does* <u>not</u> *contain*
- more than one start node and
- incomplete[3] *implicit or explicit forks and*

[3] A fork is incomplete if it is not followed by a join of all its outgoing branches. This requires a form of wellformedness, which remains to be defined precisely. A fork is implicit if it is modeled with pins and parameter sets, but not using a fork

- cycles and
- multiplicities defined for pins and
- joins of object flow.[4]

In the presence of any of the above constructs, it usually requires a semantic encoding supporting multiple instances of the same action.

We argue that the previous hypothesis formulates a sufficient condition for the possibility to encode an activity diagram correctly using single instances. We want to point out that the condition is not a necessary one, i.e., there are activity diagrams that do contain multiplicities and that can still be correctly encoded using single instances. The syntactic constructs for cycles, incomplete forks, multiplicities and joins of object flow all give rise to the possibility that more than one instance of an action is created during the execution of the activity diagram. If not every fork is followed by a corresponding join, then this leads to multiple instantiations. For example, in Figure 6 a), action D is instantiated twice according to the UML2 semantics because the implicit fork at A is not followed by a join. In Figure 6 b), action B is instantiated twice. A similar example can be constructed using explicit forks. In the presence of cycles, several instances of the same action are created during execution of the activity diagram. An example is given in Figure 6 c) where actions A and B are instantiated several times.

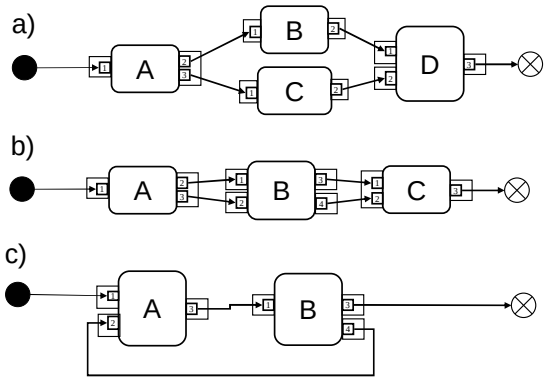

Figure 6: Pins acting as implicit forks causing multiple action instantiations.

Hypothesis 2 (Synchronous/Asynchronous communication) *An activity diagram can be encoded semantically correct using* synchronous *communication along the edges if*
- each parameter set contains at most one pin and
- it does not contain a global termination node and
- it does not contain a broadcasting action.

An activity diagram can be encoded semantically correct using asynchronous *communication along the edges if it does not contain*
- a global termination node and
- a broadcasting action.

We argue that the previous hypothesis formulates sufficient conditions when an encoding using synchronous and asynchronous communication is appropriate. Again, these conditions are not necessary conditions. For example, there may be activity diagrams that can be encoded semantically correctusing

control node.

[4]A join offers all data tokens on its outgoing edge, see [17, page 369].

asynchronous communication if there are broadcasting actions within the diagram, e.g., if these broadcasting actions are never executed. Given these two hypotheses, we can deduce hypotheses for each of the twelve classes. However, for each class the hypotheses need to be carefully checked if they are not too restrictive and to which extent they can be relaxed. We have to leave this question to future research as well as the formal proofs of the previous two hypotheses or any improved versions thereof.

For our second dimension, the concurrency of actions along the y-axis, we do not have a hypothesis. The distinction between sequential and concurrent execution leads to the question whether or not an activity diagram needs to be semantically encoded using concurrency or not. If no mutual communication dependencies between two concurrently executing action instances exist, then concurrent execution can be replaced by sequential execution. We have the impression that the distinction along the concurrency dimension leads to a different problem: How to calculate correct sequentializations of concurrent executions? Future research also needs to clarify if a purely syntactic characterization is possible, i.e., to which extent a class can be defined through the simple absence or presence of certain modeling elements or additional model constraints. Furthermore, the theoretical properties of the classes are of great interest. Questions such as the decidability of the reachability problem, the size of the resulting transition system and the effect of various semantic encodings on the size of the transition system are of high practical importance. It would also be of interest to define precise containment relationships between the classes or to prove the absence of those relationships, i.e., to work towards a class hierarchy for activity diagrams.

8.3 Practical Findings

We have already briefly mentioned that our semantic encoding of object nodes in the π-calculus leads large transition systems and a more compact encoding—in the π-calculus or another semantic domain—could be desirable. A possible approach to avoid uncontrolled explosion of the size of the resulting transition system is to apply the initial semantic mapping whenever possible and to use the more sophisticated semantic mapping only for those actions and object nodes, for which it is required. In the following, we want to summarize some of our practical findings. As our semantic encodings do not use the concept of mobility, we implemented the example process in CCS and used the Concurrency Workbench [8] to validate the semantics and to determine the size of the resulting transition systems.

We encoded several versions of our example: Version 1 is the mapping as described in Section 4. Version 2 is the same mapping except that we encoded action F using the semantics of object nodes, as described in Section 7. Version 1 gives rise to 23 states and 28 transitions and can be minimized to 10 states with 13 transitions. Version 2 gives rise to 105 states and 202 transitions, after minimization according to bisimulation 10 states and 12 transitions remained. By having a closer look at the example it becomes obvious that in this particular case the overlapping parameter sets at action F are never completing because only one token will arrive at action F.

In order to study the size of the transition system underlying action F, we encoded action F without any other actions. This encoding of action F gives rise to 20441 states and 82361 transitions. After minimization according to bisimulation still 2298 states and 9005 transitions remained. We also simulated the full behavior of action F in another example, shown in Figure 7. This example gives rise to 16609 states and 59244 transitions and after minimization to 65 states and 144 transitions. Note that in this example action F can execute twice: It first receives one token on pin 1 and pin 3 and then it continuously receives tokens on pin 2.

Our encodings show that the semantic encoding of object nodes, which might appear complex, does not necessarily lead to an explosion of the size of the underlying transition system. Techniques such as minimization show a dramatic reduction effect. Furthermore, there is a huge difference between F behaving in isolation and F being wired within some activity diagram. As already discussed in Section 6, any infinite loop will lead to aninfinite transition system that can currently not be handled by the tools.

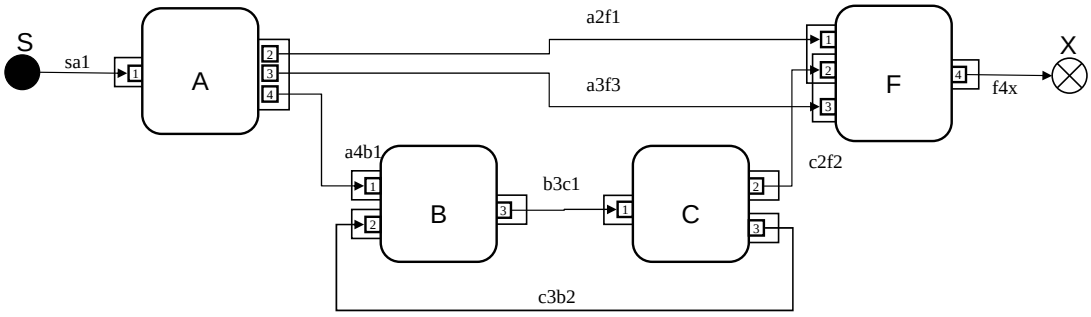

Figure 7: Example showing full behavior of action F.

9 Related Work

In the literature, a number of papers address the problem of formalizing UML activity diagrams. Early work deals with formalizing UML1.x activity diagrams: Boerger et al [7] propose a semantics of UML activity diagrams in Abstract State Machines, which is influenced by the original joint metamodel of state machines and activity diagrams in UML. Dong and ShenSheng [9] present a π-calculus semantics for UML 1.4 activity diagrams. Their processes are composed in parallel, which enables them to communicate with each other. In this way, more complex behaviors can be captured where more than one activity in a business process can be active. However, many other features remain unaddressed. Similarly, the profound work by Eshuis and Wieringa [11], is not directly applicable, because it focuses on UML 1.4, which does not have parameter sets and the rather subtle token-flow semantics. A formal semantics for UML2 is the main goal of the UML2.0 Semantics project [35], however, no results addressing behavioral models have been published so far.

Interesting discussions have been published on the π-calculus as an appropriate foundation for business process models. While some enthusiastically praise the advantages of using the π-calculus as the foundation of the Business Process Modeling Language (BPML) [28, 27], but do not give any precise mapping from BPML to the π-calculus, others [37, 36, 22] are much more critical regarding this "π-hype". The argumentation in [37, 36, 22] is based on a semantic mapping, which constructs one π-process for the entire business process model. As this is not sufficient and much better semantic mappings exist as we have shown, the arguments are no longer valid. A very profound argumentation has recently been published in [19]. The authors of [22] point out that "the π-calculus is an analytical tool for understanding" languages, but do not demonstrate how such an analytical tool can be exploited in a real-world context. In this report, we tried to use the π-calculus as such an analytical tool that helped us to develop a possible classification of activity diagrams, which we consider as being orthogonal to the mapping of activity diagrams to the workflow patterns as presented in [41, 25].

Puhlmann and Weske [20] use the π-calculus for formalizing workflow patterns. In their recent work [21], a lazy soundness for workflows is proposed and it is discussed how this can be checked in a π-calculus formalization. Their formalization, see in particular [18], is very similar to ours, which is not surprising given the semantic foundations of workflows and UML activity diagrams. Our work is complementary as it focuses on the clarification of the semantic subtleties in activity diagrams and contributes a classification of activity diagrams based on semantic expressivity. In our formalization of object nodes, we go beyond the formalization in [20, 18], who only consider action prefixes for the communication between the considered workflows, while we introduced separate processes in the object-node semantics.

As [20, 18] we also use the replication operator to capture multiple instances of a process, but we discuss the theoretical and practical problems such a formalization can cause and present possible al-

ternatives. We do not yet address specific verification problems as those discussed in [21]. Lucchi and Mazzara [13] provide a π–calculus semantics for BPEL. As BPEL desribes the orchestration of Web services, where each Web service can only receive and send a single message, their semantics does not cover most of the semantic challenges of UML2 activity diagrams. Another formalization of BPEL based on the π-calculus is presented in [43] and model checking scenarios are investigated.

An almost complete formalization of UML2 activity diagrams using Petri nets is described by Störrle. He applies procedural Petri nets to formalize control flow [29], data flow [31], exceptions and structured nodes [30]. Recently, Störrle and Hausmann [33] have identified several problems when formalizing UML2 activity diagrams with Petri nets. They have shown that in principal, a Petri net formalization suffers from several problems such as inadequate support for streaming and traverse-to-completion. This corresponds to our results, where we also identified these properties as especially difficult to formalize and even argued whether the UML2 semantics should be revised. Hausmann [12] contains a comprehensive discussion of the semantics of UML2 activity diagrams, which he uses as a case study for the Dynamic Meta Modeling [10] approach. This approach can be used to define an operational semantics by defining the transition rules for visual modeling languages. The main focus in [12] is on the traverse-to-completion semantics of activity diagrams, while we argue that restricting edges between pins to 1-1 edges and assuming a cleanly specified communication behavior may be a more promising direction for a formalization. While Hausmann investigates arbitrary edges between pins, parameter sets are not addressed in the same detail as we have discussed them. His approach is well-suited for a simulation, but does not provide very strong analytical capabilities. This work can be seen as complementary to our work which aims at a denotational semantics of UML2 activity diagrams and a better understanding of the modeling concepts that add semantic complexity.

Another operational semantics of UML2 activity diagrams is described by Vitolins and Kalnins [39]. They focus on the token game and introduce push and pull paths for tokens in the diagram. The operational semantics is described in pseudo code and a semiformal proof of equivalence between their semantics and the original UML semantics is discussed. Problems such as analytical capabilities and different classes of activity diagrams are not addressed.

10 Conclusion

The formalization of UML2 activity diagrams is a challenge because of the subtleties of the informally formulated semantics. In this paper, we begin with an initial semantic mapping, which we subsequently analyze for its shortcomings. This leads us to a systematic classification and analysis of the semantic challenges of activity diagrams. Based on their semantic expressivity, we propose a classification of activity diagrams as a basis for future study. As a formalism of choice, we are using the π-calculus. The most restricted class of activity diagrams in our classification can be semantically mapped to the π-calculus by mapping each action to a π-process and each pin to an action prefix receiving or sending a message. Less restricted classes require to map pins and, in general, object nodes to π-processes themselves in order to correctly encode their token storing behavior. The least restricted classes require to add a global communication scheme by adding additional π-processes to the semantic encoding, for example to adequately capture the semantics of the activity final node.

We believe that a classification of activity diagrams is of high interest for theoretical and practical purposes. On the theoretical side, it is very beneficial to achieve a complete characterization of the maximal possible subclasses and to further study their theoretical properties such as the size of the resulting transition system. On the practical side, it is very interesting to add profound analytical capabilities to a modeling tool, which are optimized to analyze specific classes of diagrams. As activity diagrams gain increasing interest in particular in the area of business process modeling, being able to help a designer to analyze and verify models has significant value. Finally, such a classification can also provide valuable

input for future revisions of the UML2 specification.

Acknowledgement

We thank Jan Hendrik Hausmann, Jeff Magee, and Jussi Vanhatalo for their valuable comments on drafts of this paper.

References

[1] J. Amsden et al. Business process definition metamodel. Revised Submission to the OMG, BEI/RFP bei/2003-01-06, 2004.

[2] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulation and Advanced Topics*. Mc Graw-Hill, 1998.

[3] C. Bock. UML 2 activity and action models. *Journal of Object Technology*, 2(4):43–53, 2003.

[4] C. Bock. UML 2 activity and action models part 2: Actions. *Journal of Object Technology*, 2(5):41–56, 2003.

[5] C. Bock. UML 2 activity and action models part 3: Control nodes. *Journal of Object Technology*, 2(6):7–23, 2003.

[6] C. Bock. UML 2 activity and action models part 4: Object nodes. *Journal of Object Technology*, 3(1):27–41, 2004.

[7] E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In *8th International Conference on Algebraic Methodology and Software Technology*, volume 1816 of *LNCS*, pages 293–308. Springer, 2000.

[8] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In *Computer-Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 394–397. Springer, 1996.

[9] Y. Dong and Z. ShenSheng. Using π-calculus to formalize UML activity diagram for business process modeling. In *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 47–54. IEEE, 2003.

[10] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *Proceedings of the 3rd International Conference on the Unified Modeling Language*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.

[11] R. Eshuis and R. Wieringa. Tool support for verifying UML activity diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447, 2004.

[12] J. H. Hausmann. *Dynamic Meta Modeling*. PhD thesis, University of Paderborn, October 2005.

[13] R. Lucchi and M. Mazzara. A Pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming*, 2006. in press.

[14] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[15] R. Milner. *Communicating and Mobile Systems: The Pi-calculus*. Cambridge University Press, 1999.

[16] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.

[17] Object Management Group (OMG). *Unified Modeling Language: Superstructure*, 2005. Version 2.0 formal/05-07-04.

[18] H. Overdick, F. Puhlmann, and Mathias Weske. Towards a formal model for agile service discovery and integration. In *Proceedings of the ICSOC Workshop on Dynamic Web Processes at the 3rd International Conference on Service-Oriented Computing*, pages 25–37. IBM TechReport RC 23822, 2005.

[19] F. Puhlmann. Why do we actually need the Pi-Calculus for business process management? In *Proceedings of the 9th International Conference on Business Information Systems (BIS 2006)*, volume P-85 of *LNI*, pages 77–89. gi-ev.de, 2006.

[20] F. Puhlmann and M. Weske. Using the Pi-Calculus for formalizing workflow patterns. In *3rd International Conference on Business Process Management*, volume 3649 of *LNCS 3649*, pages 153–168. Springer, 2005.

[21] F. Puhlmann and M. Weske. Investigations on soundness regarding lazy activities. In *4th International Conference on Business Process Management*, volume 4102 of *LNCS*, pages 145–160. Springer, 2006.

[22] J. Pyke and R. Whitehead. Do better maths lead to better business processes? *BPTrends*, pages 1–7, February 2004.

[23] J. Roj and M. Owen. BPMN and business process management - introduction to the new business process modeling standard. www.bpmi.org, 2003.

[24] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[25] N. Russell, W. van der Aalst, A. ter Hofstede, and P. Wohed. On the suitability of UML 2.0 activity diagrams for business process modelling. In *Proceedings of the 3rd Asia-Pacicfic Conference on Conceptual modelling*, volume 53 of *CRPI*, pages 95–104. ACS, 2006.

[26] D. Sangiorgi and D. Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[27] H. Smith and P. Fingar. Business process fusion is inevitable. *BPTrends*, pages 1–15, March 2004.

[28] H. Smith and P. Fingar. Workflow is just a Pi process. *BPTrends*, pages 1–36, January 2004.

[29] H. Störrle. Semantics of control flow in UML 2.0 activities. In *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing*, pages 235–242. Springer, 2004.

[30] H. Störrle. Structured nodes in UML 2.0 activities. *Nordic Journal of Computing*, 11(3):279–203, 2004.

[31] H. Störrle. Semantics and verification of data flow in UML 2.0 activities. *ENTCS*, 127(4):35–52, 2005.

[32] H. Störrle. Towards a Petri-net semantics of data flow in UML 2.0 activities. Technical Report 04, University of Munich, 2005.

[33] H. Störrle and J. Hausmann. Towards a formal semantics of UML 2.0 activities. In *Proceedings German Software Engineering Conference*, volume P-64 of *LNI*, pages 117–128. gi-ev.org, 2005.

[34] P. Swithinbank et al. Build a business process solution using Rational and WebSphere tools. Redbooks SG24-6636-00, IBM, 2006.

[35] The UML2.0 semantics project. http://www.cs.queensu.ca/ stl/internal/uml2/.

[36] W. van der Aalst. Pi calculus versus Petri nets: Let us eat humble pie rather than further inflate the Pi hype, 2004. Discussion paper http://tmitwww.tm.tue.nl/research/patterns/download/pi-hype.pdf.

[37] W. van der Aalst. Why workflow is NOT just a Pi process. *BPTrends*, 2, 2004.

[38] B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In *Proceedings of the 6th International Conference on Computer Aided Verification*, volume 818 of *LNCS*, pages 428–440. Springer, 1994.

[39] V. Vitolins and A. Kalnins. Semantics of UML 2.0 activity diagram for business modeling by means of virtual machine. In *Ninth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005)*, pages 181–194. IEEE Computer Society, 2005.

[40] M. von der Beeck. A comparison of Statecharts variants. In *Proceedings of the 3rd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128–148. Springer, 1994.

[41] P. Wohed, W. van der Aalst, M. Dumas, A. ter Hofstede, and N. Russell. Pattern-based analysis of the control-flow perspective of UML activity diagrams. In *Proceedings of 24th International Conference on Conceptual Modelling (ER 2005)*, volume 3716 of *LNCS*, pages 63–78. Springer, 2005.

[42] M. Wynn, W. van der Aalst, A. ter Hofstede, and D. Edmond. Verifying workflows with Cancellation Regions and OR-Joins: An approach based on Reset Nets and reachability analysis. In *Proceedings of the 4th International Conference on Business Process Management*, volume 4102 of *LNCS*, pages 389–394. Springer, 2006. Long Version as bpmcenter.org Report 06-16.

[43] K. Xu, Y. Liu, and G. Pu. Formalization, verification and restructuring of BPEL models with Pi calculus and model checking. Technical Report RC23962(C0605-012), IBM, 2006.