Towards Axiomatizing the Semantics of UML Activity Diagrams: A Situation-Calculus Perspective

Xing Tan
Semantic Technologies Laboratory
University of Toronto
Email: xtan@mie.utoronto.ca

Michael Gruninger
Semantic Technologies Laboratory
University of Toronto
Email: gruninger@mie.utoronto.ca

***Abstract*—In this paper, the UML activity diagrams are first defined graph-theoretically, with an adoption of the concepts of Petri nets tokens. The semantics of activity diagrams is further axiomatized as a logical action theory called SCAD. Example applications of SCAD are also given.**

***Keywords*-UML Activity Diagrams; Situation Calculus Ontology**

I. Introduction

The Unified Modeling Language (UML) activity diagrams are designed for graphical specifications of dynamical aspects of systems. They have been widely used to model work flows of, e.g., business processes and software systems. For each graphical notation of activity diagrams, only a textual description is provided by the Object Management Group (OMG) to define its syntax and semantics [3].

In this paper, we start by providing a graph-theoretic definition for the activity diagrams and formal characterizations of activity diagram dynamics through adopting the concepts of Petri nets tokens. We move on to propose an ontology of activity diagrams called SCAD, standing for **S**ituation-**C**alculus action theory for **A**ctivity **D**iagram; whereas situation calculus (Reiter's variety, see [2]) is a second-order logic language that provides a rigorous paradigm for axiomatizations of dynamical systems.

SCAD contains a set of actions, corresponding to the firing of diagram nodes, and a set of function fluents, corresponding to the number of tokens at diagrams nodes, which are subject to change upon firing actions. The paper also covers two example SCAD applications: one, important mathematical properties of activity diagrams are further axiomatized in SCAD. Consequently, we show that the reachability problem is PSPACE-complete in a subclass of SCAD; two, an example diagram is presented, where the projection problem is investigated in particular.

II. UML Activity Diagrams

In this section, graph-theoretic definitions to describe UML activity diagrams are introduced first, and the concept of markings to capture the dynamics of activity diagrams are presented next.

Definition 1: An **UML activity diagram** is a pair (N, E), where N is a finite set and E is a binary relation on N. The elements in N are called **nodes**. Each node belongs to one and only one of the following types: Ini, $Final$, $Branch$, $Merge$, $Fork$, $Join$, or $Action$. The elements in E are called **edges**. The edge set E consists of ordered pairs of nodes. That is, an edge is a set $\{u, v\}$, where $u, v \in N$ and $u \neq v$. By convention, we use the notation (u, v) for an edge, rather than the set notation $\{u, v\}$.

If (u, v) is an edge in an activity diagram, we say that (u, v) is incident from or **leaves** node u and is incident to or **enters** node v. Given a node $u \in N$, the set ${}^\bullet u = \{v|(v, u) \in E\}$ is the **pre-set** of u, where each v is the input of u, and the set $u^\bullet = \{v|(v, u) \in E\}$ is the **post-set** of u, where each v is the output of u. It is required that [1]

$$|{}^\bullet u| \begin{cases} = 0 & \text{if } n \text{ is the } Ini \text{ node} \\ = 1 & \text{if } n \text{ is } Branch, Fork, Action, \text{ or } Final \\ = 2 & \text{if } n \text{ is a } Merge, \text{ or a } Join \text{ node} \end{cases}$$

and

$$|u^\bullet| \begin{cases} = 0 & \text{if } n \text{ is the } Final \text{ node} \\ = 1 & \text{if } n \text{ is } Merge, Join, Action \text{ or } Ini \\ = 2 & \text{if } n \text{ is a } Branch \text{ , or a } Fork \text{ node} \end{cases}$$

The concept of tokens and its firing is adopted from Petri nets. In a Petri-net, nodes of *places* contain tokens whereas firing of a transition node make changes to the number of tokens in the places that enter, or leave the transition node. As defined below, a node in an activity diagram by itself maintains tokens and can fire as long as it contains at least one token. In addition, the left (or right) input of a $Join$ node accepts left (or right) tokens and, intuitively, one left token and one right token will be counted as a full token for the $Join$ node.

Definition 2: A **marking** of a diagram (N, E) is a mapping in the form $MK : N \rightarrow \mathcal{N}$, to indicate the distribution of tokens on the nodes of the diagram; it can be represented as an vector $MK(n_1), \ldots, MK(n_m)$ where $n_1, \ldots, n_m$ is

[1] In general, the in-degrees of $Merge$ and $Join$, and the out-degrees of $Branch$ and $Fork$, all can be integers greater than 2.

an enumeration of the node set N and for all i such that $1 \leq i \leq m$, $MK(n_i)$ tokens are assigned to node n_i.
A node n is **marked** at the marking MK if $MK(n) > 0$. A marked node u is also **enabled** and is accepted by every node $v \in u^\bullet$. The firing of an enabled node u at MK leads to the successor marking MK' (Written as $MK \stackrel{u}{\Longrightarrow} MK'$). More precisely,

1) if u is a $Branch$ node, then for every node $n \in N$,

$$MK'(n) = \begin{cases} MK(n) - 1 & \text{if } n = u \\ \{MK(n) + 1, MK(n)\} & \text{if } n \text{ accepts } u \\ MK(n) & \text{otherwise} \end{cases}$$

and we also have, $\sum_{n_i \in u^\bullet} MK(n_i) = 1$;

2) if u is a non-$Branch$ node, then for every node $n \in N$,

$$MK'(n) = \begin{cases} MK(n) - 1 & \text{if } n = u \\ MK(n) + 1 & \text{if } n \text{ accepts } u \\ MK(n) & \text{otherwise} \end{cases}$$

In other words, after the firing of u, a token is removed from u and a token is added to the only node (if u is of type Ini, $Action$, $Merge$, $Join$), each node (if u is of type $Fork$), one and only one node (if u is of type $Branch$), in the post-set of u. There is no need to fire a node with type $Final$;

3) (Exception of $Join$) if a token fired by u is accepted by the left (right) in-edge of a $Join$ node n, then $MK_{left}(n)$ ($MK_{right}(n)$) is increased by 1. $MK_{left}(n) = 1$ and $MK_{right}(n) = 1$ function as one full token at n, i.e., $MK(n) = 1$.

The **firing sequence** $\sigma = n_1, ..., n_m$ is a sequence of nodes in N. For particular σ and MK, σ is **legal** at MK if there are marking sequence $MK_0, MK_1, ..., MK_m$ such that $MK = MK_0$, $M_0 \stackrel{n_1}{\Longrightarrow} MK_1$, ..., $MK_{m-1} \stackrel{n_m}{\Longrightarrow} MK_m$ (written as $MK \stackrel{\sigma}{\Longrightarrow} MK_m$).

The **reachability problem** for an (N, E, MK_0) is to decide, for some marking MK', if there exists a firing sequence σ such that $MK_0 \stackrel{\sigma}{\Longrightarrow} MK'$. An instance (N, K, MK_0) is **k-bounded** if the number of tokens of any node $n \in N$ at any MK in the reachability set is bounded by k.

III. Situation Calculus

The situation calculus is a logical language for representing changes upon actions in a dynamical domain. The language $\mathcal{L}$ of situation calculus as stated by [2] is a second-order many-sorted language with equality.

Three disjoint sorts: *action*, *situation*, *object* (for everything else in the specified domain) are included in the language $\mathcal{L}$. For example, *rain* denotes the act of raining, and $putdown(x, y)$ denotes the act of object x putdown y on the ground. A *situation* characterizes a sequence of actions in the domain. The constant situation S_0 is to denote the empty sequence of actions, whereas the function symbol *do* is introduced to construct the term $do(a, s)$, denoting the successor situation after performing action a (such as, in a weather simulation scenario, *rain*) in situation s. The situation term $do(sunshine, do(rain, s))$ denotes the situation resulting from first $rain$ and then $sunshine$, which distinguishes itself from the other situation term $do(rain, do(sunshine, s))$. It is easy to see that, intuitively, a situation corresponds to a finite sequence of actions.

The binary predicate $\sqsubset$ specifies the order between situations. For example, $s \sqsubset s'$ stands for that the situation s' can be reached by performing one or several actions from s. $s \sqsubseteq s'$ is an abbreviation of $s \sqsubset s' \vee s = s'$. In addition, a predicate $Poss(a, s)$ is applied to specify the legality of performing action a in situation s, For example, $Poss(rain, s) \supset heavyCloudy$ says that it is possible to rain only if the sky is with heavy cloud.

In a particular domain, the language might contain situation independent relations, like $matchLocation(Toronto)$, and situation independent functions, like $size(Plot2)$. However, in many of the more interesting cases, the values of relations and functions change between situations; accordingly, a relational fluent, or a functional fluent, in $\mathcal{L}$ is defined as a predicate, or a function, respectively, whose last argument is always a situation (e.g., $captain(John, do(catchFever, S_0))$ is a relational fluent, whereas $weight(John, do(recover, s))$ is a functional fluent).

IV. SCAD

The ontology of SCAD is formally defined in this section. Aside from *situations* and *actions*, objects in SCAD include diagram nodes: Ini, $Final$, $Action$, $Branch$, $Merge$, $Fork$, and $Join$. Functions of actions in SCAD include

- $fireJ(j)$: firing a $Join$ node;
- $fireB_l(b)$: firing a $Branch$ node to its left edge;
- $fireB_r(b)$: firing a $Branch$ node to its right edge;
- $fire(p)$: firing a node other than $Join$ and $Branch$.

Functional fluents includes:

- $TknsJ_l(j, s)$: the number of left tokens at a $Join$ node j in situation s;
- $TknsJ_r(j, s)$: the number of right tokens at a $Join$ node j in s;
- $Tkns(p, s)$: the number of tokens at a non-Join node p in s.

Situation-independent relations are defined in $\mathcal{S}_{S_0}$ of SCAD, which specifies the structure of an activity diagram and include:

- $LpreL(m, n)$: the left output of a ($Fork$ or $Branch$) node m enters the left input of a $Merge$ or $Join$ node n, $LpreR(m, n)$, $RpreL(m, n)$, and $RpreR(m, n)$ are defined in a similar way;
- $Lpre(m, n)$: the left output of a ($Fork$ or $Branch$) node m enters the only input of node n that is not of type $Merge$ or $Join$, $Rpre(m, n)$ is defined similarly;

- $preL(m,n)$: the only output of a (non-Fork and non-Branch) node m enters the left input of of a type $Merge$ or $Join$ node n, $preR(m,n)$ is defined similarly;
- $pre(m,n)$: all cases including the other cases (i.e., the only input of the node m enters the only input of the node n), and all of the above cases;
- $post(m,n)$: m leaves n if and only if n enters m.

Definition 3: SCAD is defined as a logical theory $\mathcal{S}_{scad}$, which consists several sets of axioms as follows:

$$\mathcal{S}_{scad} = \mathcal{D}_f \cup \mathcal{S}_{ap} \cup \mathcal{S}_{ss} \cup \mathcal{S}_{una} \cup \mathcal{S}_{S_0}$$

where

- $\mathcal{D}_f$ is the foundational axioms (see [2] for details);
- $\mathcal{S}_{ap}$ (action preconditons axioms)
  - $Poss(fireJ(j), s) \equiv$ $TknsJ_l(j,s) \geq 1 \wedge TknsJ_r(j,s) \geq 1$ (a $Join$ node j is enabled to fire iff the number of left tokens and the number of right tokens of it, both are greater than, or equal to 1);
  - $Poss(fireB_l(b), s) \equiv Tkns(b,s) \geq 1$ (a $Branch$ node b is enabled to fire to its left iff the number of tokens it contains is greater than or equals to 1);
  - $Poss(fireB_r(b), s) \equiv Tkns(b,s) \geq 1$ (a $Branch$ node b is enabled to fire to its right iff the number of tokens it contains is greater than or equals to 1);
  - $Poss(fire(p), s) \equiv Tkns(p,s) \geq 1$ (a node p is enabled to fire iff the number of tokens it contains is greater than or equals to 1);
- $\mathcal{S}_{ss}$ (successor state axioms)
  - $Tkns(p, do(a,s)) = n \equiv \gamma_f(p,t,n,a,s)$ $\vee (Tkns(p,s) = n \wedge \neg\exists n'\ \gamma_f(p,t,n',a,s))$ where $\gamma_f(p,t,n,a,s) \stackrel{def}{=} \gamma_{f_a}(p,n,a,s) \vee \gamma_{f_b}(p,q,n,a,s)$, referring to the two sets of firing actions that makes the number of tokens at the non-$Join$ node p on situation $do(a,s)$ to n, that is:
    (1). $\gamma_{f_a}(p,n,a,s) \stackrel{def}{=} (n = Tkns(p,s) - 1 \wedge$ $(a = fire(p) \vee a = fireB_l(p) \vee a = fireB_r(p)))$ (at situation s, the number of tokens at the node p, which is of any type but $Join$, is $(n+1)$, and p fires at situation s, making the number of tokens it contains at the subsequent situation $do(a,s)$ decreased by 1);
    (2). $\gamma_{f_b}(p,q,n,a,s) \stackrel{def}{=}$ $((\exists q).\, pre(q,p) \wedge n = Tkns(p,s) - 1 \wedge (a = fire(q) \vee$ $a = fireB_l(q) \vee a = fireB_r(q) \vee a = fireJ(q)))$ (at situation s, the number of tokens at place p, which is of any type but $Join$, is $(n-1)$, and the node q, which is of any type and enters p, fires at s, making the number of tokens at $do(a,s)$ also to n);
  - $TknsJ_l(p, do(a,s)) = n \equiv \gamma_f(p,t,n,a,s) \vee$ $(TknsJ_l(p,s) = n \wedge \neg\exists n'\ \gamma_f(p,t,n',a,s))$ where $\gamma_f(p,t,n,a,s) \stackrel{def}{=} \gamma_{f_a}(p,n,a,s) \vee \gamma_{f_b}(p,q,n,a,s)$, referring to the two sets of firing actions that makes the number of left tokens at the $Join$ node p on situation $do(a,s)$ to n, that is:
    (1). $\gamma_{f_a}(p,n,a,s) \stackrel{def}{=}$ $(n = TknsJ_l(p,s) - 1 \wedge a = fireJ(p))$ (at situation s, the number of left tokens at the node p, which is of type $Join$, is $(n+1)$, and p fires at situation s, making the number of tokens it contains at the subsequent situation $do(a,s)$ decreased by 1);,
    (2). $\gamma_{f_b}(p,q,n,a,s) \stackrel{def}{=} ((\exists q).\, preL(q,p) \wedge$ $n = Tkns(p,s) + 1 \wedge (a = fire(q) \vee a = fireB_l(q)$ $\vee a = fireB_r(q) \vee a = fireJ(q)))$ (at situation s, the number of tokens at place p, which is of type $Join$, is $(n-1)$, and the node q, which is of any type and enters the left edges of p, fires at s, making the number of left tokens at $do(a,s)$ also to n),
  - The argument for the right tokens of the $Join$ node, $p\ TknsJ_r(p, do(a,s)) = n$, is similar to the argument of the left tokens, specified as above.
- $\mathcal{S}_{una}$ (unique name axioms.)
- $\mathcal{S}_{S_0}$ (initial situation axioms.)

V. Applications

In this section, a few subclasses of SCAD are first defined and we show that the reachability problems in a particular subclass is PSPACE-complete. An abbreviation as follows is used for the specification [2]:

$$exec(s) \stackrel{def}{=} (\forall a, s).do(a,s) \sqsubseteq s \supset Poss(a,s).$$

Next, a SCAD instance (Figure 1) is introduced. Example use of SCAD is demonstrated by several projection problems (i.e., the problems of deciding whether a formula is true in the situation resulting from performing a sequence of ground actions).

Two dynamical properties of activity diagrams defined in Section II are formally characterized in SCAD as follows.

- **Reachability** (Given specified markings M_n, n_i is the specified number of tokens at the nodes p_i in M_n)

$$Q_{reach}(s,p) \stackrel{def}{=} \exists s\, exec(s) \wedge (S_0 \sqsubset s) \wedge \Big(\bigcup Tkns(p_i,s) = n_i\Big)$$

- **K-boundedness**

$$Q_{kbound}(s,p) \stackrel{def}{=} exec(s) \supset Tkns(p,s) \leq k$$

[2]First introduced as Equation 4.5 in [2].

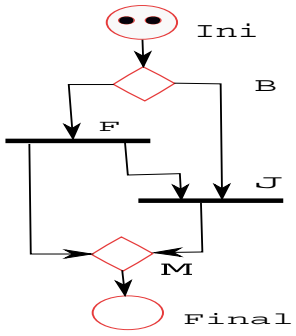

Figure 1. An activity diagram

In [4], It is shown that the reachability problem in a general K-bounded activity diagram is PSPACE-complete, that is, given a SCAD action theory $\mathcal{S}_{scad}$, deciding

$$\mathcal{S}_{scad} \cup Q_{kbound}(s,p) \models Q_{reach}(s,p)$$

is PSPACE-complete; whereas the problem is NP-Complete if one further constraint (namely reversibility) is added.

Here we show in addition that

Theorem 1: Given an $\mathcal{S}^{'}_{scad}$ where its nodes are free of $Branch$ and $Merge$ types, deciding

$$\mathcal{S}^{'}_{scad} \cup Q_{kbound}(s,p) \models Q_{reach}(s,p)$$

is PSPACE-complete.

Proof: (Sketch.) The same nondeterministic algorithm for the general case appeared in [4] can still be applied here to show that the problem is in PSPACE.

To show that it is PSPACE-hard, we transform the PSPACE-complete Deterministic Linear Space Acceptance (DLSA) problem (see [1]) into the current problem. The proof is similar to the PSPACE-hard proof for Theorem 1 in [4], where a PSPACE-complete nondeterministic Linear Space Acceptance (NLSA) problem is applied. Here the transition function is easier as no $Branch$ node is needed. In case a node belongs to multiple components, instead of using $Merge$, $Branch$ nodes and poly-bounded duplicates of the nodes are introduced, and the proof then can be carried out in essentially the same way as the one for Theorem 1 of [4]. ■

Example 1: A SCAD instance $\mathcal{S}_1$ is defined such that $\mathcal{S}_1 = \mathcal{D}_f \cup \mathcal{S}_{ap} \cup \mathcal{S}_{ss} \cup \mathcal{S}_{una} \cup \mathcal{S}_{S_0}$ where

$$\mathcal{S}_{S_0} = \{pre(Ini,B), Lpre(B,F), RpreR(B,J), RpreL(F,J),$$
$$LpreL(F,M), preR(j,M), pre(M,Final), Tkns(Ini,S_0) = 2,$$
$$Tkns(B,S_0) = 0, Tkns(F,S_0) = 0, TknsJ_l(J,S_0) = 0,$$
$$TknsJ_r(J,S_0) = 0, Tkns(J,S_0) = 0,$$
$$Tkns(M,S_0) = 0, Tkns(Final,S_0) = 0\}$$

Figure 1 is its pictorial presentation. Now a reachability problem can be stated as a SCAD entailment problem:

$$\mathcal{S}_1 \models \exists s.\, exec(s) \wedge (S_0 \sqsubset s) \wedge Tkns(Final,s) \geq 1?$$

That is, is there a sequence of executable actions such that at least one token is delivered to the $Final$ node? Define

$$\overrightarrow{a} = \{fire(Ini), fireB_l(B), fire(F), Fire(M)\},$$

it can be seen that, let $s_a = do(\overrightarrow{a}, S_0)$ [3],

$$\mathcal{S}_1 \models exec(s_a) \wedge (S_0 \sqsubset s_a) \wedge Tkns(Final,s_a) = 1.$$

From the fourth foundational axioms, it is obvious that $S_0 \sqsubset s_a$. The executability of $\overrightarrow{a}$ and $Tkns(Final,s_a) = 1$ can be verified by sequentially applying the four precondition axioms in $\mathcal{S}_{ap}$ and the three successor state axioms in $\mathcal{S}_{ss}$. For example, $Poss(fire(Ini),S_0)$, together with $exec(S_0)$, leads to $exec(do(fire(Ini),S_0))$; whereas $Tkns(Ini,do(fire(Ini),S_0)) = 1$ and $Tkns(B,do(fire(Ini),S_0)) = 1$.

Now, let

$$\overrightarrow{b} = \{fire(Ini), fire(Ini), fireB_r(B),$$
$$fireB_l(B), fire(F), FireJ(J), Fire(M), Fire(M)\}$$

It is obvious that $Tkns(Final,s_b) = 2$.

The K-boundedness of $\mathcal{S}_1$, meanwhile, can be stated as a SCAD entailment problem in the form:

$$\mathcal{S}_1 \models exec(s) \supset Tkns(p,s) \leq 2?$$

The proof is based on the observation that all executable sequences are with finite length.

VI. Summary

In this paper, a graph-theoretic specification for the structural properties of activity diagrams, with adoptions of the concept of tokens in Petri-nets to model the dynamics these diagrams, is proposed. A second-order axiomatization of this semantics of UML activity diagrams called SCAD is further provided. The design of SCAD makes use of the strong correspondence between the situation s in situation calculus, which is a sequence of actions starting from the initial state S_0, and the marking M in activity diagrams, which is resulted from a sequence of node firings starting from the firing of the initial node at initial marking.

References

[1] M. Garey and D. Johnson, Computers and intractability - a guide to NP-completeness. W.H. Freeman and Company, Cambridge, MA, USA (1979).

[2] R. Reiter, Knowledge in Actioin: Logical foundations for describing and implementing dynamical systems, The MIT Press, 2001.

[3] OMG: Unified Modeling Language (OMG UML), Superstructure, V2.1.2.,(2007).

[4] X. Tan and M. Gruninger, On the computational complexity of the reachability problem in UML activity diagrams, *In proceedings of the IEEE International Conference on Intelligent Computing and Intelligent Systems*, Shanghai, China (2009), Vol. 2, 572–576.

[3] an abbreviation for $do(Fire(M), do(fire(F), do(fireB_l(B), do(fire(Ini), S_0))))$.