Formalising UML Activity Diagrams using Finite State Processes

Roberto W. S. Rodrigues [1]

{rwdr@doc.ic.ac.uk}
Imperial College, 180, Queen's Gate SW7 2BZ, London UK.

Abstract. Among the set of diagrams of UML used to express dynamic aspects of systems, the Activity Diagram (AD) is the only one that deals with Business Processes (BP) and workflows. However, the lack of a well-defined semantics leaves the notation open to many interpretations. In this paper we provide a simple semantics by formalising the UML Activity Diagram using finite state processes (FSP). A given UML AD specification can be analysed by checking its equivalent FSP description using the LTSA model-checker. In addition, LTSA can animate the workflow behaviour from the BP models expressed in UML AD.

1 Introduction

The UML Activity Diagram (AD) is the only OMG standard notation for modelling Business Processes (BPs) and workflows [1]. Unlike State Diagrams, AD is intended to describe a sequence of actions that may belong to several objects. The diagram can then be used to model workflows by describing the overall dependencies between objects across several domains underlying the BP being specified.

Recent works have addressed the integration of AD with other UML notations. For instance, it is suggested that it can be used to describe Use Cases and Class operations [2], and as an intermediary step between Use Cases and Interaction diagrams [3]. However, little attention has been given to its use and interpretation in the context of workflow systems, and the notation has been blamed for not being expressive enough to cover complex aspects of Business Processes.

Just as the other UML diagrams, the AD was given a metamodel in the UML version 1.3 [4] that defines the relations between its syntax elements. It was one of the latest notations to be included in the UML set [5], but the given metamodel is still incomplete since it provides just an informal semantics in English prose. The use of OCL (Object Constraint Language) [6] helps to define constraints between AD model elements. However, despite of being a powerful language, OCL is too verbose for

[1] This work is sponsored by CAPES and CEFET-CE – Ministry of Education - Brazil

expressing behavioural modelling [7] and does not express control flow aspects necessary to model BP properties.

This paper proposes a formal semantics for UML Activity Diagram using Finite State Processes (FSP) [8]. Using FSP, BP models described with UML AD can be modelled as Labelled Transition Systems (LTS), which represent the computational effect of the specification.

2 UML Activity Diagram Syntax

We define the abstract syntax of the UML AD using a syntax tree as showed in figure 1. It defines the relation between model elements of UML AD as defined in its metamodel. We add a set of constraints numbered from C1 to C13 for the nodes (N) in order to define how to traverse the tree. For example, the constraint C2: N2 = N5 ∧ N6 states that every time the non-terminal node 2 is reached, the terminal nodes N5 and N6 must be taken. This derives respectively the *Begin* and *ControlFlow* symbols.

To generate a trivial UML AD with a *Begin*, one *ActionState* and *End* we can use the following paths: <N1, N2>$\rightarrow^{C2}$<N5, N6>; <N1, N3, N17>$\rightarrow^{C7}$<N18, N6> and <N1, N4>$\rightarrow^{C4}$<N7>. They generate the sequence of symbols <*Begin, ControlFlow*>, <*ActionState, ControlFLow*> and *End* with the use of the constraints C2, C7 and C4 respectively. For complex ADs, several steps must be followed, and they include traversing the tree several times, as indicated by a star symbol in the constraints. In this case, the search for a symbol is made in a mix of depth and breadth search as needed.

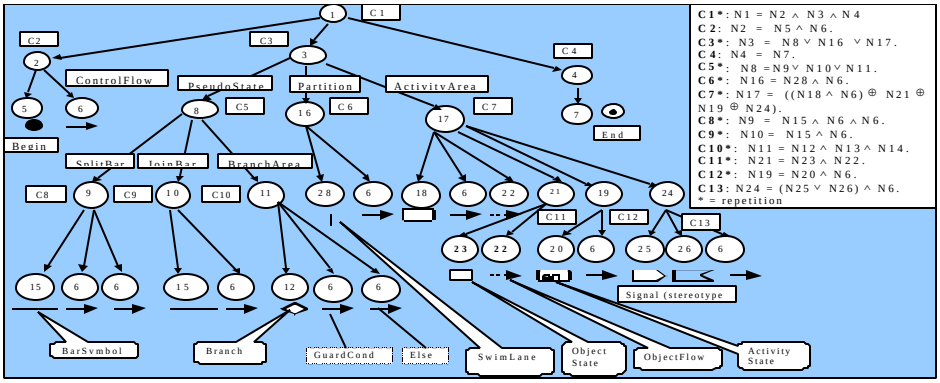

Figure 1: UML Activity Diagram Syntax Tree

3 Finite State Processes - FSP

FSP is a formalism that can be used for modelling systems as a Labelled Transition System [8]. It has several constructs to model process behaviour. In this section we describe the most relevant ones as follows.

- **Action Prefix:** A description a→P means a process that engages in an action *a*, and after *a* completes, the process behaves as described in P. *a* is the action prefix.
- **Parallel Composition:** Two or more processes are in parallel, as expressed by the operator '||' e.g. (P || Q ||…) when they have their actions interleaved during their execution.
- **Choice:** Given two actions *a* and *b* in a process (a→P | b →Q), it can engage in either a or *b* and behaves respectively either as P or Q. The symbol '|' means choice and can also indicate non-determinism if the actions are the same, e.g. (x→P | x→Q). Conditions can be associated to the choice operator using the **when** constructor.
- **Shared Actions:** Two processes synchronise when they share at least one action. For example, given P = (a->c->P) and Q = (c->d-Q), P and Q meet at some point in time when the action *c* is executed.
- **Process Labelling:** When one process has several instances, their actions have exactly the same name. To allow interaction between instances of the same process, each action can receive a distinct label so as to make them distinct processes. For example, the expression (x:P || y:P) models two concurrent instances of the same process P.
- **Relabelling:** Given two or more processes whose actions have different names, if we want them to synchronise we re-label their actions by changing their names. The relabelling operator in FSP is the slash symbol preceded by the action name to be relabelled, and succeeded by its new name in a form (*action*/*newAction*).
- **Hiding:** We can remove some actions of the alphabet of a process by hiding them using the backslash operator. For example Q = (c->d->Q)\d conceals the action *d* from Q. From now on *d* cannot be shared. There is also an interface operator @ which hides a set of chosen actions from a process.

4 Mapping AD to FSP

The objective of mapping from UML AD to FSP is to derive a model of computation expressed in LTS so as to be able to check the behaviour expressed in the AD specification. We show this mapping line by line in table 1 where for each UML AD syntax on left we provide its equivalent FSP description on right described as follows.

Begin semantics: we use a trivial process named BEGIN which has a *start* action. However, any initial action can be interpreted as *start* action.

End: An *End* action is mapped using a transition (→) to a special STOP operation in FSP. It indicates that no further actions take place in a Business Process. The control must go back to the workflow engine after a STOP operation.

ControlFLow: it is equivalent to an ordered flow of control in workflows. A sequence of *ControlFlow* in UML is mapped into FSP as a sequence of transitions (→) that indicates threads of control.

ActionState: it is a primitive element meaning any action that does not engage in any sub-states.

ActivityState: it is a set of one or more *ActionState* that takes some time to complete. An *ActivityState* in UML can then be expressed in FSP as a process and its associated actions, e.g. P **= (a® b ® P)**. In this case, this representation also models repetition of actions with recursion mechanism. For instance the process **P** above calls itself and resumes its initial state.

UML AD	**Sequence - Syntax**	**FSP**
Begin	<N1,N2>→^{C2}<N5,N6>	BEGIN = (start→BEGIN)
End	<N1,N4>→^{C4} N7	→STOP
ControlFlow		→
ActionState	<N3,N17>→^{C7}<N18,N6>	→a
ActivityState	<N1,N3,N17,N21> →^{C12} <N20,N6>	P = (a→P).
StereoType	<N1,N3,N17,N24>→^{C13} <N25,N6>	P=(a→S), S=(c→P)
JoinBar	<N1,N3,N8,N10>→^{C9}<N15,N6>	
OR-JOIN	P =(a→JoinBar | b→ JoinBar | c → JoinBar), JoinBar = (d→P).	
XOR-JOIN	P = (when cond1 a→JoinBar | b→JoinBar | c→JoinBar), JoinBar = (d→P).	
AND-JOIN	P1 = (a→d→P1). P2 = (b→d→P2). P3 = (c→d→P3). ||P = (P1 || P2 || P3).	
SplitBar	<N1,N3,N8,N9>→^{C8}<N15,N6,N6>	
OR	P = (d→SplitBar), SplitBar = (a->P | b→P | c→P).	
AND	D = (d→D). P1 = (d→a)->A). P2 =(d→b). P3 = (d→c). ||P = (D ||P1 || P2 || P3).	
Branch	<N1,N3,N8,N11>→^{C10}<N12,N6,N6>	
XOR	P = (d → Branch), Branch = (when cond1= true a→P | when cond2 = true b->P | when cond3 = true c->P).	
Proxy	<N1,N3,16>^{C6}→<N28,N6>	Synchronised actions.

Table 1: Mapping from Activity Diagram to FSP

Stereotype: we use a local process to model a presence of a *Stereotype(signal)*. Local processes are separated from the main process by a comma. For example the process P=(a->S), S=(c->P) describes a local process S that is part of P. A dot ends a process declaration. We describe the remaining mapping by the use of examples as follows.

OR-JOIN: OR-JOIN is an operator that indicates synchronisation of input actions that may or may not happen, since for any input conditions only one is necessary and sufficient to be true. For instance, if any one of given three income actions *a*, *b*, and *c* become true then the action *d* is executed. The action name *JoinBar* is just to make the mapping clearer. It is used to synchronise the actions *a*, *b* and *c* (see table 1).

XOR-JOIN: the XOR-JOIN construct indicates that no synchronisation is required. However, only one condition (if any) must be true in order to decide for the next action in the workflow. We map it to FSP using *guards* with **when** constructor. For example, given the input actions *a*, *b* and *c*, the output action *d* is chosen by evaluating the condition *cond1* (see table 1).

AND-JOIN: the meaning of AND-JOIN in workflows in general is to specify that joint incoming concurrent threads or processes must synchronise. AND-JOIN is expressed in FSP using the parallel composition operator "||". In table 1 we show an example where three processes are given. Each one can engage in the actions *a*, *b* and *c* respectively. Any of the three actions can be executed in an interleaved order, as this is the semantics of parallelism in FSP.

OR-SPLIT: the semantics of OR-SPLIT produces an opposite effect in relation to OR-JOIN. For example, suppose *d* is the income action. The control flow will follow one or more paths whatever outcome actions come true.

AND-SPLIT: the AND-SPLIT as opposed to AND-JOIN is an operation that synchronises concurrent actions at specific point in time by splitting several streams of events after one activity has occurred. We map the AND-SPLIT operation as shared actions.

XOR-SPLIT: XOR-SPLIT is an operation that must lead to no more than one consequent transition if at least one precedent condition evaluates true. Thereupon the workflow engine follows the chosen transition in the path to other actions. We map XOR-SPLIT to FSP using guarded actions. The AD symbol equivalent to XOR-SPLIT is the *branch* symbol (diamond) but also can be associated to a Bar symbol.

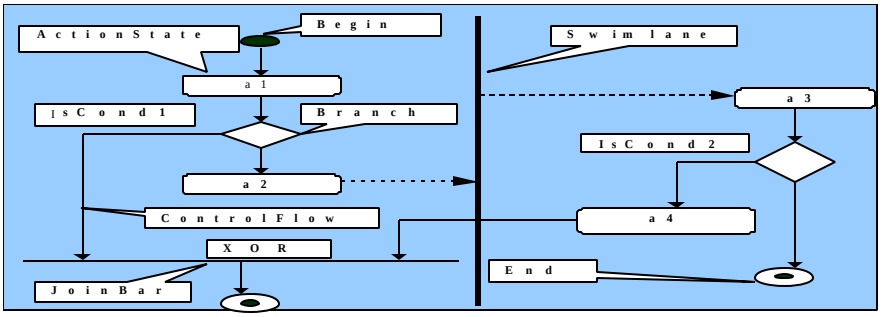

Figure 2: An example of UML Activity Diagram

A *SwimLane* is a mechanism in UML to separate activities in parts such as organisations, departments, components or even threads of execution. A *SwimLane* is mapped into FSP as a *handshake* by using shared or relabelled actions to synchronise activities of different processes. We introduce a particular abstraction called *proxy* activity to model exchanging of messages. In the following, we use the figure 2 above as input to illustrate an example of formalisation from UML AD to FSP including the *SwimLane* as follows.

We use two processes; Process_1 and Process_2 for the left and right side of the *SwimLane* respectively as showed in table 2. As FSP deals only with integers, we then

declare two variables TRUE =1 and FALSE=0. Consequently, the conditions *cond1* and *cond2* require a range from 0 to 1 as defined by T. The Proces_1 begins with a

Process_1	***SwimLane***	***Process_2***
Const TRUE =1 Const FALSE = 0 Range T = 0..1 Process_1 = (start -> WORKFLOW[TRUE]), WORKFLOW[cond1:T] = (a1->(when cond1 = = TRUE a2->proxyOUT->Process_1 | when cond1= = FALSE exit->STOP).	SwimLane= (input →connect →ouput →Swim-Lane).	Process_2= WORKFLOW2[TRUE], WORKFLOW2[cond2:T]= (proxyIN->a3->(when cond2 = = TRUE a4->SWIMLANE2 | when cond2 = =FALSE exit->STOP)), SWIMLANE2=(joinbar->STOP).
||EXAMPLE=(Process_1||SWIMLANE||Process_2)/{input/proxyOUT,output/proxyIN}.).		

Table 2: An Example of Formal Specification in FSP

start action. We assume the first condition *cond1* is true. After the start, the action *a1* is executed followed by *a2*. Since the *cond1* is TRUE, ProxyOUT action is executed next.

To model the *SwimLane* between the two processes we use shared action mechanism to synchronise both processes. The *input* action synchronises with Process_1, and *output* action with Process_2.

Note that the *SwimLane* only provides connection between processes. The action *connect* enforce this fact. However, we extend the *SwimLane* to be used in two ways. First we use dashed arrows for *SwimLane* that works as an external channel between processes. They can be configured to connect or disconnect several processes by just adding synchronisation actions. In the second way we use an one-way channel that set a straight dependencies between particular actions over which there is a need for a workflow engine to exert some control.

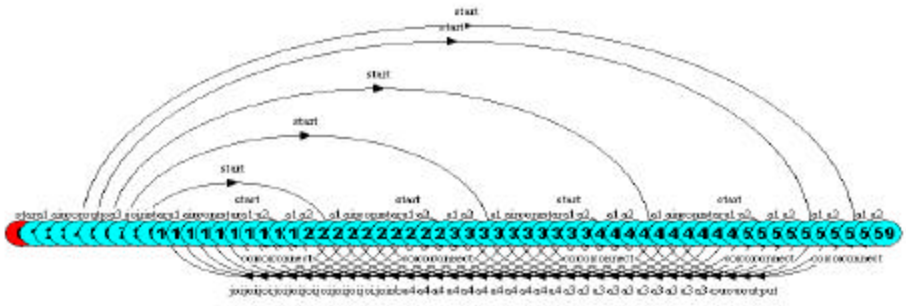

Figure 3: Animation of FSP using State Machines.

The Process_2 has behaviour similar to Process_1. The XOR-split (diamond) decides if the workflow engine exits (*exit action*) or proceeds to the action *a3*. As the guard condition is TRUE, the workflow engine engages in *a3*, then on *a4*. Eventually both processes can stop in case one of the conditions evaluates to FALSE.

The composition of Process_1, Proces_2 and the SwimLane is called EXAMPLE. It defines the overall behaviour expressed in the Activity Diagram by checking all combinations of actions, according to the conditions applied. The animation of the given UML AD is showed in figure 3. It was generated by LTSA model-checker [8].

5. Conclusion

This paper shows an early work in providing a precise semantics for UML Activity Diagrams. We propose Finite State Process with LTS formalism to be used as a common semantics. This is particularly useful in scenarios where BPs are implemented as a chain of networked workflows. In this case, several Workflow Designers can reuse AD specifications and are able to generate the same workflow behaviour for different workflow tools. In addition, as UML AD can be used at early stages to encourage domain experts to analyse different scenarios [9], LTSA can provide a compositional analysis of concurrent aspects of the Business Process under scrutiny in order to check safety and liveness properties. This and the map of ObjectState to FSP will be described in future work.

References

1. J. Rumbaugh G. Booch, and Ivar Jacobson, *The Unified Modeling Language User Guide*. Object Technologies, ed. Addison-Wesley. 1999: Addison-Wesley.
2. Perdita Stevens and Rob Pooley, *Using UML Software Engineering with Objects and Components*. Object Technology, ed. Addison-Wesley. 2000.
3. B. Paech. *On the Role of Activity Diagrams in UML - A User task Centered Development Process for UML*. in *<<UML'98>> :Beyond the Notation, First International Workshop*. 1998. Mulhouse, France: Springer.
4. OMG, *UML Unified Method: Notation Guide, Version 1.3*, . 1999, OMG.
5. Cris Kobryn, *UML 2001: A Standardization Odyssey*, in *Communications of the ACM*. 1999. p. 29-37.
6. Jos B. Warmer and Anneke G. Kleppe, *The Object Constraint Language - Precise Model with UML*, ed. B.J. Rumbaugh. 1999: Addison-Wesley.
7. Richard F. Paige and Jonathan S. Ostroff. *A Comparison of the Business Object Notation and The Unified Modeling Language*. in *<<UML'99>> - The Unified Modeling Language, Beyond the Standard*. 1999. Fort Collins, CO, USA: Springer.
8. J. Magee and J. Kramer, *Concurrency State Models & Java Programs*. 1999: Wiley.
9. Martin Fowler and Kendall Scott, *UML Distilled - Applying the Standard Object Modeling Language*. Addison-Wesley Object Technolgy Series, ed. B.J. Rumbaugh. 1997: Addison Wesley.