Modeling Transactions in UML Activity Diagrams via Nonsequential Automata

Júlio P. Machado
Pontifícia Universidade Católica do Rio Grande do Sul, Faculdade de Informática
Porto Alegre, RS, Brazil
juliopm@inf.pucrs.br

and

Paulo B. Menezes
Universidade Federal do Rio Grande do Sul, Instituto de Informática
Porto Alegre, RS, Brazil
blauth@inf.ufrgs.br

Abstract

When modeling concurrent or parallel systems, we must be aware that basic activities of each system may be constituted by smaller activities, i.e. transitions are conceptually refined into transactions. Nevertheless, the Unified Modeling Language (UML) seems to lack compositional constructs for defining atomic activities. We discuss nonsequential automata for the formal interpretation of the concept of composing transitions into transactions under UML activity diagrams. Transactions are formally defined through a special morphism between automata that maps transitions from the source automaton to transactions of the target (more concrete) automata. UML activity diagrams are then extended with a proper stereotype for defining transactions.

Keywords: Formal Specification, UML, Nonsequential Automata, Concurrent and Distributed Systems.

1 Introduction

The Unified Modeling Language (UML) [8, 3, 20, 10] is a language which may be used to describe both the structure and behavior of object-oriented systems using a combination of notations. UML offers a variety of graphical diagrams for specifying, visualizing and documenting object-oriented systems. These models can be classified as concerned with the static structure of systems and those concerned with the dynamic behavior. For the modeling of the dynamic behavior, a number of different models are offered: sequence, collaboration, statechart and activity diagrams.

UML has a precisely-defined syntax, however it still lacks a generally accepted formal semantics which precisely fixes the meaning of its diagrams. This is particularly noticeable for the notations which model the dynamic behavior of systems. Several approaches to translating UML diagrams into formal models have been based on Petri nets [17]. For example, [9] describes a formal translation of activity and collaboration diagrams into place/transition Petri nets and [7] compares different proposals for the semantics based on Petri nets. Also workflow models have been given semantics based on place/transition nets, e.g. [6] defines a formal semantics for UML activity diagrams that is suitable for workflow modeling. Widely application of Petri nets in theoretical and applied research to specify and visualize the behavior of systems are due to their intuitive graphical representation and several supporting tools.

Activity diagrams are one of the means for describing behavior of systems within the UML language and has been chosen here for its capability in describing flows of activities of a desired system. Broadly speaking, states of an activity diagram mostly represent the execution of sequential and concurrent steps in a computational process or a workflow step. Although activity diagrams have been used for modeling workflow processes [6, 7, 8, 3], such view is not going to be explored in this paper. Here we concentrate on describing groups of sequential or concurrent activities that are responsible for performing a computation, and we address the issue of modeling transactions, a feature not present in activity diagrams, by using a Petri net related model, named Nonsequential Automata ([11], [13], [15], [14]).

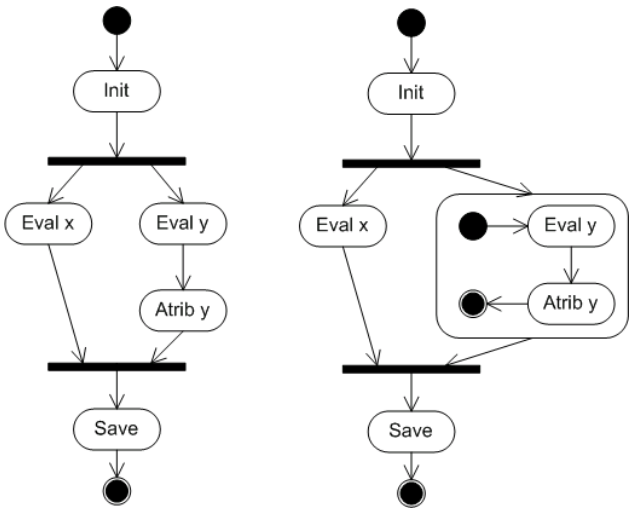

Figure 1: UML activity diagram without (left) and with (right) composite state

We remark that in our setting the term "transaction" denotes a certain activity of the system that might be composed by many, possibly concurrent, sub-activities. Moreover, we require this composition of activities to be considered atomic.

The importance of discussing composition mechanisms is based on the fact that, as pointed in [1], in order to master the complexity of systems, models demand a form of structural decomposition. When building large systems, one needs constructs to describe systems as compositions of other smaller systems. In this sense, when modeling a computational process, we need means of composing sub-activities both in a non atomic or atomic way.

All foundational work presented in this paper is developed within category theory [2], as it provides powerful techniques to unify different models through adjunctions expressing relations between their semantics [21],[2]. As the reader will see, adjunctions consist of ways of translating from one model to another.

The rest of the paper is organized as follows. Section 2 briefly recalls some basic definitions of UML activity diagrams and presents the need for the definition of atomic composition of activities (i.e. the definition of transactions). Section 3 presents the compositional semantic domain of nonsequential automata, which is going to be used as the semantics for the composition of activities in UML. Section 4 introduces translation schemes for building nonsequential automata from activity diagrams. Finally, Section 5 discusses the results and outlines possible directions for future investigations.

2 UML Activity Diagrams

Activity diagrams are one of the means for describing behavior of systems within UML. It defines an extended view of state machines focused on the flow of control from activity to activity, instead of statechart diagrams which is focused on potential states of objects and transitions among those states. Figure 1 depicts a simple example of an activity diagram for an operation.

For the most part, activity diagrams involve modeling sequential and concurrent steps in a computational process. In UML, those steps may be an action or a subactivity state. Action states represent an atomic computation without substructure and that cannot be interrupted. Subactivity states, on the other hand, represent non-atomic execution and can be decomposed and interrupted by external events (represented by transitions out of the state) - they can be think as a composite whose flow of control is made up of other activities and actions. A subactivity state is semantically equivalent to expanding its activity graph in place until there are only actions [10]. Nonetheless, they are important because they help breaking complex computations into parts. There is no difference in notation for action and subactivity states, besides the nested activity graph. Other important states are initial and final states.

Transitions between states are by means of arcs connecting the source and destination states. Transitions may be adorned with an event trigger, a guard condition and an action. The event trigger is the event whose reception by the source state makes the transition enabled. A transition without an explicit event is called a completion transition, indicating that it is enabled once the activity/action in the source state has been completed. A guard condition is a boolean expression that is evaluated when a transition is triggered by an event, causing the transition to fire if its value is true. Optionally, an action may be executed in response to the transition. To aid the visualization, branches are introduced to specify alternate paths based on boolean

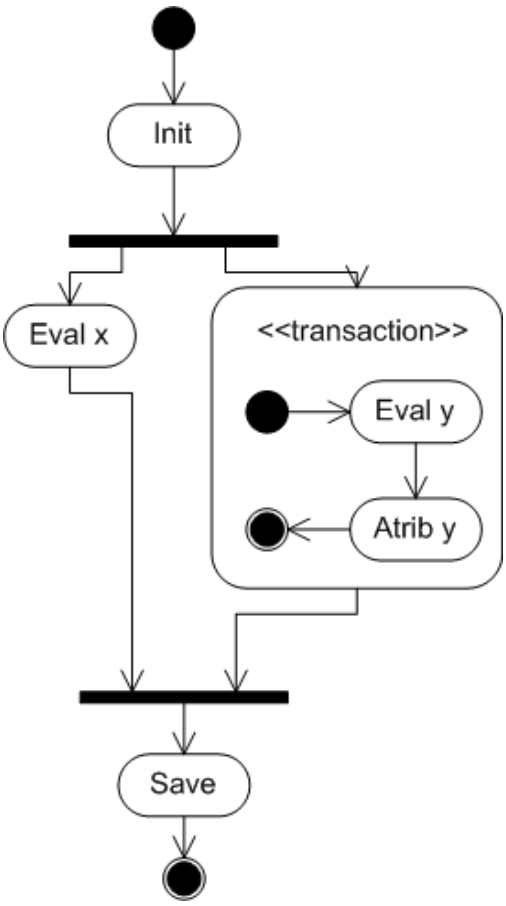

Figure 2: UML activity diagram with transaction composite state

expressions. A transition leaving an action state can be labeled with guard conditions and action expressions, but not with events. Normally, as stated in [10], an activity diagram will not present transitions with events, as activity diagrams assume that computations proceed without external event-based interruptions [1]. In order to capture complex transitions, horizontal bars represent the fork and join of concurrent steps of computation. Bellow the fork, the activities associated with each path continue in parallel. A join represents the synchronization of two or more concurrent flows of control, meaning that each waits until all incoming paths have reached the join.

In UML, activities are defined as non-atomic computations thus capable of being composed by subactivities. Borrowing the concept of composite state from general state machines, activity diagrams are provided a way of composing several actions/activities into new activities. A composite state may contain either sequential or concurrent substates. Nevertheless, the composite is in essence non-atomic, allowing transitions out of the composite with the effect of interrupting its entire computation process. In other words, activity diagrams lack a composite state for atomic composition of activities.

To overcome the lack of an atomic subactivity state, we introduce a new notation for a composite state based on the idea of atomic transaction. The notation is based on extension mechanisms of UML [3, 10] and introduce new constraints over the basic composite state, so the new "transaction state" is not allowed to be interrupted by explicit external events. The new composite state is decorated with the stereotype $<< transaction >>$ as depicted in figure 2. The semantics for the whole activity diagram and specially the transaction composite is presented in the next sections.

3 Nonsequential Automata

When employing Petri net based models to represent systems, one must be aware that basic activities of each component may involve smaller internal activities. In other words, transitions are conceptually refined into transactions.

A convenient top-down approach to represent transactions is to start with an abstract model and then

[1] In this case, ordinary state machines and collaboration diagrams are preferable.

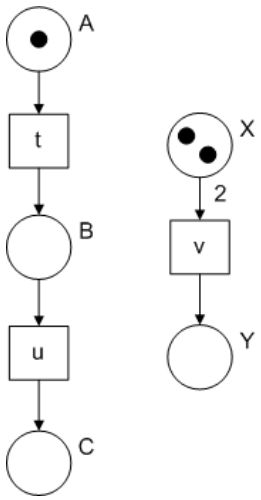

Figure 3: Marked place/transition Petri net

refine each transition, that might represent a complex activity, into a refined net that offers a more precise representation of the activity. Several approaches have appeared in the literature ([5], [12], [13]) that present different techniques for refinement. The approaches are related to the notion of general net morphism proposed by Petri, in which a refined net is collapsed into a transition of a more abstract counterpart. Here, we concentrate on the model named nonsequential automata.

Nonsequential Automata ([11], [13], [15], [14]) constitute a non interleaving semantic domain, with its foundations on category theory, for reactive, communicating and concurrent systems. Nonsequential Automata follows the so-called "Petri nets are monoids" approach [16] and is similar to Petri nets, but it is a more concrete model - it can be seen as computations from a given place/transition net.

Petri net, in this paper, means the general case of place/transition net with weighted arcs and without capacity restriction for places. So, we recall the definition of place/transition nets and then Petri nets as graphs in order to clarify the notation used for defining nonsequential automata. Also, for the sake of simplicity, we drop the notion of initial marking and initial/final states. The more interested reader is referred to [17] and [18] for more details.

Definition 1 (Place/Transition Net) *A place/transition net is a triple $N = \langle S, T, F\rangle$ where S is a set of places, T is a set of transitions, with S and T disjoints, and $F : (S \times T) \cup (T \times S) \to \mathbb{N}$ is the flow relation.*

The flow relation specifies how many tokens are consumed or produced in each place when a transition fires. A marking $m : S \to \mathbb{N}$, which is a finite multiset of places representing the number of tokens in each place, will be written $m = \{n_1A_1, \ldots, n_kA_k\}$ with n_i the number of tokens in place A_i (i ranging over $1, \ldots, k$) or as $n_1A_1 \oplus \ldots \oplus n_kA_k$ where $\oplus$ is the monoidal operation. For example, figure 3 represents the Petri net $N = \langle\{A, B, C, X, Y\}, \{t, u, v\}, \{(A, t), (t, B), (B, u), (u, C), (X, v), (v, Y)\}\rangle$ with initial marking $m = \{A, 2X\}$.

In order to define Petri net as a graph, which is more suitable to our discussion, we follow the same approach in [16], where nodes are elements of a commutative monoid. In this case, nodes and arcs stand for states and transitions of a net, where for each transition t, n_i tokens consumed in places A_i and n_j tokens produced in places B_j will be written the arc $t : \bigoplus n_iA_i \to \bigoplus n_jB_j$, for i and j ranging over $1, \ldots, k$. The transitions on the Petri net of figure 3 are represented by arcs $t : A \to B$, $u : B \to C$ and $v : 2X \to Y$. Therefore, a Petri net is basically a graph with a monoidal structure on nodes.

Before going any further, let us clarify all the semantic models in this paper are introduced as a class of objects equipped with a notion of behavior-preserving morphism, making each model into a category.

Definition 2 (Petri Category) *A (place/transition) Petri net is a graph $N = \langle S^{\oplus}, T, \delta_0, \delta_1\rangle$ where the set of nodes $S^{\oplus}$ is the free commutative monoid generated by the set of places and $\delta_0, \delta_1 : T \to S^{\oplus}$ are total functions called source and target of arcs in T. A Petri net morphism $h : N \to N'$ is a graph morphism $h = \langle h_S : S^{\oplus} \to S'^{\oplus}, h_T : T \to T'\rangle$ i.e. $h_S \circ \delta_i = \delta'_i \circ h_T$ for $i \in \{0, 1\}$ where h_S is required to be a monoid homomorphism. Petri nets as graphs and their morphisms define the category* ***Petri****.*

This view of Petri nets as graphs was based on the idea of nodes as elements of a commutative monoid over the set of places. The figure 4 depicts the behavior of the net in figure 3 when starting with a specific marking. What then if we change the initial marking? We have to compute all reachable markings again.

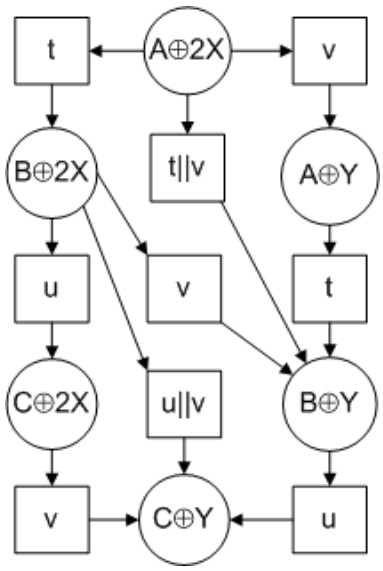

Figure 4: Behavior of a Petri net

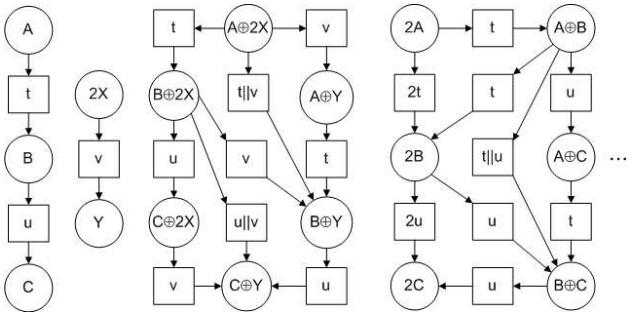

Figure 5: A nonsequential automaton corresponding to a Petri net

But what if we could get a more concrete model with all possible markings and capable of making explicit all implicit concurrencies in the net? This is the key for the nonsequential automata.

Nonsequential Automata constitute a categorical semantic domain around the concepts of state and transition. It consists of a reflexive graph with monoidal structure on both states and transitions, initial and final states and labeling on transitions. The interpretation of a structured state is the same as in Petri nets: it is viewed as a "bag" of local states representing a notion of tokens to be consumed or produced. A structured transition is a way to specify that the component transitions are independent i.e., structured transitions specify which component transitions are concurrent with another (see, for example, transitions $t||v$ and $u||v$ in figure 5).

In the next definitions **CMon** denotes the category of commutative monoids and $k \in \{0, 1\}$ (for simplicity, we omit that $k \in \{0, 1\}$).

Definition 3 (Nonsequential Automaton) *A nonsequential automaton* $NA = \langle \mathrm{V}, \mathrm{T}, \delta_0, \delta_1, \iota, \mathrm{L}, lab \rangle$ *is such that* $\mathrm{V} = \langle V, \oplus, 0 \rangle$, $\mathrm{T} = \langle T, ||, \tau \rangle$, $\mathrm{L} = \langle L, ||, \tau \rangle$ *are* ***CMon****-objects of states, transitions and labels respectively,* $\delta_0, \delta_1 : \mathrm{T} \to \mathrm{V}$ *are* ***CMon****-morphisms called source and target respectively,* $\iota : \mathrm{V} \to \mathrm{T}$ *is a* ***CMon****-morphism for mapping identities, and* $lab : \mathrm{T} \to \mathrm{L}$ *is a* ***CMon****-morphism for labeling transitions such that* $lab(t) = \tau$ *whenever there is* $v \in V$ *where* $\iota(v) = t$*. Therefore, a nonsequential automaton can be seen as* $NA = \langle \mathrm{G}, \mathrm{L}, lab \rangle$ *where* $\mathrm{G} = \langle \mathrm{V}, \mathrm{T}, \delta_0, \delta_1, \iota \rangle$ *is a reflexive graph internal to* ***CMon*** *representing the automaton shape,* L *is a commutative monoid representing the labels of the transitions and lab is the labeling morphism associating a label to each transition.*

In a nonsequential automaton, a transition labeled by τ represents a hidden transition. As the automaton shape is represented by a reflexive graph, each state has an associated identity transition which is interpreted as a "no operation" or "idle". Note that, by definition, all identity transitions are hidden (i.e. labeled by τ).

The nonsequential automaton in figure 5 is represented by $\langle \{A, B, C, X, Y\}^{\oplus}, \{t, u, v\}^{||}, \delta_0, \delta_1, \iota, \{t, u, v\}^{||}, lab \rangle$ with δ_0, δ_1, ι determined by transitions $t : A \to B$, $u : B \to C$, $v : 2X \to Y$, and labeling $t \mapsto t$, $u \mapsto u$,

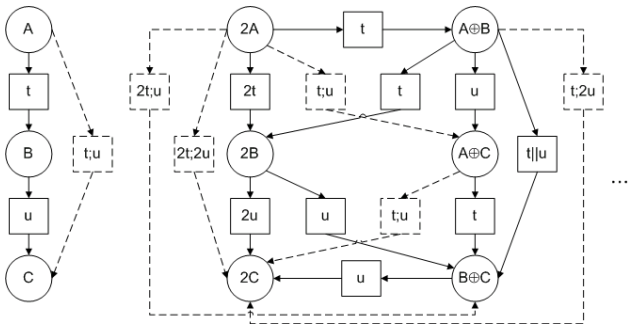

Figure 6: Nonsequential automaton and its corresponding computational closure

$v \mapsto v$. In the graphical representation of the automaton identity arcs are omitted and, for a given node A and arcs $t : X \to Y$ and $\iota_A : A \to A$, the structured arc $t||\iota_A : X \oplus A \to Y \oplus A$ will be simply noted by $t : X \oplus A \to Y \oplus A$. In this sense, in figure 5, the identity arcs $\iota_A, \iota_B, \iota_C, \iota_{2X}, \iota_Y, \iota_{A\oplus B}, \ldots$ were omitted, and transitions like $t||\iota_Y : A \oplus Y \to B \oplus Y$ where written $t : A \oplus Y \to B \oplus Y$.

Two important points must be discussed before going any further. First, the nonsequential automaton in figure 5 was not completely drawn as it has infinite distinguished nodes, for they are elements of a freely generated monoid chosen to represent its states. This choice was not obligatory as the definition of nonsequential automaton mentions any commutative monoid. This is different from other Petri net related models in which the monoid must be freely generated. Second, structured transitions, like $t||v$, explicitly determines the "independence square", i.e. transitions t and v are independent, and should not be confused with synchronized transitions presented by Meseguer and Montanari [16].

Definition 4 (NAut Category) *A nonsequential automaton morphism $h : NA \to NA'$ with $NA = \langle \mathrm{V}, \mathrm{T}, \delta_0, \delta_1, \iota, \mathrm{L}, lab\rangle$ and $NA' = \langle \mathrm{V}', \mathrm{T}', \delta_0', \delta_1', \iota', \mathrm{L}', lab'\rangle$ is a triple $h = \langle h_V, h_T, h_L\rangle$ with $h_V : \mathrm{V} \to \mathrm{V}'$, $h_T : \mathrm{T} \to \mathrm{T}'$, $h_L : \mathrm{L} \to \mathrm{L}'$ **CMon**-morphisms, such that $h_V \circ \delta_k = \delta_k' \circ h_T$, $h_T \circ \iota = \iota' \circ h_V$ and $h_L \circ lab = lab' \circ h_T$. Nonsequential automata and their morphisms constitute the category **NAut**.*

Given an appropriate notion of morphisms between two nonsequential automata defined above, we are able to define atomic composition of transitions through the concept of refinement. A refinement maps transitions into transactions reflecting an implementation of an automaton on top of another, based on constructions presented in [15]. It is defined as a special morphism of automata where the target one (more concrete) is enriched with its computational closure (all the conceivable sequential and nonsequential computations that can be split into permutations of original transitions). The construction of the computational closure was inspired by [16]. The idea for building the computational closure here is analogous to the example previously discussed for state machines, except for the fact it also includes nonsequential computations besides sequential ones. Considering the nonsequential automaton in figure 5, its computational closure is partially depicted in figure 6 (added transitions were drawn with a dotted pattern). Please note a composition operator ";" appeared in the structured transitions.

The computational closure of a nonsequential automaton is formally defined as the composition of two adjoint functors between the **NAut** category and the category **CNAut** (defined in the next paragraphs) of nonsequential automata enriched with it computations: the first one basically enriches an automaton with a composition operation on transitions, and the second functor forgets about the composition operation. Also, some equations about concurrency are introduced. This composition leads to an endofunctor called transitive closure functor tc. Then, the refinement morphism φ from NA into (the computations of) NA' can be defined as $\varphi : NA \to \mathsf{tc}NA'$.

In the text that follows, the categories are built using the approach known as internalization [2], leading to the notion of structured (internal) graphs, where nodes and arcs may be objects of different categories. The category of categories internal to **CMon** (remember **CMon** is the category of commutative monoids) is denoted by **Cat(CMon)** and **RGr(CMon)** is the category of reflexive graphs internal to **CMon**.

Definition 5 (CNAut Category) *Consider the category **Cat(CMon)**. The category **CNAut** is the comma category $id_{Cat(CMon)} \downarrow id_{Cat(CMon)}$ where $id_{Cat(CMon)}$ is the identity functor in **Cat(CMon)**. Therefore, a **CNAut**-object is a triple $CNA = \langle \mathsf{G}, \mathsf{L}, lab\rangle$ where G, L are **Cat(CMon)**-objects and lab is a **Cat(CMon)**-morphism.*

Notice that in order to build the computations, we have enriched **NAut** by the substitution of its shape from a reflexive internal graph $\mathrm{G} = \langle \mathrm{V}, \mathrm{T}, \delta_0, \delta_1, \iota\rangle$ to a **Cat(CMon)**-object $\mathsf{G} = \langle \mathrm{V}, \mathrm{T}, \delta_0, \delta_1, \iota, ;\rangle$ with a

composition operation, and similarly with its labels. The composition operation ";" was responsible for the newly added transitions in figure 6. This construction is formally defined by the following functor:

Definition 6 (Functor nc) *Let $NA = \langle \mathrm{G}, \mathrm{L}, lab \rangle$ be a **NAut**-object and $h : NA \to NA'$ be a **NAut**-morphism. The functor $\mathsf{nc} : \textbf{\textit{NAut}} \to \textbf{\textit{CNAut}}$ is such that:*

- ***RGr(CMon)**-object $\mathrm{G} = \langle \mathrm{V}, \mathrm{T}, \delta_0, \delta_1, \iota \rangle$ is taken into the **Cat(CMon)**-object $\mathsf{G}' = \langle \mathrm{V}, \mathrm{T}', \delta_0', \delta_1', \iota', ; \rangle$ with ι' induced by ι and T', δ_0', δ_1', $\_;\_ : \mathrm{T}' \times \mathrm{T}' \to \mathrm{T}'$ inductively defined as follows*

$$\frac{t{:}a{\to}b \in \mathrm{T}}{t{:}a{\to}b \in \mathrm{T}'} \qquad \frac{t{:}a{\to}b \in \mathrm{T}' \;\; u{:}b{\to}c \in \mathrm{T}'}{t;u{:}a{\to}c \in \mathrm{T}'} \qquad \frac{t{:}a{\to}b \in \mathrm{T}' \;\; u{:}c{\to}d \in \mathrm{T}'}{t||u{:}a{\oplus}c{\to}b{\oplus}d \in \mathrm{T}'}$$

  subject to the following equational rules

$$\frac{t \in \mathrm{T}'}{\tau;t = t \;\; t;\tau = t} \qquad \frac{t{:}a{\to}b \in \mathrm{T}'}{\iota_a;t = t \;\; t;\iota_b = t} \qquad \frac{t{:}a{\to}b \in \mathrm{T}' \;\; u{:}b{\to}c \in \mathrm{T}' \;\; v{:}c{\to}d \in \mathrm{T}'}{t;(u;v) = (t;u);v}$$

$$\frac{t \in \mathrm{T}' \;\; u \in \mathrm{T}'}{t||u = u||t} \qquad \frac{t \in \mathrm{T}'}{t||\tau = t} \qquad \frac{\iota_a \in \mathrm{T}' \;\; \iota_b \in \mathrm{T}'}{\iota_a||\iota_b = \iota_{a \oplus b}} \qquad \frac{t \in \mathrm{T}' \;\; u \in \mathrm{T}' \;\; v \in \mathrm{T}'}{t||(u||v) = (t||u)||v}$$

- ***CMon**-object L is taken into the **Cat(CMon)**-object $\mathsf{L}' = \langle 1, \mathrm{L}', !, !, !_\iota, ; \rangle$ with L' inductively defined as above, and $!$ and $!_\iota$ meaning the unique obvious mappings.*

- *The **NAut**-object $NA = \langle \mathrm{G}, \mathrm{L}, lab \rangle$ is taken into the **CNaut**-object $CNA = \langle \mathsf{G}', \mathsf{L}', lab' \rangle$ where lab' is the morphism induced by lab.*

- *The **NAut**-morphism $h = \langle h_V, h_T, h_L \rangle$ is taken into the **Cat(CMon)**-morphism $h = \langle h_G, h_L \rangle : CNA \to CNA'$ where $h_G = \langle h_V, h_{T'} \rangle$, $h_L = \langle !, h_{L'} \rangle$ and $h_{T'}$, $h_{L'}$ are the monoid morphisms generated by the monoid morphisms h_T and h_L, respectively.*

Definition 7 (Functor cn) *Let $CNA = \langle \mathsf{G}, \mathsf{L}, lab \rangle$ be a **CNAut**-object and $h : CNA \to CNA'$ be a **CNAut**-morphism. The functor $\mathsf{cn} : \textbf{\textit{CNAut}} \to \textbf{\textit{NAut}}$ is such that:*

- ***Cat(CMon)**-object $\mathsf{G} = \langle \mathrm{V}, \mathrm{T}, \delta_0, \delta_1, \iota, ; \rangle$ is taken into the **RGr(CMon)**-object $\mathrm{G}' = \langle \mathrm{V}, \mathrm{T}', \delta_0', \delta_1', \iota' \rangle$, where T' is T subject to the equational rule*

$$\frac{t{:}a{\to}b \in \mathrm{T}' \;\; u{:}b{\to}c \in \mathrm{T}' \;\; t'{:}a'{\to}b' \in \mathrm{T}' \;\; u'{:}b'{\to}c' \in \mathrm{T}'}{(t;u)||(t';u') = (t||t');(u||u')}$$

  and δ_0', δ_1', ι' are induced by δ_0, δ_1, ι, restricted to T'.

- *The **Cat(CMon)**-object $\mathsf{L} = \langle \mathrm{V}, \mathrm{L}, \delta_0, \delta_1, \iota, ; \rangle$ is taken into the **CMon**-object L', where L' is L subject to the analogous equational rule.*

- *The **CNAut**-object $CNA = \langle \mathsf{G}, \mathsf{L}, lab \rangle$ is taken into the **NAut**-object $NA = \langle \mathrm{G}', \mathrm{L}', lab' \rangle$ with lab' the **RGr(CMon)**-morphism canonically induced by the **Cat(CMon)**-morphism lab.*

- *The **CNAut**-morphism $h = \langle h_G, h_L \rangle$ with $h_G = \langle h_{G_V}, h_{G_T} \rangle$, $h_L = \langle h_{L_V}, h_{L_T} \rangle$ is taken into the **NAut**-morphism $h = \langle h_{G_V}, h_{G_{T'}}, h_{L_{T'}} \rangle : NA \to NA'$ where $h_{G_{T'}}$ and $h_{L_{T'}}$ are the monoid morphisms induced by h_{G_T} and h_{L_T} respectively.*

Definition 8 (Transitive Closure Functor) *The transitive closure functor is $\mathsf{tc} = \mathsf{cn} \circ \mathsf{nc} : NAut \to NAut$.*

The functor cn introduces an important requirement about concurrency in the computational closure: $(t;u)||(t';u') = (t||t');(u||u')$. That is, the computation determined by two independent composed transitions $t;u$ and $t';u'$ is equivalent to the computation whose steps are the independent transitions $t||t'$ and $u||u'$. As illustration, consider transitions $t : A \to B$ and $u : B \to C$ from our working example (figure 6). For $t||u : A \oplus B \to B \oplus C$ we have

$$t||u = (\iota_A;t)||(u;\iota_C) = (\iota_A||u);(t||\iota_C) = u;t$$

$$u||t = (\iota_B;u)||(t;\iota_B) = (\iota_B||t);(u||\iota_B) = t;u$$

Thus, the concurrent execution of two independent transitions is equivalent to its sequential execution in any order. Also, the inference rules of the functor nc inductively defined the new set of composed transitions, adding, for example, the transition $t;2u : A \oplus B \to 2C$ by sequentially composing $t : A \oplus B \to 2B$ and

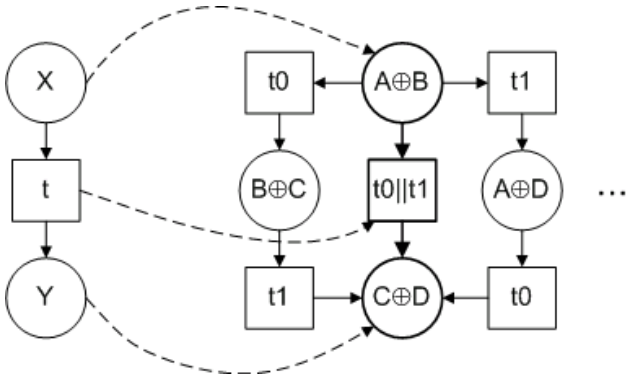

Figure 7: Refinement morphism for transaction $t : X \to Y$

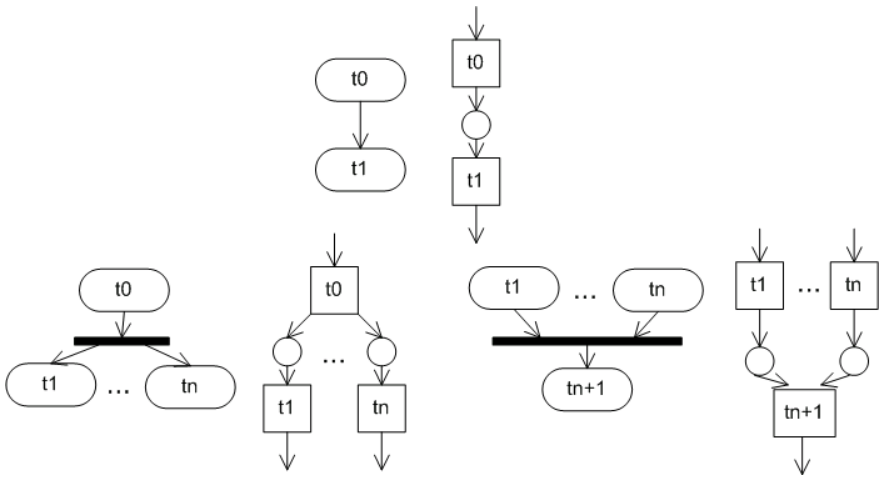

Figure 8: Basic translation schemes from activity diagram to nonsequential automaton

$2u : 2B \to 2C$. But how about transition $u;t;u : A \oplus B \to 2C$? It was not drawn in the figure because several transitions actually represent equivalence classes induced by the set of equational rules. In the case of $u;t;u$ it is in the same class of $t;2u$:

$$t;2u = t;(u||u) = (t||\iota_B);(u||u) = (t;u)||(\iota_B;u) = (t;u)||u = u||(t;u) =$$

$$(u;\iota_C)||(\iota_A;(t;u)) = (u||\iota_A);(\iota_C||(t;u)) = u;(\iota_C||(t;u)) = u;((t;u)||\iota_C) = u;t;u$$

To illustrate the refinement morphism, given two nonsequential automata NA and NA' with free monoids on states and labeled transitions respectively induced by transitions $t : X \to Y$, and $t_0 : A \to C$, $t_1 : B \to D$, suppose we want to build a transaction containing both t_0 and t_1. First we apply the transitive closure functor tc, enriching NA' with all sequential and nonsequential computations. For the last step we need to build the refinement morphism by mapping the corresponding states and transitions. The refinement $\varphi : NA \to \mathsf{tc}NA'$ is given by $X \mapsto A \oplus B$, $Y \mapsto C \oplus D$, $t \mapsto t_0||t_1$ (see figure 7). Notice that due to the equations, we actually get a class of transitions containing $t_0||t_1$, $t_0;t_1$ and $t_1;t_0$, represented as $t_0||t_1$ in the figure.

4 Mapping Activity Diagrams to Nonsequential Automata

As we can see from previous sections, an activity diagram is already close to a Petri net. So, the basic translation schemes from activity diagrams into nonsequential automata are targeted into constructing local transitions for a nonsequential automaton.

Figure 8 shows this principle for action states and complex transitions. The labeling for transitions corresponds to labels from the diagram states. Initial and final states represent the corresponding initial and final places/states of the automaton [2].

[2]The explicit representation of initial and final states for nonsequential automata was not presented in order to simplify the discussion of the semantic model.

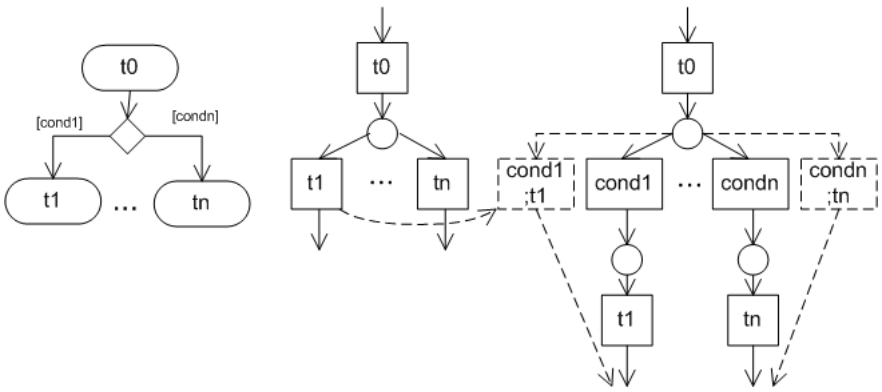

Figure 9: Translation scheme for guarded transitions

The translation of guarded transitions is based on the refinement morphism from nonsequential automata. It is a two step translation: first, conditions are translated to transitions corresponding to a nondeterministic choice following figure 9, then, a refinement associates each flat transition t_i to a sequential composition $cond_i; t_i$.

The central core of the composite transaction state also makes use of nonsequential automata refinement. The source automaton corresponds to the basic translation using the previous mappings, where the composite state is viewed as only one state. The target automaton corresponds to the translation taking into account the subactivity states of the composite. The refinement then maps the more abstract transition into the concrete implementation of the transaction obtained via the computational closure of the target automaton. Figure 10 partially depicts the target nonsequential automaton for our initial example of activity diagrams (figure 1). Notice it explicits all possible computational paths, including the transition labeled $Evalx||(Evaly; Atriby)$ as the desired behavior for the fork/join construction in the activity diagram. For the refinement, the transaction state is represented by the atomic composition $Evaly; Atriby$.

Due to space limitations and to simplify the presentation, in this paper we have not dealt with event generation and handling. Generally speaking, for Petri net related models, events may be modeled as tokens or transitions with different consequences on the resulting behavior (see [7] for a discussion on both alternatives). Notice when modeling reactive systems, events should not be abstracted away and thus activity diagrams alone are not sufficient in the development process.

5 Conclusion

Models capture the important aspects of the system being modeled from a certain point of view and simplify or omit the rest. Which aspects are essential is a matter of judgment that depends on the purpose of each model. We believe transactions are an important part of today systems and they deserve a first class mechanism in modeling languages, specially UML.

Following that premise, this paper presented an extension to UML activity diagrams centered on a construction for defining atomic composition of actions and activities. We have extended the UML notation with a particular stereotype for composite states and defined its semantics by the means of refinement morphisms over nonsequential automata.

Different lines of research are related to this work, manly semantics based on Abstract Machines, Petri nets and Workflow Statecharts. In [4] abstract state machines are used to provide semantics for activity diagrams in UML; [19] uses a process-like algebra for describing activity diagrams; and extensive work has been put on defining semantics for modeling workflow in [7, 6].

Some aspects of activity diagrams have been left out of this presentation and need further investigation. The most important are events and its associated action semantics.

Acknowledgment

This work was partially supported by CNPq (Project HoVer-CAM, GRAPHIT, E-Automaton) and FINEP/CNPq (Project Hyper-Seed) in Brazil.

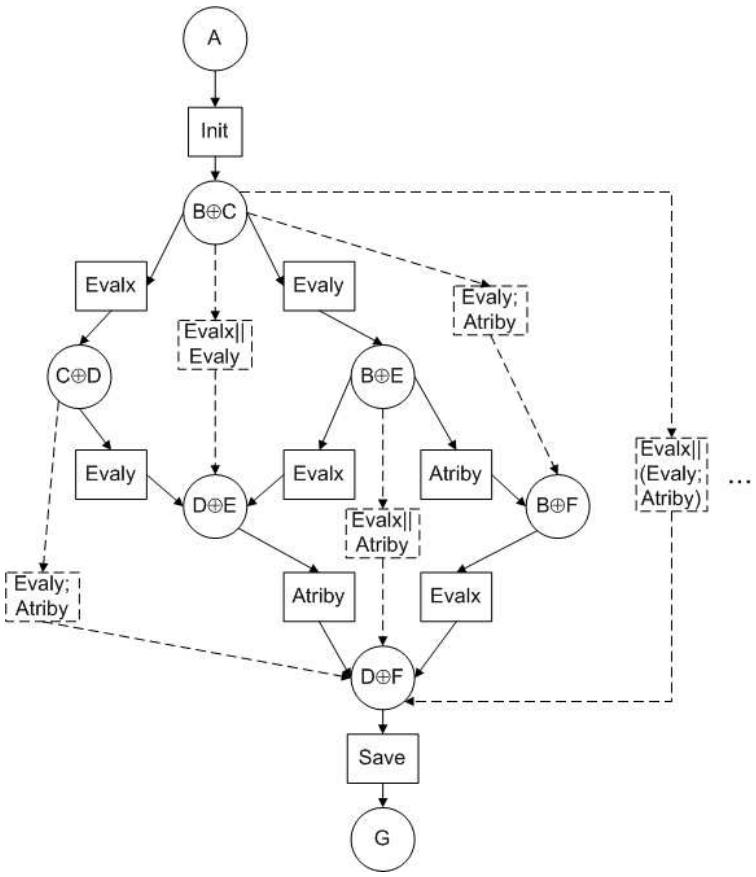

Figure 10: Nonsequential automata for activity diagram

References

[1] C. André and J. P. Rigault. Variations on the semantics of graphical models for reactive systems. In *System Management and Cybernetics*. IEEE Press, 2002.

[2] A. Asperti and G. Longo. *Categories, Types and Structures: an introduction to category theory for the working computer scientist*. MIT Press, Cambridge, 1990.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[4] E. Borger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In *Lecture Notes in Computer Science - 8th International Conference on Algebraic Methodology and Software Technology*, volume 1816, pages 293–308. Springer-Verlag, 2000.

[5] W. Brauer, R. Gold, and W. Vogler. A survey of behaviour and equivalence preserving refinements of petri nets. In *Lecture Notes in Computer Science - Advances in Petri Nets 1990*, volume 483, pages 1–46. Springer-Verlag, 1991.

[6] R. Eshuis. *Semantics and Verification of UML Activity Diagrams for Workflow Modelling*. PhD thesis, University of Twente, 2002.

[7] R. Eshuis and R. Wieringa. Comparing petri net and activity diagram variants for workflow modelling - a quest for reactive petri nets. In *Lecture Notes in Computer Science - Petri Net Technology for Communication Based Systems*, volume 2472, pages 321–351. Springer-Verlag, 2003.

[8] M. Fowler and K. Scott. *UML Distilled - A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2000.

[9] T. Gehrke, U. Goltz, and H. Wehrheim. The dynamic models of UML: Towards a semantics and its application in the development process. Technical Report 11/98, Institut fur Informatik, Universitat Hildesheim, 1998.

[10] Object Management Group. *Unified Modeling Language Specification, version 1.5*. Rational Software Corporation, 2003.

[11] P. Blauth Menezes and J. F. Costa. Compositional reification of concurrent systems. *Journal of the Brazilian Computer Society*, 2(1):50–67, 1995.

[12] P. Blauth Menezes and J. F. Costa. Synchronization in petri nets. *Fundamenta Informaticae*, 26(1):11–22, 1996.

[13] P. Blauth Menezes, J. F. Costa, and A. S. Sernadas. Refinement mapping for general (discrete event) system theory. In *Lecture Notes in Computer Science - 5th International Conference on Computer Aided Systems Theory and Technology*, volume 1030, pages 103–116. Springer-Verlag, 1996.

[14] P. Blauth Menezes, J. P. Machado, and S. A. da Costa. Explicit and implicit nondeterministic refinement for concurrent, interacting systems. In *PDPTA2002 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, 2002. CSREA.

[15] P. Blauth Menezes, A. S. Sernadas, and J. F. Costa. Nonsequential automata semantics for concurrent, object-based language. In *Electronic Notes in Theoretical Computer Science - 2nd US-Brazil Joint Workshops on the Formal Foundations of Software Systems*, volume 14. Elsevier, 1998.

[16] J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105–155, oct 1990.

[17] W. Reisig. *Petri Nets: an introduction*, volume 4 of *Eatcs Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[18] W. Reisig and G. Rozenberg, editors. *Lectures on Petri nets: advances in Petri nets*. Springer-Verlag, 1998. Lecture Notes in Computer Science 1491.

[19] R. W. S. Rodrigues. Formalising UML activity diagrams using finite state processes. In *Workshop on Dynamic Behaviour in UML Models, 3rd International Conference on the Unified Modeling Language*, 2000.

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

[21] G. Winskel and M. Nielsen. *Handbook of Logic in Computer Science*, volume 4, chapter Models for Concurrency, pages 1–148. Oxford University Press, 1995.