Journal of Universal Computer Science, vol. 12, no. 7 (2006), 958-979
submitted: 28/1/06, accepted: 6/4/06, appeared: 28/7/06 © *J.UCS*

Defining Atomic Composition in UML Behavioral Diagrams

Júlio Pereira Machado
(Pontifícia Universidade Católica do Rio Grande do Sul, Brazil,
juliopm@inf.pucrs.br)
Paulo Blauth Menezes
(Universidade Federal do Rio Grande do Sul, Brazil, blauth@inf.ufrgs.br)

Abstract: UML may be used to describe both the structure and behavior of object-oriented systems using a combination of notations. For the modeling of the dynamic behavior, a number of different models are offered such as interaction, state and activity diagrams. Although compositional techniques for modeling computational processes demand means of composing elements both in non-atomic or atomic ways, UML seems to lack compositional constructs for defining atomic composites. We discuss proper extensions for diagrams that are able to cope with the concept of atomic composition as the basic element for describing transactions (in our settings the term "transaction" denotes a certain operation of a system that might be atomically composed by many, possibly concurrent, operations). Atomic compositions are then formally defined through a special morphism between automata in a domain called Nonsequential Automata.
Key Words: UML, semantics, nonsequential automata
Category: F.3.2, F.1.1, D.2.0, D.1.5

1 Introduction

The Unified Modeling Language (UML) [Rumbaugh et al. 2004] may be used to describe both the structure and behavior of object-oriented systems using a combination of notations. For the modeling of the dynamic behavior, a number of different models are offered such as interaction, state and activity diagrams.

When modeling concurrent or parallel systems with such diagrams, we must be aware that basic activities of each system may be constituted by smaller activities, i.e. transitions may be conceptually refined into transactions. This important notion is present in different fields of computer science like operating system's primitives, implementation of synchronization methods for critical regions, database management systems and protocols for communications, just to name a few. In this sense, when modeling computational processes, we need means of composing subactivities both in a non-atomic or atomic way. Nevertheless UML seems to lack compositional constructs for defining atomic actions/activities/operations and, nowadays, major information systems like web-based shopping, web-services, e-busines, etc, are transactional in its majority and are being modeled in UML (using activity and state machine diagrams) without a notation for concerning which behaviors are transactional.

In this work, we concentrate on describing groups of sequential or concurrent activities that are responsible for performing a computation, and we address the issue of modeling atomic compositions for transactions. We remark that in our settings the term "transaction" denotes a certain atomic operation of a system that might be composed by many, possibly concurrent, operations. Although denoting atomic computations in a concurrent scenario by the term "transaction" is a slight abuse of terminology (specially in the field of databases), these abstract notion for transactions has also been employed by others e.g. [Bruni and Montanari 2004]. Also, even though most commercial applications are based on transactions for which ACID properties (atomicity, consistency, isolation and durability) must be guaranteed in some form, transactions primitives are all based on the same idea of grouping series of actions in atomic blocks. Algorithms for correctly implementing transactions may be found in specialized topics on database systems [Ullman and Widom 2002] or operating systems [Tanenbaum 2001].

In order to correctly introduce the notion of transactions, we need to analyze the UML official documentation. The UML specification by OMG [OMG 2005b] [OMG 2004] is built on a semi-formal semantics, composed by a set of metalanguage, restrictions and text in natural language. The metalanguage is basically a set of class diagrams which describe the basic building blocks of UML models (it can be seen as the abstract syntax of the language). The Object Constraint Language (OCL) [OMG 2005a] further defines constraints over models so they can be considered well-formed. In our approach, the idea is to focus only on necessary constructs from the UML metamodel for exposing the behavior (to be understood as a sequence of observable actions) of software artifacts. From this set, we extend the metamodel with elements denoting atomic composites. The graphical notations for the new composites are based on the non-atomic ones and are further decorated with proper stereotypes. Also, new OCL expressions are built to define new constraints over atomic compositions. Finally, the well-formed models are mapped to nonsequential automata, thus formally defining its semantics. In this paper we present the mapping to nonsequential automata and we do not address the full UML profile for atomic composites.

Nonsequential Automata [Menezes et al. 1996] [Menezes et al. 1998] constitute a non-interleaving semantic domain, with its foundations on category theory, for reactive, communicating and concurrent systems. It follows the so-called "Petri nets are monoids" approach [Meseguer and Montanari 1990] 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.

The rest of the paper is organized as follows. Section 2 briefly presents nonsequential automata, which is going to be used as the semantics for atomic composition in UML. Section 3 introduces (through working examples) translation

schemes for building nonsequential automata from activity and state machine diagrams. Finally, sections 4 and 5 discusses the results and outlines possible directions for future investigations.

2 Nonsequential Automata

Nonsequential Automata [Menezes et al. 1996] [Menezes et al. 1998] constitute a categorical semantic domain around the concepts of state and transition following the "Petri nets are monoids" approach by [Meseguer and Montanari 1990]. It was developed to supply a compositional domain with refinement capabilities and it is a more concrete model than Petri nets (it can be seen as computations from a given place/transition net).

To gently introduce the idea behind nonsequential automata, we start with an example of a simple place/transition Petri net as presented in [Reisig 1985]. The Petri net in [Fig. 1] (left) has three places $\{A, B, C\}$ and two transitions $\{t, u\}$, in which $t(u)$ consumes one token from place $A(B)$ and produces one token in place $B(C)$.

Following the "token game" we can compute all possible reachable markings from an initial marking by applying all transitions that are currently enabled. In [Fig. 1] (right) we present a case graph depicting the reachable markings for the Petri net of our example starting from the marking consisting of two tokens in place A. Notice the graph shows that both transitions t and u may be fired concurrently in a certain marking, i.e. $t||u$. This view of Petri nets as graphs was based on the idea of nodes as elements of a commutative monoid over the set of states (see the symbol $\oplus$ for monoidal operator in the states).

The [Fig. 1] depicted the behavior of the Petri net when starting with a specific marking. What then if we change the initial marking? We have to compute all reachable markings again. 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 automaton (partially) depicted in [Fig. 2].

In the next definitions **CMon** denotes the category of commutative monoids. A monoid will be denoted $\langle M, \oplus, e\rangle$, where M is a set, $\oplus$ is an associative binary operation on M, and e is an identity for $\oplus$.

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, $\delta_0, \delta_1 : \mathrm{T} \rightarrow \mathrm{V}$ are **CMon**-morphisms called source and target respectively, $\iota : \mathrm{V} \rightarrow \mathrm{T}$ is a **CMon**-morphism for mapping identities, and $lab : \mathrm{T} \rightarrow \mathrm{L}$ 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

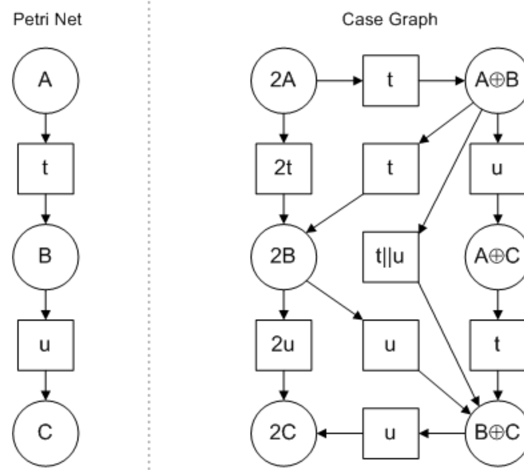

Figure 1: Place/transition Petri net (left) and corresponding case graph for initial marking (right)

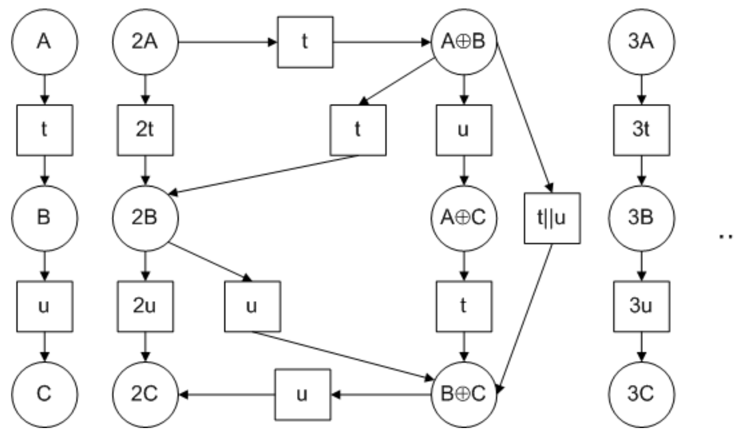

Figure 2: Nonsequential automaton

internal to **CMon** representing the automaton shape, L is a commutative monoid representing the labels of transitions and *lab* is the labeling morphism associating a label to each transition. In this definition, a transition labeled by τ represents a hidden transition, and each state has an associated identity transition which is interpreted as a "no operation" or "idle" (and by definition are labeled by τ).

According to the definition, the automaton consists of a reflexive graph with monoidal structure on both states and transitions, 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. For example, $\langle \{A, B, C\}^{\oplus}, \{t, u\}^{||}, \delta_0, \delta_1, \iota, \{t, u\}^{||}, lab \rangle$ with δ_0, δ_1, ι determined by transitions $t : A \to B$, $u : B \to C$, and labeling $t \mapsto t$, $u \mapsto u$, is represented in [Fig. 2] (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 $t||\iota_A : X \oplus A \to Y \oplus A$ is simply noted $t : X \oplus A \to Y \oplus A$). This nonsequential automaton was not completely drawn as it has infinite distinguished nodes, for they are elements of a freely generated monoid chosen to represent its states. Also, structured transitions, like $t||u$, explicitly determine the "independence square", i.e. transitions t and u are independent.

In order to enrich the model we define next a notion of morphism. 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 $h_V \circ \delta_k = \delta_k' \circ h_T$ (for $k \in \{0, 1\}$), $h_T \circ \iota = \iota' \circ h_V$ and $h_L \circ lab = lab' \circ h_T$. Nonsequential automata and their morphisms constitute the category **NAut**.

We are able to define atomic composition of transitions through the concept of refinement. 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). Considering the previous nonsequential automaton, its computational closure is also partially depicted in [Fig. 3] (added transitions were drawn with a dotted pattern). Please note a composition operator ";" appeared in the structured transitions.

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

In the text that follows, the categories are built using the approach known as internalization [Asperti and Longo 1990], 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** is denoted by **Cat(CMon)** and **RGr(CMon)** is the category of reflexive graphs internal to **CMon**. Details on defining these internal categories may be found in [Menezes et al. 1996].

A nonsequential automaton enriched with its computations $CNA = \langle \mathsf{G}, \mathsf{L}, lab \rangle$ is such that G, L are **Cat(CMon)**-objects and lab is a **Cat(CMon)**-morphism.

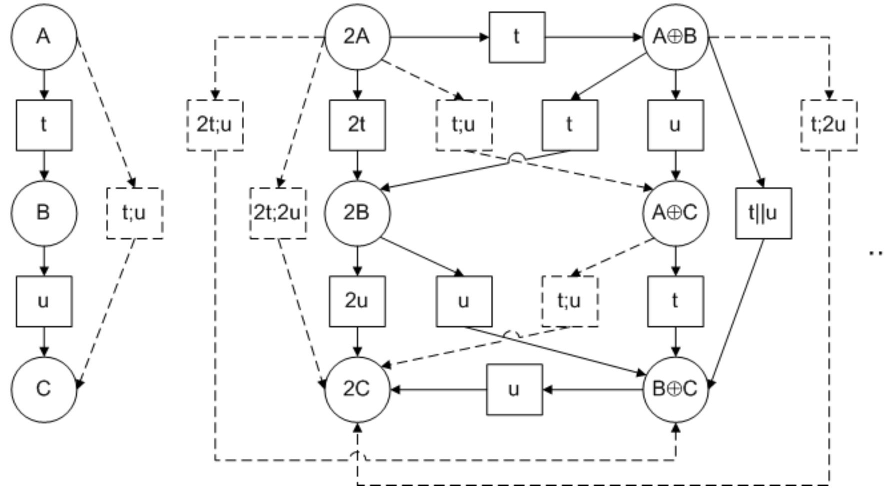

Figure 3: Nonsequential automaton and its computational closure

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 [Fig. 3].

Let $NA = \langle \mathrm{G}, \mathrm{L}, lab \rangle$ be a **NAut**-object and $h : NA \to NA'$ a **NAut**-morphism. The functor $\mathsf{nc} : \mathbf{NAut} \to \mathbf{CNAut}$ 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

$$\frac{t : a \to b \in \mathrm{T}}{t : a \to b \in \mathrm{T}'}$$

$$\frac{t : a \to b \in \mathrm{T}' \quad u : b \to c \in \mathrm{T}'}{t; u : a \to c \in \mathrm{T}'}$$

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

subject to the following equational rules

$$\frac{t \in \mathrm{T}'}{\tau ; t = t \; t ; \tau = t} \qquad \frac{t : a \to b \in \mathrm{T}'}{\iota_a ; t = t \; t ; \iota_b = t}$$

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

$$\frac{t \in \mathrm{T}'}{t || \tau = t} \qquad \frac{\iota_a \in \mathrm{T}' \quad \iota_b \in \mathrm{T}'}{\iota_a || \iota_b = \iota_{a \oplus b}}$$

$$\frac{t \in \mathrm{T}' \quad u \in \mathrm{T}'}{t || u = u || t}$$

$$\frac{t \in \mathrm{T}' \quad u \in \mathrm{T}' \quad v \in \mathrm{T}'}{t || (u || v) = (t || u) || v}$$

- **CMon**-object $\mathrm{L} = \langle 1, \mathrm{L}, !, !, !_\iota \rangle$ 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 such that

$$\frac{t \in \mathrm{T}}{lab'(t) = lab(t)}$$

$$\frac{t ; u \in \mathrm{T}'}{lab'(t ; u) = lab'(t) ; lab'(u)} \qquad \frac{t || u \in \mathrm{T}'}{lab'(t || u) = lab'(t) || lab'(u)}$$

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

Let $CNA = \langle \mathsf{G}, \mathsf{L}, lab \rangle$ be a **CNAut**-object and $\mathsf{h} : CNA \to CNA'$ be a **CNAut**-morphism. The functor cn : **CNAut** $\to$ **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

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

  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.

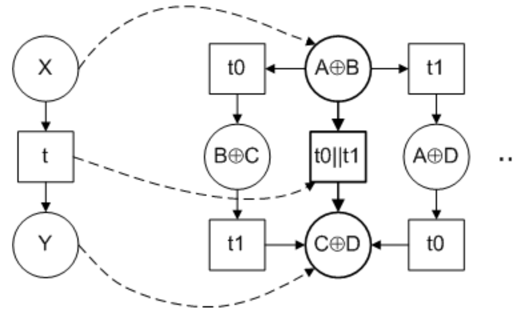

Figure 4: Refinement morphism between nonsequential automata

- 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 $\mathsf{h} = \langle h_G, h_L \rangle$ 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 $h = \left\langle h_{G_V}, h_{G_{T'}}, h_{L_{T'}} \right\rangle : NA \to NA'$ where $h_{G_{T'}}$ and $h_{L_{T'}}$ are the monoid morphisms induced by h_{G_T} and h_{L_T} respectively.

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. Then we 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 [Fig. 4]). Other mappings would also be possible, e.g. $t \mapsto t_0; t_1$ or $t \mapsto t_1; t_0$.

3 Behavioral Diagrams

The Unified Modeling Language (UML) [Rumbaugh et al. 2004] is a graphical language which offers a variety of graphical diagram models 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: activity diagrams, state machine diagrams, interaction diagrams and use case diagrams. From this set of diagrams, this work will concentrate on activity diagrams and state machine diagrams for describing procedural and parallel behavior.

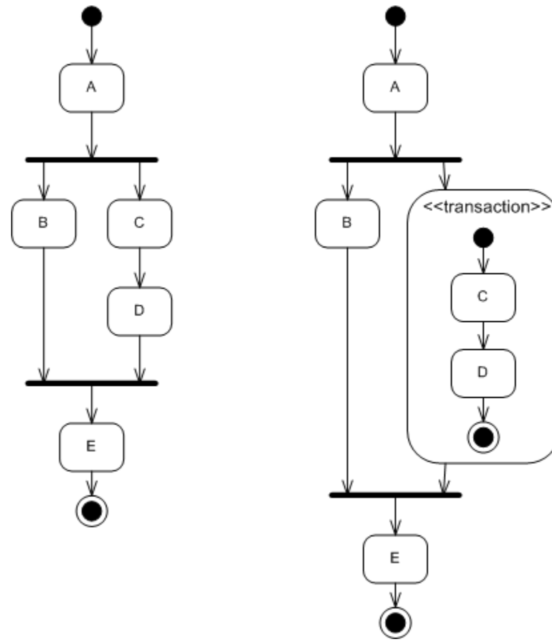

Figure 5: UML activity diagram without (left) and with composite state (right)

The following sections briefly presents the basics of each diagrams and the corresponding semantic mapping to nonsequential automata. In order to simplify the presentation, we chose to describe the mapping by the use of easy to follow examples.

3.1 Activity Diagrams

Activity diagrams are one of the means for describing behavior of systems within UML focused on the flow of control from activity to activity. The most basic node is the action node, which represents an atomic action. Activities are represented by non-atomic composites of sequential or concurrent actions/activities. The control flow is described by special nodes as fork/join for concurrency, decision/merge for alternative paths of execution and initial/final nodes.

Our working example ([Fig. 5] - left) depicts a simple activity diagram for a sequence of operations. Suppose we are interested in defining the sequence of actions “C” and “D” as atomic. To overcome the lack of an atomic activity composite, we introduce a new notation based on the idea of atomic transaction. The new composite activity is decorated with the stereotype “<<transaction>>” as depicted in [Fig. 5] (right).

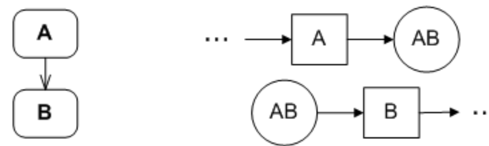

Figure 6: Mapping for sequential composition of action nodes

The semantics for activity diagrams take into account the fact it comprises a token game similar to Petri nets (according to the definition in [OMG 2005b] page 314). So, the semantic mappings from activity diagrams into nonsequential automata are targeted into constructing local transitions for a nonsequential automaton (see [Fig. 13] for local transitions obtained from our working example).

Before applying the mapping we need to transform the activity diagram in such a way each action node has only one incoming/outgoing edge. We do this as a precaution to avoid misinterpretation of activities control flow because implicit merging/joining of edges has changed from previous UML versions [Bock 2003]. Previous versions were based on implicit merging of edges, and the current definition applies an implicit join.

Each action node consumes/produces control tokens as the steps of computation progress through the activity diagram. For nonsequential automata, this semantics belongs to transitions. Thus, each action node corresponds to a nonsequential automaton transition, whose origin denotes the necessary tokens for its firing (preconditions), and whose destiny denotes the tokens produced after its firing (postconditions), taking into account the different kind of nodes from its incoming/outgoing edges.

In [Fig. 6], an outgoing edge from action node and the corresponding incoming edge in the target action node represent sequential composition by sharing a nonsequential automaton state. The example shows a fragment of an activity diagram ([Fig. 6] left) with two action nodes “A” and “B” which are sequentially composed by a direct edge (which we are going to name “AB”). As the UML description implies whenever action node “A” completes its execution, a token is put in its outgoing edge; thus, in terms of nonsequential automata, the state “AB” is the target of the local transition “A” ([Fig. 6] right). Accordingly, an action node is enabled whenever there is a token in its incoming edge, and so the edge “AB” were represented as the source state “AB” of the nonsequential atomaton transition “B” ([Fig. 6] right).

The set of initial states of the nonsequential automaton are the ones marked by initial nodes in the activity diagram (see, for example, the state “IA” in [Fig. 7]). On the other hand, final states are the ones related to activity final nodes in the activity diagram (see, for example, the state “AF” in [Fig. 8]).

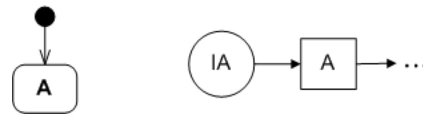

Figure 7: Mapping for initial nodes

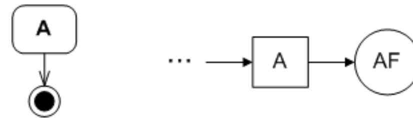

Figure 8: Mapping for final nodes

As pointed previously, edges and control nodes are mapped to a consistent set of nonsequential automaton states according to its purpose. Next we define the mappings for parallel composition and merge of flows.

Fork/join nodes in figures [Fig. 9] and [Fig. 10] demand the use of nonsequential automaton structured states in order to represent concurrent actions. The idea is to use the monoidal operator on nonsequential automaton states in order to get the UML concept of concurrently enabled edges by the presence of multiple tokens. The fork node produces a structured state with all tokens necessary for its outgoing edges, representing the duplicate of tokens across the outgoing edges of the activity diagram. In the example, the activity diagram control node ([Fig. 9] left) is to be interpreted as an hyperedge with one single action node ("A") as source and multiple action nodes ("B" and "C") as target, in such a way the compound edge is represented by nonsequential automaton states "AB" and "AC" ([Fig. 9] right). Similarly, the join node synchronize different control flows through a structured state aggregating each incoming edge.

For decision/merge nodes (see [Fig. 11] and [Fig. 12]), we took an alternative approach. Analogous to fork/join node, the control node is seen as an hyperedge, but it induces several local nonsequential automata transitions, one for each

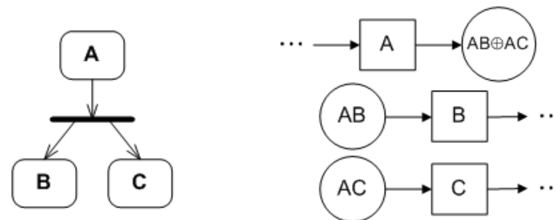

Figure 9: Mapping for fork nodes

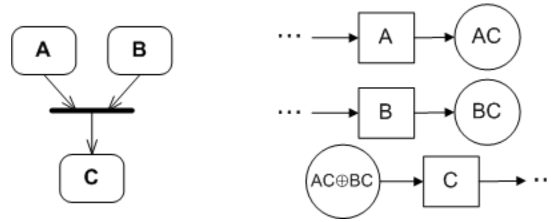

Figure 10: Mapping for join nodes

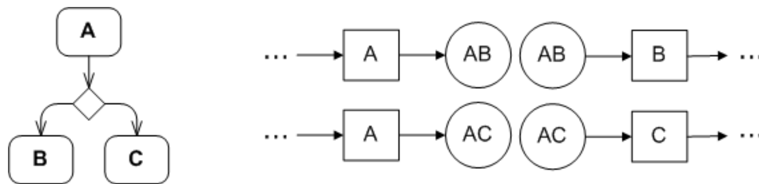

Figure 11: Mapping for decision nodes

alternative path. For the activity diagram fragment in [Fig. 11](left), the edges attached to the decision node are represented by nonsequential automata states "AB" and "AC", and the action node gives rise to labels "A", "B" and "C". We have not used the term transition because the mapping of outgoing edges was based on the idea of reducing the choice (according to guards attached to these edges) to nondeterminism in the corresponding nonsequential automaton by using different transitions labeled with the same label (this is the case of the two transitions whose targets are states "AB" and "AC" labled with "A" in [Fig. 11] right).

The central core of the composite transaction node makes use of nonsequential automata refinement. The source automaton corresponds to the basic translation using the previous mappings, where the composite node is viewed as only one nonsequential automaton transition. The target automaton corresponds to the translation taking into account the subactivity nodes of the composite.

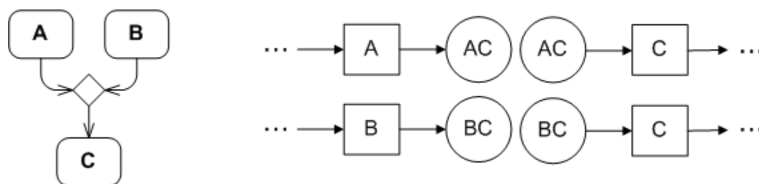

Figure 12: Mapping for merge nodes

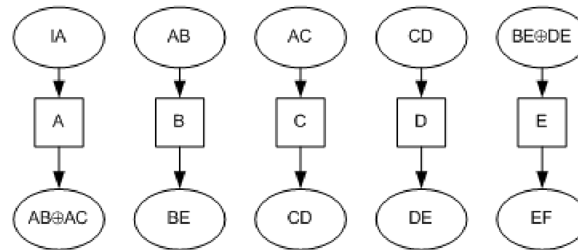

Figure 13: Local transitions for the nonsequential automaton

The refinement then maps the more abstract transition into the concrete implementation of the transaction obtained via the computational closure of the target automaton. Actually, the source automaton is obtained by a sequence of functorial operations (relabeling and restriction) over the target automaton (see [Menezes et al. 1996] and [Menezes et al. 1998] for definition of these operations). [Fig. 14] partially depicts the automata (based on local transitions from [Fig. 13]) and refinement (dashed arrows for transition refinement, the mapping of states were not shown) for our working example of activity diagrams. Notice it does explicit all possible computational paths, including the transaction state (labeled "T" in the source automaton) represented by the atomic sequential composition "C;D". Also, the fork of control flow in the activity diagram is correctly depicted by the independence square "B||T" mapped to "B||(C;D)".

3.2 State Machine Diagrams

State machine diagrams are one of the means for describing behavior of systems within UML focused on a number of states an object may hold during its lifetime. It is one of the most intuitive diagrams because its foundations on automata, Mealy and Moore machines are well known.

A state machine is a graph of states and transitions. Transitions connect different states and are fired by triggering events. The response to events may include the execution of an effect (an action or activity) and a change to a new state. The most basic set of nodes are the state node and initial/final state nodes, representing the basic units of control for this diagram. The transition flow between states, specified by transitions, may be modified by special nodes (called pseudo-states) such as fork/join for concurrency, junction for sequential composition of effects and choice for alternative paths. Also, composite states are present as a mean to simplify the reuse of transitions and introduce the possibility of concurrency among different states.

Here we are interested in using state machines to describe sequence of observable effects/activities of a system. We are not focusing on the sequence of valid

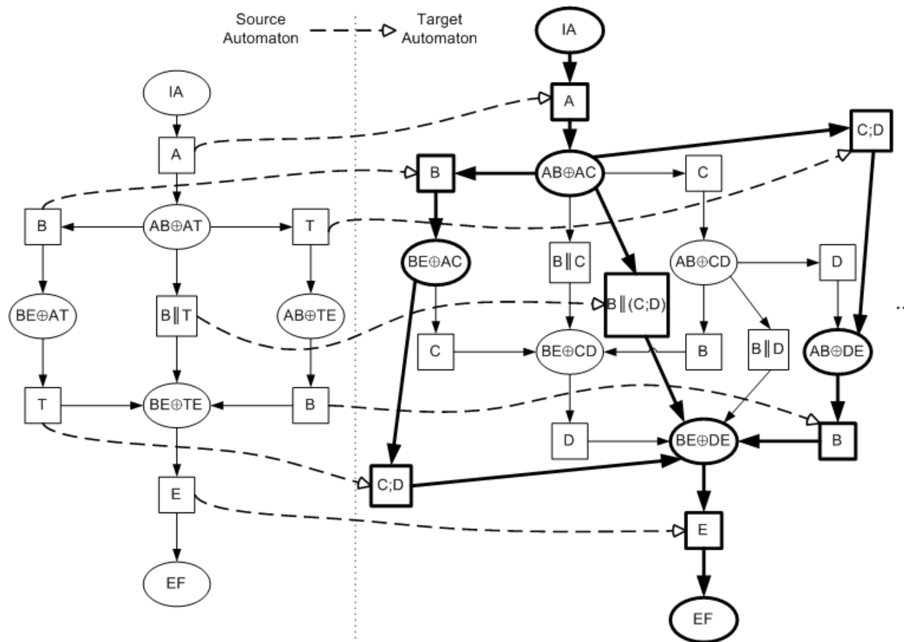

Figure 14: Nonsequential automata refinement for activity diagram with atomic composite in [Fig. 5]

states or the sequence of events that trigger transitions between states. Thus, the state machine view we are employing is related to descriptions of dynamic behavior of uses cases, collaborations and methods as pointed in [Rumbaugh et al. 2004] and, for these objects, a state represents a computation step in its execution.

Our working example ([Fig. 15] - left) depicts a simple state machine diagram in which the flow between states are by completion transitions. This diagram may be seen as specifying a sequence of actions ("A", "B" and "C") much like an activity diagram, except it includes the states in which an action is valid and the resulting state. Suppose we are interested in defining some kind of transactional composite state in which compounded state transitions cannot be interrupted by transitions crossing boundaries to any state outside the composite "E1", resulting an atomic sequence of actions "A" and "B". We introduce a new notation to regions inside composite states decorated with the stereotype "<<transaction>>" ([Fig. 15] - right).

When defining the semantic mapping, we followed the premise of compatibility between the state machine view and the activity diagram view. This is important because states may carry a notion of ongoing activity and its behavior should be compatible to activities expressed in activity diagrams. This

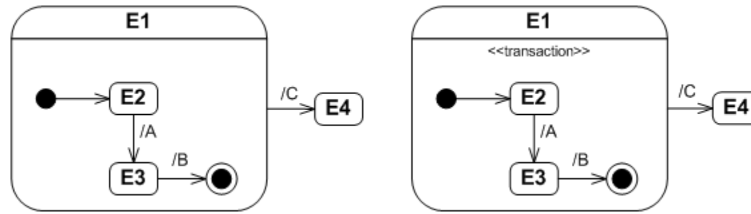

Figure 15: UML state machine diagram without (left) and with transactional composite (right)

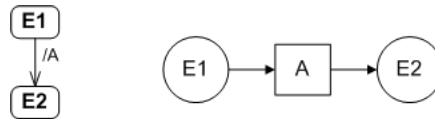

Figure 16: Mapping for basic states and transitions

notion of compatibility may be formally described by an equivalence relation on nonsequential automata.

The semantic mappings from state machine diagrams into nonsequential automata are targeted into constructing local transitions for a nonsequential automaton (see [Fig. 24] for local transitions obtained from our working example).

The basic mapping is such that states from the UML diagram are mapped to nonsequential automata states, and transitions labeled with effects are mapped to nonsequential automata transitions. Following [van der Aalst 2000], our semantics for state machines has abstracted away events for communicating with the system environment. In this paper, only completion events are being considered. Completion events are implicitly associated to transitions that lack an explicit trigger event. Thus, the notion of completion is represented by nonsequential states which are "consumed" much like the token game in Petri nets, and the effect appear as a transition in the corresponding automaton. [Fig. 16] shows local transitions for nonsequential automata obtained from simple state and basic transitions.

Initial states (see [Fig. 17]) are the ones marked with initial pseudo-states in the state machine diagram, and final states (see [Fig. 18]) are the ones related to final states in the diagram. In both cases, the mapping is analogous to basic states and transitions, and the resulting states will play an important role in composite states (defined next). The outermost initial state in the state machine diagram will be marked as the automaton initial state (the same for the final state).

For composite states that have been decomposed into regions (either nonorthogonal with only one region, or orthogonal with two or more concurrent re-

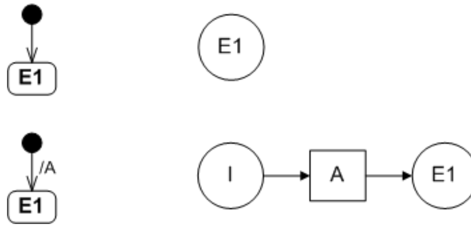

Figure 17: Mapping for initial states

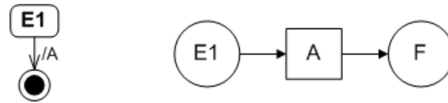

Figure 18: Mapping for final states

gions), the chosen domain (and mapping) for state machines bring as side effect an abstraction from the state hierarchy implied from composites. What we get is a flat view of the machine where implicit transitions from composites have become explicit for every compounded state. Although not presented here, a flattening should first be applied before the mappings because of implicit transitions generated by composite states much like the procedure described in [Eshuis and Wieringa 2003]. In [Fig. 19] (left), concurrent orthogonal regions are entered explicitly (by applying fork pseudo-states) and implicitly (by using transitions into the enclosing composite state). Notice the semantics makes use of nonsequential automata structured states from representing distributed concurrency. The idea for the mapping is, again, to manipulate the transition as an hyperedge with one source and several target. For join pseudo-nodes and completion transitions from composite states (see [Fig. 20]) the mapping is analogous. Although both constructions were presented, we advocate the use of implicit transitions into composite states, once this construction is compositional and avoids crossing state boundaries.

The central core of the new composite transaction state makes use of nonsequential automata refinement, following the same ideas developed for activity diagrams. The source automaton corresponds to the basic translation using the previous mappings, where the composite state induces only one nonsequential automaton transition. The target automaton corresponds to the translation taking into account the substates of the composite. In [Fig. 21] the composite transaction involved in the refinement mapping were shown in dashed pattern, representing the sequential flow of effects “A” and “B” (this transition is the result of the calculus of the nonsequential automaton closure and will be the

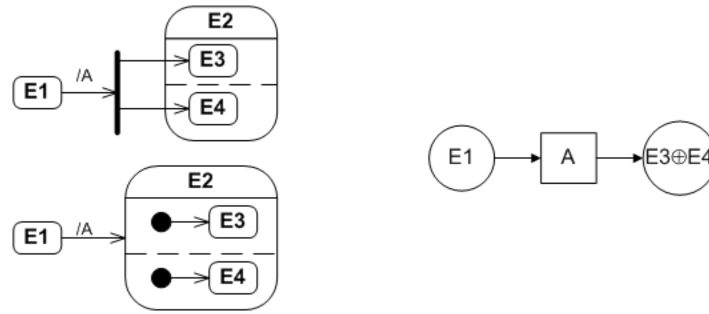

Figure 19: Mapping for composite states and fork pseudo-states

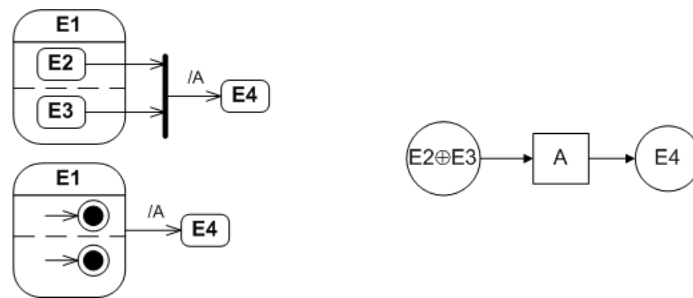

Figure 20: Mapping for composite states and join pseudo-states

target in the refinement morphism). The figure also highlights the fact the composite is now atomic and transitions that cross boundaries are not permitted (in other words, the only exit point is the final state, which acts as a commit).

Besides the explicit transaction composite states, an intrinsic notion of atomic composition can be found in the "run-to-completion" mechanism of state machine diagrams and also in composition of steps in some pseudo-states. Run-to-completion may be defined as "a transition or series of actions that must be

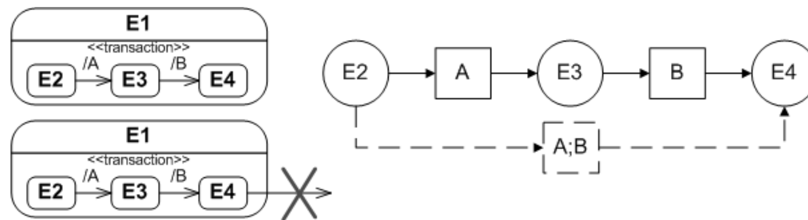

Figure 21: Mapping for transaction region in composite state

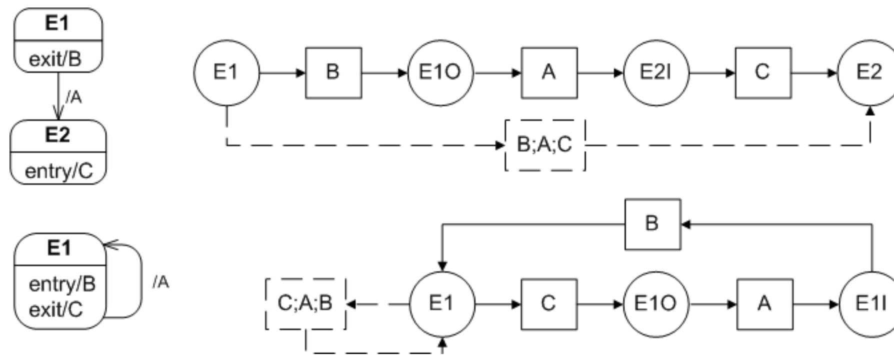

Figure 22: Mapping for entry and exit activities

completed in its entirety" [Rumbaugh et al. 2004]. Shortly, the firing of a transition is accomplished by a sequence of steps: the current state is exited and the exit activity of the state is executed; then the effect of the transition is executed; finally the entry activity of the state being entered is executed. This behavior is depicted in [Fig. 22]. Notice the mapping introduce "dummy" states (labeled with "I" for input and "O" for output) wherever there is states with entry and/or exit activities. Again, the dashed pattern were used to highlight the atomic compositions to be used in the refinement.

Pseudo-states that imply run-to-completion include junction states and choice states. Junction and choice states are vertices that are used to chain together multiple transitions between states. In [Fig. 23] compound transitions are shown for junctions. In the mapping, each state generate a nonsequential automaton state and each junction pseudo-state generate a "dummy" nonsequential automaton state just for the sake of building the composite paths. The resulting nonsequential automaton explicits the atomic sequential composition of alternative paths to be taken. Please notice that in this version we are not dealing with variables in the state space and consequently the mapping for choice pseudo-states will be analogous.

Going back to our working example, we are now able to build the local transitions for the nonsequential automaton (see [Fig. 24] for the set of transitions). [Fig. 25] partially depicts the automata and refinement (dashed arrows) for our working example of state machine diagram. Notice it does explicit the computational path "A;B" for the atomic sequential composition from transaction state "E1"(labeled "TE1" in the source automaton).

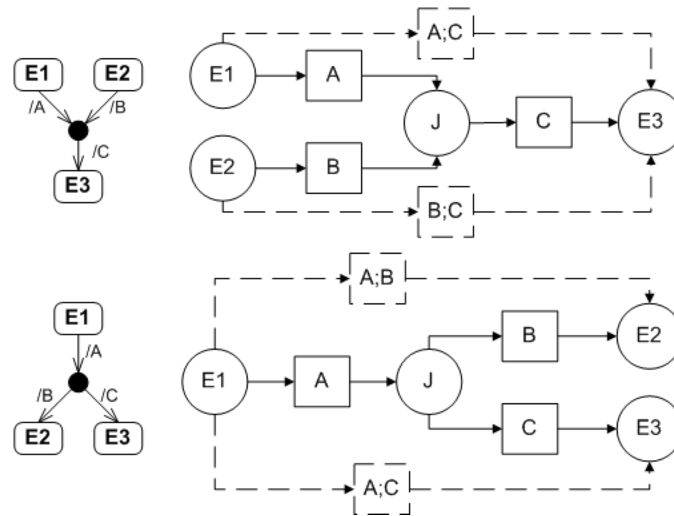

Figure 23: Mapping for junction pseudo-states

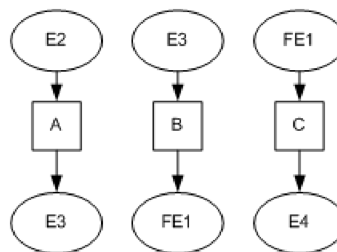

Figure 24: Local transitions for the nonsequential automaton

4 Other Approaches

Some approaches to translating UML diagrams into formal models based on Petri nets are closely related to this work. For example, [Gehrke et al. 1998] describes a formal translation of activity and collaboration diagrams into place/transition Petri nets and [Eshuis and Wieringa 2003] compares different proposals for the semantics based on Petri nets targeting workflow models based on activity diagrams.

Although such works have succeeded in defining semantics for activity diagrams, one further important question remained open - the need for models that include the diagonal compositionality requirement as stated by Gorrieri [Gorrieri 1990]. Therefore, we should be able to further define levels of abstractions of systems before or after a synchronization/refinement composition in

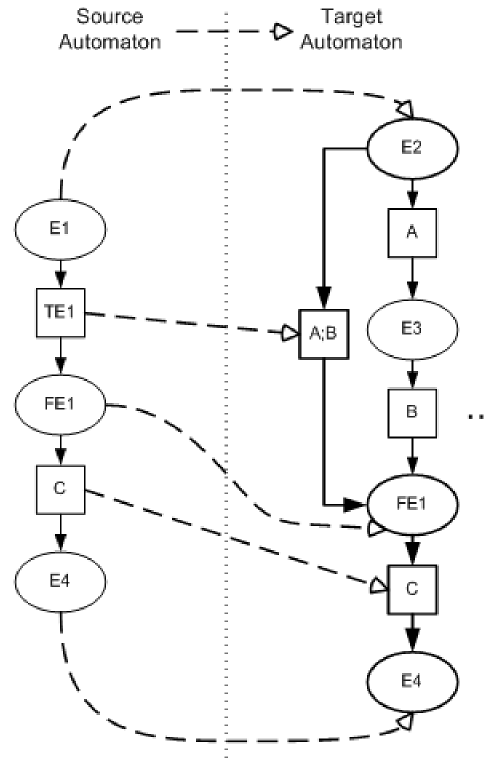

Figure 25: Nonsequential automata refinement for state machine diagram with atomic composite in [Fig. 15]

order to obtain the same resulting system. Here, again, we are in a delicate situation because, as shown in [Menezes and Costa 1996], most Petri net models do not imply the diagonal compositionality requirement. Our goal, thus, have been to apply a semantic model for describing compositional constructs that could cope with the diagonal compositionality requirement, and nonsequential automata have shown this desired property.

Regarding the semantic domain, Zero-Safe Nets [Bruni and Montanari 1997] [Bruni and Montanari 2001] are an approach to the modeling of transactions built on top of ordinary place/transition Petri nets extended with a mechanism for transition synchronization. The constructions for computational closure and refinement are very similar to nonsequential automata and are also based on category theory. In this model, zero-places are used for coordinating the atomic execution of several transitions, which, from an abstract point of view, will appear as synchronized (or belonging to a transaction). In fact, the relation between nonsequential automata and zero-safe nets must be further investigated

following the approach by [Winskel and Nielsen 1995] where a scene for a formal classification of models for concurrency was set.

The main differences between this proposal and related works may be summarized as follows: we are based on the UML 2.0 specification, in which activity diagrams have been decoupled from state diagrams; the applied semantic domain is compositional, in contrast to domains based on Petri nets or statecharts semantics; we are dealing with mechanisms for atomic compositions and not just non-atomic composites.

5 Concluding Remarks

Transactions are an important part of today systems and they deserve a first class mechanism in modeling languages, especially UML. Following that premise, this work presented an extension to UML diagrams centered on constructions for defining atomic composition of actions/activities/operations. The use of nonsequential automata specifies the semantics unambiguously and enables an elegant definition for atomicity. Regarding previous works [Machado and Menezes 2004], this is the first time we present the ideas for activity and state machine diagrams in a compatible way. Also, this paper is an extended version from the paper [Machado and Menezes 2006].

In this paper we have not dealt with event handling. Generally speaking, for Petri net related models, events may be modeled as tokens or transitions with different consequences on the resulting behavior (see [Eshuis 2002] for a discussion on both alternatives). We are currently working on adding events into the semantic mapping.

Also, for a complete presentation of atomic compositions we are working on the definition of a UML profile based on the semantics presented in this paper.

References

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

[Bock 2003] Bock, C.: "UML 2 Activity and Action Models"; Journal of Object Technology, 2, 4(2003), 43-53.

[Bruni and Montanari 1997] Bruni, R., Montanari, U.: "Zero-Safe Nets, or Transition Synchronization Made Simple"; Proc. EXPRESS'97, Electronic Notes in Theoretical Computer Science 7, Elsevier (1997).

[Bruni and Montanari 2001] Bruni, R., Montanari, U.: "Transactions and Zero-Safe Nets"; Unifying Petri Nets, Lecture Notes in Computer Science 2128, Springer, Berlin (2001), 380-426.

[Bruni and Montanari 2004] Bruni, R., Montanari, U.: "Concurrent Models for Linda with Transactions"; Mathematical Structures in Computer Science, 14, 3(2004), 421-468.

[Eshuis 2002] Eshuis, R.:"Semantics and Verification of UML Activity Diagrams for Workflow Modelling"; PhD thesis, University of Twente, Enschede (2002).

[Eshuis and Wieringa 2003] Eshuis, R., Wieringa, R.:"Comparing Petri net and activity diagram variants for workflow modelling - a quest for reactive Petri nets"; Petri Net Technology for Communication Based Systems, Lecture Notes in Computer Science 2472, Springer, Berlin (2003), 321-351.

[Gehrke et al. 1998] Gehrke, T., Goltz, U., Wehrheim, H.:"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).

[Gorrieri 1990] Gorrieri, R.:"Refinement, Atomicity and Transactions for Process Description Language"; PhD thesis, University of Pisa, Pisa (1990).

[Ullman and Widom 2002] Ullman, J. D., Widom, J.:"A First Course in Database Systems"; 2nd ed., Prentice-Hall, New Jersey (2002).

[Machado and Menezes 2004] Machado, J. P., Menezes, P. B.:"Modeling Transactions in UML Activity Diagrams via Nonsequential Automata"; Actas de la XXX Conferencia Latinoamericana de Informatica, CLEI (2004), 543-553.

[Machado and Menezes 2006] Machado, J. P., Menezes, P. B.:"Defining Atomic Composition for UML Behavioral Diagrams"; Proc. X Brazilian Symposium on Programming Languages, IME/UFMG (2006), 277-290.

[Menezes and Costa 1996] Menezes, P. B., Costa, J. F.:"Synchronization in Petri Nets"; Fundamenta Informaticae, 26, 1(1996), 11-22.

[Menezes et al. 1996] Menezes, P. B., Costa, J. F., Sernadas, A. S.:"Refinement Mapping for General (Discrete Event) System Theory"; Proc. 5th International Conference on Computer Aided Systems Theory and Technology, Lecture Notes in Computer Science 1030, Springer, Berlin (1996), 103-116.

[Menezes et al. 1998] Menezes, P. B., Sernadas, A. S., Costa, J. F.:"Nonsequential Automata Semantics for Concurrent, Object-based Language"; Proc. 2nd US-Brazil Joint Workshops on the Formal Foundations of Software Systems, Electronic Notes in Theoretical Computer Science 14, Elsevier (1998).

[Meseguer and Montanari 1990] Meseguer, J., Montanari, U.:"Petri Nets are Monoids"; Information and Computation, 88, 2(1990), 105-155.

[OMG 2004] OMG:"UML 2.0 Infrastructure"; Technical Report ptc/04-10-14, Object Management Group (2004) `http://www.omg.org`

[OMG 2005a] OMG:"UML 2.0 OCL Specification"; Technical Report ptc/05-06-06, Object Management Group (2005) `http://www.omg.org`

[OMG 2005b] OMG:"UML 2.0 Superstructure"; Technical Report formal/05-07-04, Object Management Group (2005) `http://www.omg.org`

[Reisig 1985] Reisig, W.:"Petri Nets: an introduction"; EATCS Monographs on Theoretical Computer Science 4, Springer, Berlin (1985).

[Rumbaugh et al. 2004] Rumbaugh, J., Jacobson, I., Booch, G.:"The Unified Modeling Language Reference Manual"; 2nd ed., Addison-Wesley, Boston(2004).

[Tanenbaum 2001] Tanenbaum, A. S.:"Modern Operating Systems"; Prentice-Hall, Upper Saddle River(2001).

[van der Aalst 2000] van der Aalst, W. M. P.:"Workflow Verification: Findind Control-Flow Errors Using Petri-Net-Based Techniques"; Business Process Management, Lecture Notes in Computer Science 1806, Springer, Berlin (2000), 161-183.

[Winskel and Nielsen 1995] Winskel, G., Nielsen, M.:"Models for Concurrency"; Handbook of Logic in Computer Science, Volume 4, Oxford University Press (1995), 1-148.