Composing Transitions into Transactions in UML Diagrams

Júlio Pereira Machado[1] and Paulo Blauth Menezes[2]

[1] Faculdade de Informática, Pontifícia Universidade,
Católica do Rio Grande do Sul, Porto Alegre, RS, Brasil
juliopm@inf.pucrs.br
[2] Instituto de Informática, Universidade Federal do Rio Grande do Sul,
Porto Alegre, RS, Brasil
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 may be conceptually refined into transactions. Nevertheless, the Unified Modeling Language seems to lack compositional constructs for defining atomic actions/activities/operations. We discuss proper extensions for UML behavioral diagrams that are able to cope with the concept of transaction. Transactions are formally defined through a special morphism between automata in a semantic domain called Nonsequential Automata.

1 Introduction

The Unified Modeling Language (UML) [1] 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, just no name a few. In this sense, when modeling a computational process, we need means of composing subactivities both in a non atomic or atomic way. Nevertheless, the UML seems to lack compositional constructs for defining atomic actions/activities/operations.

In this work[1], 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. We remark that in our setting the term

[1] This work was partially supported by CTXML/Microsoft/PUCRS, CNPq (Projects HoVer-CAM, GRAPHIT, E-Automaton) and FINEP/CNPq (Project Hyper-Seed) in Brazil.

R. Moreno Díaz et al. (Eds.): EUROCAST 2005, LNCS 3643, pp. 50–55, 2005.

"transaction" denotes a certain activity of the system that might be composed by many, possibly concurrent, subactivities. Moreover, we require this composition of activities to be considered atomic.

2 Nonsequential Automata

Nonsequential Automata [2,3] 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 [4] 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. In the next definitions **CMon** denotes the category of commutative monoids and $k \in \{0, 1\}$ (for simplicity, we omit that $k \in \{0, 1\}$).

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 transitions and *lab* is the labeling morphism associating a label to each transition.

According to the definition, the automaton 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. 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 figure 1 (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.

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 figure 1 (added transitions were drawn with a dotted pattern).

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 one (nc) basically enriches an automaton with a composition operation

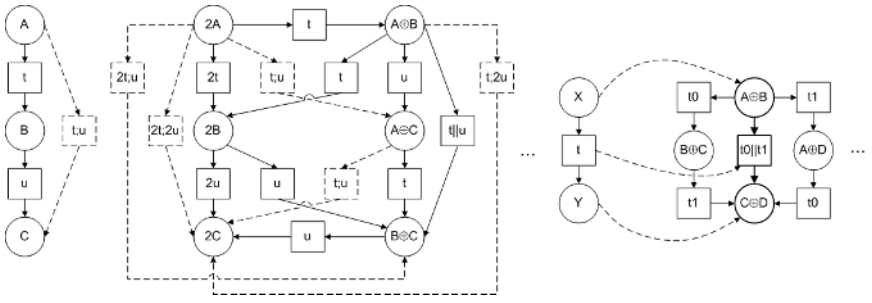

Fig. 1. Nonsequential automaton with computational closure (left) and refinement morphism (right)

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 $\varphi : NA \to \mathsf{tc}NA'$. Both functors were presented in [5] and due to limitations are not being rephrased here. The transitive closure functor is $\mathsf{tc} = \mathsf{cn} \circ \mathsf{nc} : NAut \to NAut$. 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. For the last step 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 figure 1 - right). 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.

3 Transactions in UML Diagrams

In order to correctly introduce the notion of transactions, we need to analyze the UML official documentation. The UML specification by OMG [6,7] posses 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) further defines constraints over models so they can be considered well-formed.

In our approach, a basic set of metamodel elements is selected. The idea is to focus only on constructs 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 nonatomic ones and are further decorated with proper stereotypes. Also, new OCL expressions are built to define the new constraints over atomic compositions. One example of a new constraint for the atomic composite state in sate diagrams is the one that does not allow internal states to be interrupted by explicit external events. Finally, the

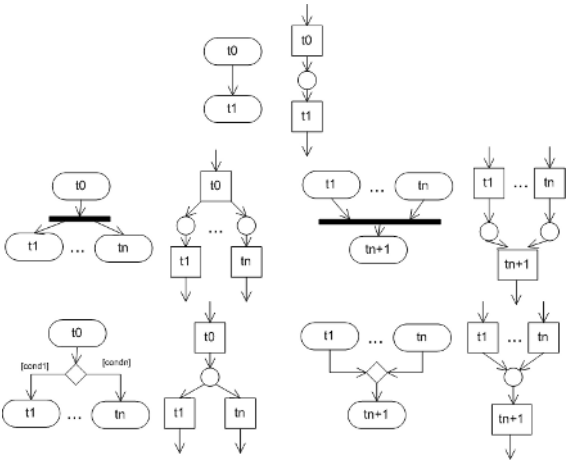

Fig. 2. Semantic mapping examples

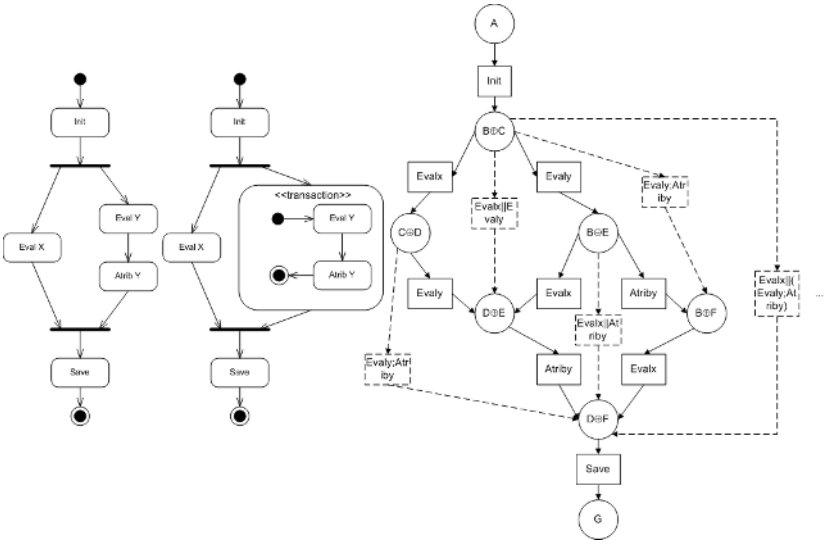

Fig. 3. UML activity diagram without (left) and with composite state (center) , and nonsequential automaton for its semantics (right)

well-formed models are mapped to nonsequential automata, thus formally defining its semantics. We define one new atomic composite for activity diagrams, state diagrams and sequence diagrams, but due to space limitations, our discussion and working example are based only on activity diagrams. A similar approach for state and sequence diagrams have been employed.

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 rep-

resented by nonatomic 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 (figure 3 - left) depicts a simple activity diagram for a sequence of operations in a pseudo programing language. Suppose we are interested in defining the sequential sequence of actions "Eval Y" and "Attrib Y" 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 figure 3 (center).

The semantics for activity diagrams take into account the fact it comprises a token game similar to Petri nets. So, the semantic mappings from activity diagrams into nonsequential automata are targeted into constructing local transitions for a nonsequential automaton. 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. Each action node consumes/produces control tokens as the steps of computation progresses 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, and whose destiny denotes the tokens produced after its firing. Edges and control nodes are mapped to a consistent set of nonsequential automaton states according to its purpose. Figure 2 depicts the resulting states and transitions in a nonsequential automaton.

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. 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 3 partially depicts the target nonsequential automaton for our working example of activity diagrams. Notice it explicits all possible computational paths, including the transaction state represented by the atomic sequential composition $Evaly; Atriby$.

4 Concluding Remarks

We believe 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. Its semantics were defined as nonsequential automata refinement morphisms.

Other approaches to translating UML diagrams into formal models have been based on Petri nets [8]. For example, [9] describes a formal translation of activity

and collaboration diagrams into place/transition Petri nets and [10] compares different proposals for the semantics based on Petri nets. Also, other works have used formal methods to verify the behavior of UML specifications [11,12]. 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 nonatomic composites.

References

1. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. 2 edn. Addison-Wesley (2004)
2. Menezes, P.B., Costa, J.F.: Compositional reification of concurrent systems. Journal of the Brazilian Computer Society **2** (1995) 50–67
3. Menezes, P.B., Costa, J.F., Sernadas, A.S.: 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., Springer-Verlag (1996) 103–116
4. Meseguer, J., Montanari, U.: Petri nets are monoids. Information and Computation **88** (1990) 105–155
5. Machado, J.P., Menezes, P.B.: Modeling transactions in uml activity diagrams via nonsequential automata. In: Actas de la XXX Conferencia Latinoamericana de Informatica, CLEI (2004) 543–553
6. OMG: Uml 2.0 superstructure ftf. Technical Report ptc/04-10-02, Object Management Group (2004)
7. OMG: Uml 2.0 infrastructure final adopted specifcation. Technical Report ptc/03-09-15, Object Management Group (2003)
8. Reisig, W.: Petri Nets: an introduction. Volume 4 of Eatcs Monographs on Theoretical Computer Science. Springer-Verlag (1985)
9. 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)
10. Eshuis, R., Wieringa, R.: 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., Springer-Verlag (2003) 321–351
11. Shen, W., Compton, K., Huggins, J.: A validation method for uml model based on abstract state machines. In: Proceedings of EUROCAST. (2001) 220 – 223
12. Knapp, A., Merz, S.: Model checking and code generation for uml state machines and collaborations. In: Proceedings of 5th Workshop on Tools for System Design and Verification. (2002) 59 – 64