BEHAVIOR ASSEMBLY AND COMPOSITION OF USE CASES – UML 2.0 PERSPECTIVE

Vladimir Mencl[1], Frantisek Plasil[1,2], Jiri Adamek[1]

[1]Charles University, Faculty of Mathematics and Physics
Department of Software Engineering, Distributed Systems Research Group
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
{mencl,plasil,adamek}@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz,
phone: +420 2 2191 4266, fax: +420 2 2191 4323

[2]Academy of Sciences of the Czech Republic
Institute of Computer Science
plasil@cs.cas.cz, http://www.cs.cas.cz

ABSTRACT

Designing components and composing them into an architecture inherently involves describing their behavior. The classical software engineering approach to specifying requirements for large-scale components is to start with use cases. However, employing use cases to component architectures triggers the need of (i) assembling the behavior specified by several use cases, (ii) composing the behavior of communicating entities, and (iii) reasoning on consistency of the composed behavior. Applying a modeling language, such as UML, while dealing with these issues is desirable.

Based on the composite structures framework, the emerging standard UML 2.0 defines a hierarchical component model; here, behavior of components may be specified with use cases. UML 2.0 provides four behavior specification mechanisms. The key goal of this paper is to evaluate whether and how these behavior specification mechanisms address the issues above. We show that only Interactions implicitly allow for addressing all of them.

KEYWORDS
Software Methodologies, UML, Use Cases, Formal Methods

1. Introduction

Designing components and composing them into software architectures includes the need of specifying their behavior. In current component models, component interfaces are often described in terms of signatures of operations. This allows to reason on correctness of component composition at the level of syntactical type correspondence. However, when considering composition of components obtained from independent sources (e.g., different vendors and/or development teams), such a specification is clearly not sufficient. A specification of the component's behavior is essential, describing the correct usage of the services provided by the component, the intended use of the services required by the component, as well as the dependence among interactions occurring on the individual interfaces of the component.

The emerging standard UML 2.0 [1] introduces new constructs StructuredClassifier and Encapsulated-Classifier for capturing hierarchical composition. Based on these enhancements, UML 2.0 introduces a hierarchical component model. In this model, the communication of a Component is encapsulated in a set of provided and required interfaces. Thus, a UML 2.0 Component corresponds to the concept of a component considered by the architecture description languages (ADLs) such as Darwin [2], SOFA [3] and Fractal [4].

Besides the structured classifiers, UML 2.0 also introduces a new generic behavior model. The Class metaclass is extended with the ability to own a Behavior and to specify behavioral features (such as operations). There are four behavior specification mechanisms (subtypes of Behavior) to provide concrete semantics of the generic behavior model; the subclasses provided are Interaction, Activity, StateMachine and its specialization ProtocolStateMachine. Compared to UML 1.x, StateMachines have been significantly reworked; now, they support inheritance (of state machines) and nested states. Of the two types of state machines considered, behavioral State Machines are intended for implementation specification and Protocol State Machines for usage specification.

UML 2.0 provides support for specifying software architectures based on a nested component model; at all levels of nesting, the components may be amended with behavior specifications. In particular, behavior of a component may be specified with use cases, possibly employing one of the behavior specification mechanisms considered. As the recommended software engineering approach is to apply use cases at the beginning of the design process [5, 6], this creates a particular interest in evaluating how these behavior specification mechanisms support behavior assembly and composition.

1.1. Use Case Basics

In principle a use case [7, 8, 9] specifies a set of scenarios determining how a group of entities (software

components, human beings, business entities) communicate and internally perform to achieve a certain goal. There is a variety of behavior specification mechanisms available for use cases based on formal methods [10, 11, 12]; for an overview of the approaches, we refer the reader to [13]. As to UML, it provides modeling constructs for use cases; however, the constructs target mainly the relations among use cases and between a use case and the entities involved in it. While UML 1.5 [9] leaves the actual way the behavior of a use case is specified open, UML 2.0 [1] is more specific as it provides several behavior specification mechanisms such as Interactions and State Machines which may be employed in a use case.

A use case is written from the perspective of one of the entities – *SuD* (system under discussion), specifying what actions SuD executes in a particular communication (*scenario*) with other entities (*actors*) to achieve a specific goal. Typically, each use case specifies only a part of the scenarios associated with SuD.
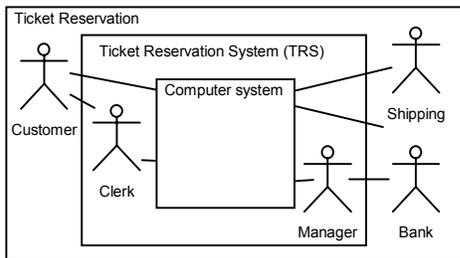

Figure 1: Scope diagram of Ticket Reservation

Entities can be nested as illustrated on the scope diagram in Fig. 1, which also shows the communication links among them (by convention, the parent entity is called *scope*). The entities correspond to future components; in Fig. 2, we sketch a possible component design of the system. We will use these envisioned components to demonstrate the UML 2.0 behavior specification mechanisms. For now, to simply illustrate the issues addressed by this paper, we use fragments of use cases employing the plain English textual notation (fig. 3), modeling communication of the entities Ticket Reservation System (TRS) and Bank.

Apparently, in addition to "Pay for a Ticket" and "Deliver Ticket", there would be more use cases of TRS – all of them together specifying the behavior of TRS. The scenarios specified by these individual use cases may be required to be performed in a sequence, concurrently, etc.; in general we call the process of "putting use cases together"
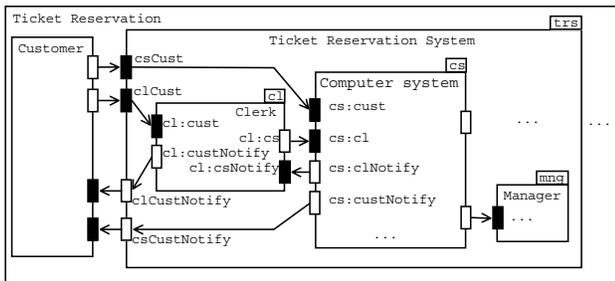

Figure 2: Component diagram for the Ticket Reservation case study

Use Case: TRS1 Pay for a Ticket	Use Case: TRS2 Deliver Ticket	Use Case: B1 Validate Payment
Scope: Ticket Reservation	Scope: Ticket Reservation	Scope: Ticket Reservation
SuD: Ticket Reservation System	SuD: Ticket Reservation System	SuD: Bank
Actors: **Customer**, **Bank**	Actors: **Shipping**, **Customer**	Actors: **Ticket Reservation System**
Main success scenario specification:	**Main success scenario specification:**	**Main success scenario specification:**
1. Customer selects to buy the reserved tickets.	1. System looks up the Customer's address	1. The bank receives a payment validation request
2. System validates the ticket reservation.	2. System validates the address with the Shipping service	2. Bank validates the payment and sends response.
3. System informs the Customer about the amount to be paid.	3. System send the tickets via the Shipping service	**Extensions:**
4. Customer enters payment information.	4. System sends the customer a shipment tracking id	2a Validation fails
5. System validates the payment information with the Bank	...	2a1 Bank sends notification that validation failed
...		2a2 Use case aborts
		...

Figure 3: Fragments of use cases of the entities Ticket Reservation System and Bank

behavior assembly which yields the *assembled behavior* of an SuD. In this view, the *use case model* of TRS is determined by all its use cases and by the way they are combined.

In a similar vein, for composed entities the question is whether it is possible to obtain the behavior of the containing entity by *composition* of the assembled behaviors of the nested entities. The *composed behavior* has to capture the internal communication among the nested entities, as well as those of their activities that are visible outside as the behavior of the containing entity. In the example above, TRS communicates with Bank in such a way that the steps 1 and 2 of B1 correspond to the step 5 of TRS1. Except for this special case (synchronization), in which the complementary communication actions become an internal activity, the Bank and TRS operate in parallel and their activities interleave. The communication of Manager and Bank reaching outside of TRS gives an example of an externally visible communication.

1.2. Goal and Structure of the Paper

Obviously, in order to utilize all the potential of the use case idea, there is the need to precisely define the semantics of the behavior specification mechanism employed in a use case specification. The emerging standard UML 2.0 features mechanisms to capture relations among use cases as well as to specify behavior of a use case. Therefore, an interesting problem is to which extent these UML mechanisms support behavior assembly, behavior composition, and consistency reasoning. Needless to say, behavior composition and consistency reasoning are extremely important for software component and architecture design in itself. Addressing this problem is the key goal of this paper.

The paper is organized as follows: Section 2 provides a generic formal model of use cases which captures all the important use case-related abstraction. Section 3 addresses the key goal of the paper by analyzing the UML 2.0

behavior specification mechanisms and identifying how the assembled and composed behavior can be constructed in UML 2.0. Section 4 contains evaluation and discusses related work. The concluding Section 5 outlines future work.

2. Basic and Trace-Based UC View

In [14], we introduced *Generic UC View*, a generic formal model of use cases, mostly inspired by our previous work on behavior protocols designed for behavior specification of nested software components [15, 16]. In this section, we introduce *Basic UC View* (a modification of Generic UC View) and its specialization *Trace-Based UC View*, which allow us to analyze how the behavior specification mechanisms in UML 2.0 address behavior assembly and behavior composition.

2.1. Basic Concepts

An entity S composed of sub-entities $A_1, ... ,A_n$ forms the *scope* of each A_i; the topmost scope is called *system*. An entity A_i communicates through *connections* with other (*actors*) A_j of the scope S, and with other external actors located in the parent scope; in this setting, A_i is the SuD (System under Discussion). Advantageously, the nesting of entities and their scopes can be expressed as a scope diagram; an entity is represented either by the stick-figure symbol or by a rectangle, a line represents a connection. Note that a scope diagram captures the relations among entities, not among use cases and entities (as a UML use case diagram does). Figure 1 shows the scope diagram of a sample *Ticket Reservation* system. The use cases TRS1 and TRS2 in Fig. 3 describe the interaction of Ticket Reservation System (as the SuD) with its surrounding actors in the scope of Ticket Reservation.

2.2. Basic UC View

Scenarios, behavior. Assuming A is an entity in a system Σ, a particular way of activity of A in a run of Σ is captured as a *scenario*. The possible scenarios of A in any run of Σ form the *behavior* of A. In order to accommodate a broad range of systems and behavior specification mechanisms, we denote by $Scenarios_\mu$ the set of scenarios and sub-scenarios reflecting the activities of all possible entities in a considered collection of systems; by the subscript μ we emphasize that the elements of $Scenarios_\mu$ are expressed in a specification mechanism μ. All the scenarios of A expressible in μ form the *μ-behavior* of A denoted $Com_\mu(A)$, $Com_\mu(A) \subseteq Scenarios_\mu$.

Use case. Given an entity A as SuD, we denote by $UC_{\mu\,i}^{A}$ the i-th use case specified for A; $Com_\mu(UC_{\mu\,i}^{A}) \subseteq Scenarios_\mu$ stands for the behavior specified by the use case. All the use cases which can be written in μ for A form the domain U_μ^A, formally $U_\mu^A = \{ UC_{\mu\,i}^{A} \}$ where i is from a given set of indexes.

Example. For illustration, consider the textual use cases from Fig. 3 (specification mechanism "Text"). By the convention above, we should denote them as $UC_{Text}^{TRS}{}_1$ and $UC_{Text}^{TRS}{}_2$. The set $Scenarios_{Text}$ is defined as the set of all possible sequences of steps of the entities shown in Fig. 1. Therefore, $Com_{Text}(UC_{Text}^{TRS}{}_1) = \{ <$ Customer selects to buy ..., System validates .., The Bank ... $> \}$, $Com_{Text}(UC_{Text}^{TRS}{}_2) = \{ <$ System looks up ..., System validates ..., ... tracking_id ... $> \}$. Note, however, that Fig. 3 captures just fragments of the behavior specification; in reality, $Com_{Text}(UC_{Text}^{TRS}{}_1)$ and $Com_{Text}(UC_{Text}^{TRS}{}_2)$ would be much larger (due to extensions and variations).

Use case expressions. We intend to define *use-case expressions* with the intention to capture assembled behavior – i.e., the behavior specified by several use cases for a particular A as SuD. Therefore we assume that use case expression are formed by means of *assembling operators* from the set $OP = \{ + ; | * \}$, denoting alternative, concatenation, parallel composition, and repetition. Our choice of OP is driven by the intention to assemble behavior via a "minimal set" of operations with intuitively clear meaning. For example, if $UC_{\mu\,1}^{A}$, $UC_{\mu\,2}^{A}$ are use cases, the use case expression $UC_{\mu\,1}^{A} + UC_{\mu\,2}^{A}$ means that either A behaves like $UC_{\mu\,1}^{A}$ specifies, or A behaves like $UC_{\mu\,2}^{A}$ specifies (alternative). In a similar way, $UC_{\mu\,1}^{A} ; UC_{\mu\,2}^{A}$ means that A chooses one of the "runs" specified by $UC_{\mu\,1}^{A}$ and then it behaves according to $UC_{\mu\,2}^{A}$ (sequencing). $UC_{\mu\,1}^{A} *$ stands for repetition, i.e., A behaves like specified by $UC_{\mu\,1}^{A}$ several times, while $UC_{\mu\,1}^{A} | UC_{\mu\,2}^{A}$ expresses parallel execution of the behaviors specified by $UC_{\mu\,1}^{A}$, $UC_{\mu\,2}^{A}$. All use cases in a use case expression have to employ the same behavior specification mechanism μ; again, the symbol denoting this specification mechanism is used as a subscript, e.g., $e_\mu = UC_{\mu\,1}^{A} + UC_{\mu\,2}^{A}$.

Above, we defined the meaning of the operators intuitively. To give an operator $op \in OP$ a precise meaning in a specification mechanism μ, we have to associate it with an operation θop: $Scenarios_\mu \times Scenarios_\mu \rightarrow Scenarios_\mu$. This way the behavior described by an expression of the form $Uc_{\mu\,i}^{A} op UC_{\mu\,j}^{A}$ is $Com_\mu(UC_{\mu\,i}^{A}) \theta op Com_\mu(UC_{\mu\,j}^{A})$. In a similar way, we inductively define the behavior of a general expression of the form $e_{\mu,1} op e_{\mu,2}$ as $Com_\mu(e_{\mu,1} op e_{\mu,2}) = Com_\mu(e_{\mu,1}) \theta op Com_\mu(e_{\mu,2})$, provided $Com_\mu(e_{\mu,1})$ and $Com_\mu(e_{\mu,2})$ were already defined.

For illustration, imagine textual use case $UC_{Text}^{TRS}{}_1$ and $UC_{Text}^{TRS}{}_2$ and the operation $\cup$ as the operation associated with $+$. This way $Com_{Text}(UC_{Text}^{TRS}{}_1 + UC_{Text}^{TRS}{}_2) = Com_{Text}(UC_{Text}^{TRS}{}_1) \cup Com_{Text}(UC_{Text}^{TRS}{}_2) = \{<$Customer selects to buy ..., System validates .., The Bank ...$>$, $<$System looks up..., System validates ..., ... tracking_id ...$>\}$.

Operators interpreted in μ. In general, for a particular μ, it can be impossible to associate an operation from $Scenarios_\mu \times Scenarios_\mu \rightarrow Scenarios_\mu$ with each of the operators in OP. Therefore, we introduce $OP_\mu \subseteq OP$ containing all those operators associated with such an operation.

Example. By a detailed analysis of the textual use case specification mechanism it becomes clear that it is hard to find a sound semantics for parallel composition of textual use cases (unless some limits are imposed on overlapping of the specified actions); on the other hand finding a

"reasonable" semantics for the remaining operators is easy via simple text operations, so that $OP_{Text} = \{ +, ;, * \}$.

Characteristic use case, native operations. Every behavior specification mechanism μ typically defines native operations upon behavior specifications. In Basic UC View, a native operation $f\colon U_\mu^A \times U_\mu^A \to U_\mu^A$ combines behavior specifications of use cases into behavior specification of a new use case; Sect. 3 provides a number of examples in this respect. To address behavior assembly, we find it very useful to construct a "summary use case" [7] $UC_{\mu\,i}^A$ from a set of use cases $\{ UC_{\mu\,k}^A \}$ via native operations by using a use case expression e_μ (its syntax tree) as guidelines. Driven by this motivation, we define $UC_{\mu\,i}^A$ to be a *characteristic use case* of a use case expression e_μ if $Com_\mu(e_\mu) = Com_\mu(UC_{\mu\,i}^A)$.

Implementing operators from OP_μ via native operations. In order to construct a characteristic use case $UC_{\mu\,i}^A$ of e_μ, we have to identify which of the operators from OP_μ have an implementation via native operations of μ.

We say that an operation $f\colon U_\mu^A \times U_\mu^A \to U_\mu^A$ *implements* $op \in OP_\mu$, if for all use cases $UC_{\mu\,i}^A$, $UC_{\mu\,j}^A$ it holds that $Com_\mu(UC_{\mu\,i}^A \ op \ UC_{\mu\,j}^A) = Com_\mu(UC_{\mu\,i}^A \ f \ UC_{\mu\,j}^A)$. Here, $UC_{\mu\,i}^A op \ UC_{\mu\,j}^A$ is a use case expression, while $UC_{\mu\,i}^A f UC_{\mu\,j}^A$ is a new use case.

If all the operators from OP_μ are implemented by native operations in μ, then, for every use case expression e_μ, a use case $UC_{\mu\,k}^A$ can be constructed recursively by following the syntactic structure of e_μ in such a way that $Com_\mu(UC_{\mu\,k}^A) = Com_\mu(e_\mu)$, i.e., $UC_{\mu\,k}^A$ is a *characteristic use case* of e_μ.

Example. Let us implement the operation "$;$" via a native operation of the use case specification mechanism Text. Such a native operation takes two textual use cases $UC_{Text\,i}^{TRS}$, $UC_{Text\,j}^{TRS}$ and creates a new use case $UC_{Text\,k}^{TRS}$, consisting of the steps of the main success scenario of $UC_{Text\,i}^{TRS}$, followed by the steps of the main success scenario of $UC_{Text\,j}^{TRS}$ (the steps are renumbered accordingly). The extensions are modified in a similar way.

Assembled behavior. As mentioned in Sect. 1.1 informally, a use case model of an entity A (SuD) is determined by all its use cases and by the way they are combined (specifying the assembled behavior of A). To capture these concepts formally, we assume that a use case model of A is a pair $< UR_\mu^A, e_\mu^A >$ where $UR_\mu^A \subseteq U_\mu^A$ is the set of relevant use cases ("all its use cases") upon which a use case expression e_μ^A is written. This expression specifies the *assembled behavior* $Com_\mu(e_\mu^A)$ approximating the behavior $Com_\mu(A)$. If there exists a characteristic use case of e_μ^A, we call it *representative use case* of the use case model.

2.3. Trace-Based UC View

To capture behavior composition, the content of the domain $Scenarios_\mu$ has to be defined explicitly. To do this, we introduce Trace-Based UC View (an extension of Basic UC View) based on traces. We have chosen this approach because traces are well understood and established in the behavior specification community [17]. To our knowledge, there are no formally-defined behavior semantics (suitable for the problem we are addressing) other than traces, a labeled transition system (LTS) [17] and an algebraic specification (which coexist with LTS as an alternative in defining operator semantics in some process algebras). Semantics defined via LTS and algebraic specifications, on which many well-known formal methods are based (Petri Nets, process algebras – CSP, CCS) have, from the point of view of this paper, features equivalent to trace based semantics. The main difference here is that there is the option to define simulation and bisimulation equivalences on an LTS (or via an algebraic specification), while on traces only trace equivalence can be defined (which is weaker than simulation and bisimulation). However, as we use behavior compliance [15, 16] as our equivalence relation, this is not an issue.

In Trace-Based UC View, an activity of A is a finite sequence of atomic events from a finite domain; these events can be represented by event labels from a domain Act_μ (also finite) and a scenario is a *trace* (finite sequence of event labels). Then, $Com_\mu(A) \subseteq Scenarios_\mu = Act_\mu{}^*$.

Composed behavior. Let us assume the following setting: An entity C is composed of entities A and B, which communicate via a connection (and via other connection with the environment of C). The goal is to obtain the behavior of C composed of $Com_\mu(A)$ and $Com_\mu(B)$. A and B perform their activities in parallel; explicit synchronization may occur whenever A and B communicate. The event labels of the actions serving for communication with the entities connected to A form the set $Msg_\mu(A) \subseteq Act_\mu$.

For entities A and B we define the relation $Pair_\mu^{A,B} \subseteq Msg_\mu(A) \times Msg_\mu(B)$ capturing the synchronization of events of A and B, e.g., sending and receiving a message. Any pair of events $t^A \in Msg_\mu(A)$ and $t^B \in Msg_\mu(B)$ is *synchronized* if $Pair_\mu^{A,B}(t^A,t^B)$ holds and in this case we call t^A,t^B a *synchronizing pair*. For technical reasons, we denote $Sync_\mu(A,B)$ the set of all labels representing events of A that are synchronized with some events of B.

Composed behavior of A and B is captured by interleaving of their traces (reflecting parallel execution) and merging the synchronizing pairs of the events (internal communication) into new tokens. This way, the composed behavior is the set of all traces resulting from interleaving of each $s^A \in Com_\mu(A)$, $s^B \in Com_\mu(B)$ in the following way:

(1) Every synchronizing pair t^A,t^B is merged into an internal event of C represented by a new event label $\tau t^A t^B$ (this is an enhancement to the internal event idea in CSP [17]).

(2) A resulting trace is constructed by interleaving of event labels from s^A and s^B; every occurrence of a synchronizing pair t^A, t^B such that t^A is directly followed by t^B or t^B is directly followed by t^A is replaced with $\tau e^A e^B$ according to the rule (1).

(3) An event label $t^A \in Sync_\mu(A,B)$ (as well as $t^B \in Sync_\mu(B,A)$) may not occur in a resulting trace and may only be processed via the rule (2). Traces where this condition would be violated are not included in the resulting behavior.
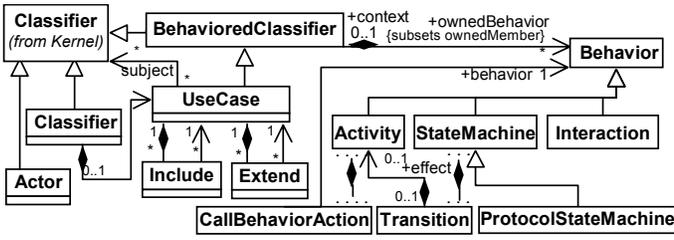
Figure 4: Relevant parts of the UML metamodel

2.4. What is it Good for: Consistency Reasoning

To reason on consistency of a hierarchical system specified via use cases, the following issues should be addressed:

(a) Does the composed behavior of the entities $A_1, A_2, ..., A_n$ (forming a scope S) as specified by the use case expressions $e_\mu^{A1}, e_\mu^{A2}, ..., e_\mu^{An}$ (determining assembled behaviors of these entities) comply with the behavior specified for S by e_μ^S?

(b) Does the assembled behavior as specified by e_μ^A really reflect the desired behavior of A? As this is hard to address directly, we will consider equivalence checking – in general reasoning on whether two use case expressions e_μ and e_μ' specify the same behavior, i.e. whether $Com_\mu(e_\mu) = Com_\mu(e_\mu')$.

(c) Does an entity A_i communicate with its neighboring entity A_j (in the same scope S) in the way A_j expects (and vice versa), i.e., are the assembled behaviors e_μ^{Ai} and e_μ^{Aj} of A_i and A_j "compatible"?

Note that all these issues require obtaining assembled behavior of the entities involved, and, moreover, (a) requires obtaining the composed behavior of $A_1,..,A_n$. As an aside, we have defined in [15, 18] consistency relations for trace-based scenarios (by the concepts of language conformance and consent operator) addressing the issues (a) – (c). Moreover, a tool exists to evaluate the relations in an automatized way.

3. Analyzing UML 2.0

In this section, we analyze how UML 2.0 supports assembled and composed behavior. By convention, we use *underlined italics* to refer to the abstractions defined by Basic UC View and sans-serif font for UML metamodel elements. We show the parts of the UML metamodel related to use cases in Fig. 4 (synthesized from several diagrams of the UML 2.0 specification [1]; the core part comes from the UseCases package).

An *entity A* is interpreted by a Classifier. The UseCase metaclass interprets the concept of a *use case* $UC_\mu^A{}_j$; the Classifier associated via the subject association is the *SuD* (*A*) of the use case. A UseCase is associated with a Behavior specification via the ownedBehavior association (inherited from BehavioredClassifier); the associated behavior may be an instance of one of the behavior specification mechanisms predefined in UML 2.0: Activity, Interaction, ProtocolStateMachine, and StateMachine. Therefore, in terms of Sect. 2, there are four concrete use case specification mechanism defined by UML 2.0.

For each of them, we below provide a brief characteristic and show: (i) how *Scenarios* are interpreted and whether a scenario is a trace, (ii) how and whether *assembled behavior*, (iii) *representative use case*, and (iv) *composed behavior* can be obtained, and (v) whether consistency reasoning is possible.

3.1. Interaction

In the Interaction specification mechanism (denoted Int), behavior is defined in terms of Messages sent among collaborating Classifiers, resulting into partially-ordered sets of event occurrences (basically, Interactions are an enhancement of sequence diagrams from UML 1.5). A communicating Classifier is represented by a Lifeline (e.g., Clerk in Fig. 5). A Message (e.g., `lookupTicket`) connects two MessageEnds. A MessageEnd may be attached to a Lifeline, capturing an EventOccurrence – the message being sent or received (e.g., both ends of `lookupTicket`). Alternatively, a MessageEnd may be attached to the boundary of the Interaction, forming a Gate (`validatePayment` in Fig. 5).

Scenarios: Scenario is a trace – a sequence of event occurrences. More formally, behavior of an Interaction is defined by a pair $[P,I]$, where P is the set of valid traces and I is the set of invalid traces. The set of event labels corresponding to operations associated with MessageEnds, together with the event kind (*send* or *receive*), form the domain of event labels Act_{Int}. Further, $Scenarios_{Int} = Act_{Int}*$ and $Com_{Int}(UC_{Int}{}^A{}_j) = P$.

Assembled behavior: To define semantics of use case expressions, each operator from *OP* (used in Int) has to be associated with an operation θop. As scenarios are traces, for each of the operators (`+ ; * |`) such an operation is defined upon languages (sets of traces): union, concatenation, repetition and parallel composition (based on interleaving). Thus, $OP_{Int} = \underline{OP}$.

Representative use case: Interactions feature the construct CombinedFragment to combine Interactions together and InteractionOccurrence to refer to (include) an existing Interaction. The semantics of Combined-Fragment is defined in terms of the sets P and I, based upon the traces of its operands and an internal special operator. We define functions *Alt*, *Seq*, *Par,* and *Rep*, constructing a new Interaction as an CombinedFragment employing the operator alternative, strict sequencing, parallel execution, and loop respectively. Each operand of the CombinedFragment is an InteractionOccurrence referring to the respective parameter of the function; these functions provide an implementation of all the operations in OP_{Int}. Thus, Interactions permit to construct a representative of all use case expressions.

Composed behavior: We interpret the set of event occurrences on MessageEnds that are a Gate as $Msg_{Int}(A)$. Further, we define $Pair_{Int}{}^{A,B}(t^A,t^B)$ iff (t^A=<*send*, m> $\land$ t^B=<*receive*, m>) $\lor$ (t^A=<*receive*, m> $\land$ t^B=<*send*, m>). As scenarios are traces, and $Pair_{Int}{}^{A,B}(t^A,t^B)$ is defined, composed behavior is interpreted in Interactions.

Consistency reasoning: It is not explicitly addressed in Interactions. However, as the semantics of Interactions is defined via traces, it is possible to use the relation of language conformance and the consent operator for consistency reasoning (sect. 2.4).

3.2. Activity

The Activities specification mechanism (denoted Av) employs graphs (similar to Petri Nets) to specify behavior in terms of passing control and data tokens along edges connecting nodes of the graph; of the node types considered, a ControlNode may coordinate the flow of the activity, while ExecutableNode models executable actions, an important subclass of ExecutableNode is Action. In Fig. 6, `cs.lookupTicket` is a CallBehaviorAction (special case of Action); below, a ForkNode (specialization of ControlNode) creates two concurrent branches. Further, `csNotify.reportTicketPrice` is an AcceptEvent-Action (also special case of Action) and the DecisionNode branching into alternative branches guarded by constraints `paymentDeclined` and `paymentProvided` is another specialization of ControlNode.

Scenarios: An execution of an Activity generates a scenario, capturing history of the Actions executed. However, there is no assumption of atomicity of Actions (Actions can overlap; an action *"takes place over a period of time"* [1, p.265]). Thus, an execution cannot be interpreted as a trace; UML 2.0 does not define any further details on *Scenarios$_{Av}$*.

Assembled behavior: As the semantics of parallelism and interleaving/overlapping of Actions is not clearly defined in Activities, it is not possible to find an interpretation of "|". On the other hand, operators "+", ";" and "*" are each associated with an operation θop; these operations are defined via concatenation of scenarios. Therefore, $OP_{Av} = \{ +, ;, * \}$.

Representative use case: In Activities, there is no construct to include another Activity via reference; therefore, the only way to implement operations from OP_{Av} with native operations is to construct a new graph actually including the graphs being combined. Note that CallBehaviorAction cannot be used as an "include" mechanism, because the semantics of this action is to call a new behavior (in a new execution context); the semantics of this action (in the definition of *Scenarios$_{Av}$*) is just the call of the behavior, and not the behavior being called itself.

Activity graphs can be combined together with ControlNodes, however, only "+" can be interpreted this way, via DecisionNode. As for interpretation of ";", it inherently needs to properly identify the end of execution of an Activity. However, when an Activity creates multiple control tokens, this becomes a cumbersome task with too many special cases to handle (this should be a subject of further research). Therefore we conclude that it is not practically possible to implement *sequencing* (and neither *repetition)* via native operations and that constructing a representative use case is possible only in the case when the only operator used in the use case expression is "+".
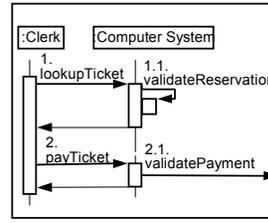

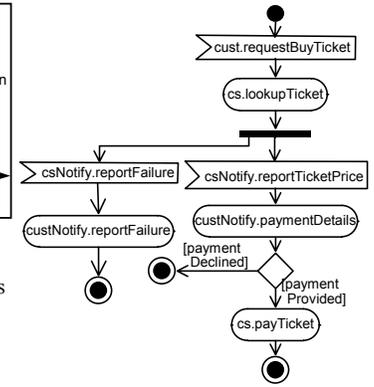**Figure 5**: A use case of Computer System captured as an Interaction

Figure 6: A use case of Clerk captured as an Activity

Composed behavior: As Scenarios$_{Av}$ are not traces, it is not possible to interpret composed behavior in Activities. Moreover, when considering whether it would be possible to establish a *Pair$_{Av}^{A,B}$* relation, we see that requests are generated by an InvocationAction (or a subclass, e.g., CallBehaviorAction), and received by an AcceptEvent-Action (or a subclass). However, these two class hierarchies have different structure and it would not be possible to establish a *Pair$_{Av}^{A,B}$* relation, even if *Scenarios$_{Av}$* were traces.

Consistency reasoning: As *Scenarios$_{Av}$* cannot be interpreted as traces, consistency reasoning is not possible.

3.3. State Machine

The StateMachines specification mechanism (denoted SM) is based on Harel state-charts [19], which enrich the automaton abstraction with composite states and their orthogonal regions. In principle, a transition from a composite state applies to all its substates; orthogonal regions model concurrently executing sub-machines. Harel state-charts define semantics via (i) event triggered transitions, and (ii) via atomic *actions* associated with entering or leaving a state, and with a transition. Time-durable *activities* are supported, but kept outside of the model. Special actions *start(X)* and *stop(X)* are used to start and stop an activity *X*. Thus, an execution of a Harel state-chart can be interpreted as a trace formed as a sequence of the action names. Also, accepting an event *X* may be captured as a special action *accepted(X)*.

On the contrary, UML 2.0 StateMachine replaces Harel's atomic action by the non-atomic Activity (sect. 3.2). In addition, a State may also specify a doActivity, which starts after entering the State (after completing the entry Activity) and *"is aborted prior to its completion"* if the state is exited before the doActivity completes. However, UML 2.0 does not specify the exact semantics of *aborting* a doActivity, neither the semantics of aborting the non-atomic Action possibly executed within a doActivity. Because of this unclear semantics, it is hard to employ the Harel's start(X)/stop(X) trick to wrap a non-atomic Action.

The State Machine demonstrated in Fig. 7 starts by receiving the event `cust.buyTicket`; the activity `validateReservation` is an internal action, while

`custNotify.reportTicketPrice` invokes an operation. Guards are specified in square brackets.

Scenarios: As the Activities are non-atomic and the issue of aborting a doActivity is not resolved, it is not possible to interpret execution of a StateMachine as a trace in general. The concept of a scenario is not explicitly captured, we thus only implicitly assume the behavior of a StateMachine reflected as subset of $Scenarios_{SM}$, capturing histories of execution of StateMachines (States visited, events processed, Transitions fired, and Activities executed).

Assembled behavior: The way we introduced $Scenarios_{SM}$ permits to define (based on concatenation) an operation θop to be associated with the operator "**+**", "**;**" and "*****" respective. Although StateMachines explicitly consider parallelism (with orthogonal regions), parallel execution is limited by the *run-to-completion* semantics: an event t_2 may be accepted only after a running transition triggered by accepting an event t_1 completes (i.e., all the Activities associated with processing t_1 complete). Even though, as a special case, the transitions triggered by the same event execute in parallel, orthogonal regions do not interpret "|" in general and thus $OP_{SM} = \{\ \textbf{+}\ ,\ \textbf{;}\ ,\ \textbf{*}\ \}$.

Representative use case: StateMachines permit to include (by reference) another StateMachine via a submachine state. To implement the operations in OP_{SM} ("**+**", "**;**" and "*****"), we define functions *Alt*, *Seq*, and *Rep*, constructing a new StateMachine. This StateMachine contains submachine states referring to the respective parameters of the function; the submachine states are joined via a junction Pseudo-State (Alt), via a Transition (Seq) and with a loop Transition (Rep); these functions implement all the operations of OP_{SM}.

Composed behavior: In StateMachines, scenarios are not traces and thus, it is not possible to interpret composed behavior. Moreover, reception of events is captured in triggers associated with Transitions, while responses are specified in the associated Activities. Thus, even if $Scenarios_{SM}$ were traces, it would not be feasible neither to establish a $Pair_{SM}^{A,B}$ relation, nor to acquire composed behavior.

Consistency reasoning: As scenarios cannot be interpreted as traces (and moreover, as composed behavior is not interpreted), consistency reasoning is not possible.

3.4. Protocol State Machines

The ProtocolStateMachine specification mechanism (denoted PSM) is a specialization of StateMachine and is used to specify the correct order of sequencing of Operation calls on a Classifier, typically an Interface (special view on an Classifier). Unlike the general StateMachine, PSMs are not permitted to associate Activities neither with a State (such as doActivitity) nor with a Transition (the effect Activity). A key point is that an Operation is specified in the trigger of the Transition (typically, the trigger will be a CallTrigger, although other types of events are permitted as well). Thus, depending on whether the Interface is used as a provided or required

Interface of a Port, a PSM specifies either Operation call events consumed or emitted by the Interface. However, as no "reaction" events are specified, a single PSM can specify only one "direction of communication".

Figure 8 demonstrates a PSM defining the order of operation calls to be received by the Computer System while performing the behavior specified by the StateMachine in Fig. 7. The PSM has been acquired by capturing only the events processed, omitting all the activities.

Scenarios: A PSM captures the order of invocations of Operations specified as the trigger of its Transitions. A PSM transition corresponds to the whole duration of the method invocation, and, same as in StateMachines, the *run-to-completion* semantics requires that no event may be accepted until the processing of the previous event completes. Consequently, Operation invocations cannot interleave; only after an operation call completes another call may be started.

Even though UML 2.0 does not introduce traces for PSMs explicitly, it is natural to interpret them as a sequence of Operation invocations; due to the non-atomicity of invocations, we represent an operation invocation by two atomic events, *request* and *response*; these events form the domain Act_{PSM}. Then, $Scenarios_{PSM} = Act_{PSM}*$.

Assembled behavior: As a trace model is defined for PSMs, semantics of all the operators (**+** **;** ***** |) is defined by associating them with the same operations upon languages as in Sect. 3.1 (Activities) and thus, $OP_{PSM} = \underline{OP}$. Note that "|" may create traces that cannot be generated by a PSM (also see below).

Representative use case: The operations "**+**", "**;**" and "*****" can be interpreted via functions *Alt*, *Seq*, and *Rep*, defined in the same way as for StateMachines. However, as a Transition is non-atomic (represented in a trace by two events request and response), "|" may result into traces where other events interleave with the pair of the related request and response events. Due to the *run-to-completion* semantics of PSM, such a behavior cannot be represented with a PSM. Therefore "|" cannot be implemented with a
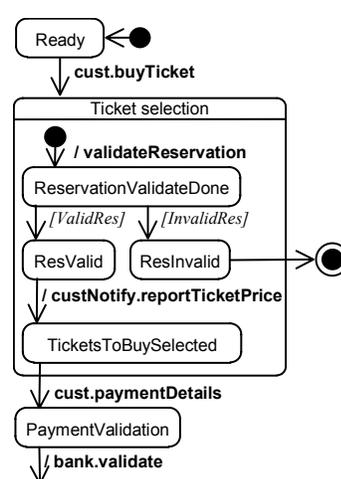

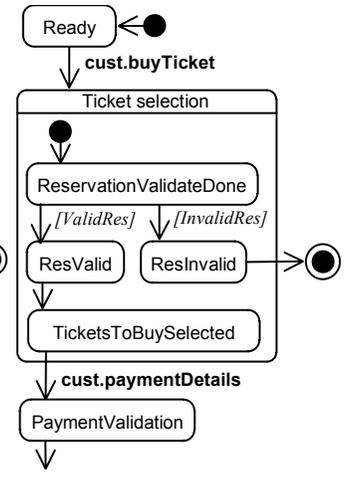**Figure 7**: Fragment of a Computer System use case captured as a State Machine

Figure 8: Fragment of a use case of Computer System captured as a Protocol State Machine

native operation and a representative use case can only be acquired when the operations are limited to "**+**", "**;**" and "*****".

Composed behavior: As noted above, PSMs are intended to specify behavior on an Interface, and a PSM specifies only the Operation call events but not the "reaction events"; although $Msg_{PSM}(A)$ may be interpreted as the set of all events considered by a PSM, it is not possible to establish $Pair_{PSM}^{A,B}$. At least the former argument implies that <u>*composed behavior*</u> cannot be interpreted in ProtocolStateMachines.

Consistency reasoning: The support of traces in PSMs provides a solid basis for defining a consistency relation. This issue is explicitly considered in UML 2.0; a ProtocolConformance relation may be established between a PSM and a StateMachine (or between two PSMs). However, the semantics of the relation is not clearly defined.

4. Evaluation and Related Work

Discussion: We claim that assembled behavior, representative use case, and composed behavior are important concepts which should be reflected in software design tools to make the use case idea really work. This claim is also inspired by the practitioners' concept of summary use case (e.g. [7]). Contrary to UML, we do not consider in our Basic UC View the Include and Extend use case relations as we focus on assembled and composed behaviors where no relations upon use cases are needed. Also some behavior specification mechanism feature a construct "include" resp. "extends"; however none of them can be interpreted as a binary operation $U_\mu^A \times U_\mu^A \to U_\mu^A$ (nor, in general, as an operation upon specifications only). The bottom line is that these constructs are not significant in our approach as the behavior of an included use case seamlessly becomes a part of behavior of the including use case. Thus, we "skip" use cases included in another use case, considering "complete" use cases only.

The way we modify Generic UC View in this paper allows to analyze the behavior specification mechanisms in the emerging UML 2.0. We conclude that only Interactions define trace-based scenarios and support obtaining assembled behavior, representative, use case as well as composed behavior. On the contrary, behavior of Activities and StateMachines cannot be interpreted via traces. Assembled behavior can be obtained only for a limited set of assembly operators ($OP_{AV} = OP_{SM} = \{+ ; *\}$). Moreover, while for StateMachines the whole OP_{SM} can be implemented via native operations, for Activities only "**+**" is the case. Only for expressions featuring solely these (natively implemented) operators a representative use case can be constructed. Behavior of ProtocolStateMachines (PSM) can be captured as traces, however, PSMs are not designed to specify the behavior of an entity, but only of a single Interface of a Port. Thus, PSMs do not provide an interpretation of composed behavior either.

To justify that the idea of assembled behavior, representative use case, composed behavior, and consistency

reasoning is sound, we refer the reader to Pro-cases [14], based on behavior protocols [15] as a model of Generic UC View (they also fit into the Trace-Based UC View). Pro-cases (PC) feature trace-based scenarios and thus, all behavior assembly operations are defined on traces ($OP_{PC} = \underline{OP}$); moreover, native operations exist to implement the whole OP_{PC}. Composed behavior is interpreted and a decidable consistency relation defined. A tool exists to verify this relation.

Related Work: In [20], the authors analyze how behavior can be specified via composition of UML StateMachines. Due to the *run-to-completion* semantics and due to reactions encapsulated in Activities associated with a transition, StateMachines cannot model recursive calls; an attempt to do so will lead to a deadlock. The problem is addressed by introducing *method state machines* (MSMs). They implement method bodies as state machines containing special constructs, *invocation boxes*, referring to methods implemented by other MSMs. This extension of state machines also allows to interpret assembled behavior (all operations from <u>OP</u>).

Use Case Maps (UCM) [21] employ a visual notation to capture the trajectory of a use case through the hierarchy of a system. Similar to <u>*use cases*</u>, UCM models behavior by capturing significant/typical scenarios. Instead of assembling behavior, UCM puts focus on describing behavior at multiple levels of entity nesting and visualizing such scenarios in a single diagram.

The analysis of support for use cases in UML 1.x in [22] also identifies the need to distinguish between "complete" and "fragment" use cases, reflecting the way we select the set of relevant use cases in the definition of a use case model. Further, this work focuses on relations between a use case and its scenarios, concluding that the use case should be associated with a set of scenarios. Interestingly, this work also identifies the need to capture the connections among entities involved in a use case (SuD and actors); the proposed *use case units* are similar to the scope diagrams (sect. 1.1).

The UML (1.4) view of <u>*use cases*</u> is also analyzed in [23], however, the authors rather focus on use case relations (include, extend), claiming that these are not defined precisely enough to avoid ambiguities and misunderstandings.

In [24], the author aims at improving the support provided by CASE tools in the software development process. He points out that UML behavior specification mechanisms have "unclear semantics" and specify "partly redundant and partly complementary information (without underling formal approach)". He also identifies lack of formal checks for consistency among different behavior specification mechanisms. Options are analyzed for automating the creation of class diagrams from a UML use case model; an approach for generating StateMachines from sequence diagrams is described.

In [25] a use case is represented as an Activity diagram where for each Action a graph transformation rule is defined upon the Collaborations of the objects involved in the

Action. Conflicts/dependencies between Actions in different use cases (between $\underline{UC^A_i}$ and $\underline{UC^A_k}$) can be identified.

5. Conclusion and Future Work

We demonstrated Basic UC View, a simple formal model capturing key abstractions in use case specifications; with this model, we identified the concepts of assembled behavior, representative use case, and composed behavior, which should be reflected in software design tools to make the use case idea really work. We employed it and its specialization Trace-Based UC View to analyze the four behavior specification mechanisms available in UML 2.0. As a result we showed that none of these mechanisms explicitly addresses assembled behavior, representative use case, nor composed behavior. However, we draw a way to interpret these concepts in Interactions.

Thus, although UML 2.0 introduces a hierarchical component model and a behavior specification framework which can be employed to specify behavior of components starting with use cases, except for Interactions, the framework does not support reasoning on consistency issues in component based design when use cases are employed to specify the behavior.

In our future work, we aim at enhancing Protocol-StateMachines to capture the interleaving of Operation invocations on multiple Ports of a Classifier in such a way that composed behavior could be obtained; this has been already partially achieved in [26] with Port State Machines (PoSM). To support consistency reasoning, we aim to formally define the compliance relation by means of OCL.

To provide a proof-of-the-concept, a UML extension will be designed and a tool implemented supporting use case expressions. This way, assembled behavior of a use case model will be captured. Moreover, PoSMs will be included in this extension, employing the verifier tool [27] already available for behavior protocols to reason on consistency of PoSM specifications.

References

[1] Object Management Group (OMG): Unified Modeling Language: Superstructure, version 2.0, Revised Final Adopted specification, ptc/04-05-02, http://www.omg.org/

[2] Magee, J., Kramer, J.: *Dynamic Structure in Software Architectures*, in Proceedings of SIGSOFT FSE 1996, San Francisco, California, USA, October 16-18, 1996. ACM SIGSOFT Software Engineering Notes 21(6), Nov. 1996

[3] Plasil, F., Balek, D., Janecek, R.: *SOFA/DCUP Architecture for Component Trading and Dynamic Updating*, In Proceedings of ICCDS '98, Annapolis, IEEE Computer Soc.

[4] Bruneton, E. Coupaye, T., Stefani, J.B.: *The Fractal Component Model*, Draft 2.0-3, February 5, 2004, http://fractal.objectweb.org/specification/

[5] Graham, I.: *Object-Oriented Methods: Principles and Practice*, Addison-Wesley Pub Co, ISBN: 020161913X, 3rd edition December 2000

[6] Larman, C.: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Prentice Hall PTR, ISBN: 0130925691, 2nd ed, 2001

[7] Cockburn, A.: *Writing Effective Use Cases*, Addison-Wesley Pub Co, ISBN: 0201702258, 1st edition, January 2000

[8] Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley Pub Co; ISBN: 0201544350; 1st edition (June 30, 1992)

[9] Object Management Group (OMG): Unified Modeling Language (UML), version 1.5, formal/2003-03-01

[10] D'Souza, D. *Components with Catalysis*, www.catalysis.org, 2001

[11] Grieskamp, W., Lepper, M., Schulte, W., Tillmann, N.: *Testable Use Cases in the Abstract State Machine Language*, APAQS'01, December 10 - 11, 2001, Hong Kong

[12] Stevens, P.: *On Use Cases and Their Relationships in the Unified Modelling Language*, in Proceedings, FASE 2001 (part of ETAPS 2001), Genova, Italy April 2-6, 2001, Springer LNCS 2029, ISBN 3-540-41863-6

[13] Hurlbut R. R.: *A Survey of Approaches For Describing and Formalizing Use Cases*, Expertech, Ltd., XPT-TR-97-03, http://www.iit.edu/~rhurlbut/xpt-tr-97-03.pdf

[14] Plasil, F., Mencl, V.: Getting "Whole Picture" Behavior in a Use Case Model, in *Transactions of SDPS: Journal of Integrated Design and Process Science* 7(4), pp. 63-79, 2003

[15] Plasil F., Visnovsky, S.: Behavior Protocols for Software Components. *IEEE Trans. Software Eng.* 28(11), Nov. 2002

[16] Plasil, F., Visnovsky, S., Besta, M.: *Bounding Behavior via Protocols*, Proceedings of TOOLS USA '99, Santa Barbara, CA, Aug. 1999.

[17] Bergstra J. A., Ponse A., Smolka S.A.: *Handbook of Process Algebra*, Elsevier 2001, ISBN 0444828303

[18] Adamek, J., Plasil, F.: *Behavior Protocols Capturing Errors and Updates*, Proceedings of the 2nd International Workshop on Unanticipated Software Evolution, ETAPS, Warsaw, 2003

[19] Harel, D.: *Statecharts: A visual formalism for complex systems*, Science of Computer Programming 8 (1987), pp. 231-274, Elsevier Science Publishers B.V. (North-Holland)

[20] Tenzer, J., Stevens, P.: *Modelling recursive calls with UML state diagrams*, Proceedings of FASE 2003 (part of ETAPS 2003), Warsaw, Poland, Apr. 2003, LNCS 2621, Springer

[21] Amyot, D., Mussbacher, G.: *On the Extension of UML with Use Case Maps Concepts*, in Proceedings of UML 2000, York, UK, Oct., 2000, LNCS 1939, Springer, 2000

[22] Isoda, S.: *A Critique of UML's Definition of the Use-Case Class*, in Proceedings of UML 2003, Oct 20-24, San Francisco, 2003, Springer-Verlag, LNCS 2863

[23] Genova, G., Llorens, J., Quintana, V.: *Digging into Use Case Relationships*, in Proceedings of UML 2002, Dresden, Germany, LNCS 2460 Springer 2002

[24] Zuendorf, A.: From Use Cases to Code – Rigorous Software Development with UML, Tutorial T4, ICSE 2001: May 12-19, 2001, Toronto, Ontario, Canada

[25] Hausmann, J. H., Hecke, R., Taentzer, G.: *Detection of Conflicting Functional Requirements in a Use Case-Driven Approach*, ICSE 2002, Orlando, FL, USA, May 19-25, 2002

[26] Mencl, V.: *Specifying Component Behavior with Port State Machines*, Accepted for publication in proceedings of Compositional Verification of UML Models workshop (Oct 21, 2003, part of UML 2003) in a volume of the Electronic Notes in Theoretical Computer Science, Elsevier Science

[27] Mach, M., Plasil, F.: *Addressing State Explosion in Behavior Protocol Verification*, Accepted for publication in proceedings of SNPD'04, Beijing, China, Jun 2004