Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science 101 (2004) 3–24

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Transformation and Verification of Executable UML Models

Günter Graw [1]

ARGE ISKV, Essen, Germany

Peter Herrmann [2]

University of Dortmund, Dortmund, Germany

Abstract

In addition to static structures, the Unified Modelling Language (UML) supports the specification of dynamic properties of objects by means of statechart and sequence diagrams. Moreover, the upcoming UML 2.0 standard defines several kinds of actions to specify invocations, computations and the access of structural features. The formal specification technique compositional Temporal Logic of Actions (cTLA) provides for modular descriptions of behavior constraints and its process composition operation corresponds to superposition. Furthermore, cTLA facilitates the selection of an arbitrary subsystem of a complex specification which is composed of processes. We introduce an approach for formal-based refinement verifications of detailed UML models which fulfill more abstract ones. In a first step of the verification, the abstract and the detailed model are transformed to cTLA specifications. Thereafter, we can prove that the cTLA specification of the more detailed model implies the cTLA description of the more abstract one by application of the model checker TLC (Temporal Logic Checker).

Keywords: UML, statecharts, sequence diagrams, compositional Temporal Logic of Actions, cTLA, TLC.

1 Introduction

UML (Unified Modeling Language) [5] is the defacto standard for the specification of software under the care of the OMG (Object Management Group)

[1] Email: grawg@gmx.de

[2] Email: Peter.Herrmann@udo.edu

doi:10.1016/j.entcs.2004.09.006

since 1997. The OMG is also responsible for the evolution of the UML. Meanwhile several attempts have been started to form an executable version of UML (cf. e.g., [29]). Stephen Mellor and Marc Balcer [26] provided an xUML (executable UML) profile which supports the enaction of UML models by the aid of simulation tools. Moreover several companies (e.g. Kennedy Carter, Projecttech) developed tools which support the enaction and simulation of executable UML models. The executability of UML models is in particular of interest for the MDA (Model Driven Architecture) initiative (cf. e.g., [22]) started by the OMG two years ago, which has the aim to generate executable code from models specified in UML.

Our approach has stronger requirements on the correctness of models which are used as input for transformation and generation tools. The upcoming UML 2.0 standard [21,27,32] includes an improved action semantics and new semantical foundations for sequence diagrams and activity diagrams. Our former work concentrated on the formalization of UML 1.4 models using cTLA (compositional Temporal Logic of Actions) as a foundation [10,11,8] in order to prove formally that systems are refined in a correct fashion. cTLA [16] is a specification technique based on Lamport's TLA (Temporal Logic of Actions) [23] with a strong emphasis on constraint-oriented modelling and composition of specification blocks which are described in a process-like notation. The ability to compose arbitrary processes to subsystems facilitates the specification of interesting properties — even, if the specified system has a remarkable complexity — in an acceptable amount of time.

Moreover, cTLA supports refinement proofs verifying that a system fulfills certain properties. Due to the compositional structure of cTLA system definitions, a verification can be reduced to relatively easy provable lemmas (cf. [16]) each stating that a — usually small — subsystem realizes a single cTLA process describing a certain aspect of the property as well as some — normally trivial — consistency checks.

The structured verification is a basis for so-called specification and verification frameworks supporting specification and verification in certain application domains. At first, a framework contains libraries of cTLA processes which may be instantiated and composed in order to create specifications of systems and properties in a rather easy fashion. At second, the framework contains theorems proven by the framework designers which correspond directly to the proof steps of a structured verification. Thus, a refinement proof can be performed by selecting suitable theorems and by performing some simple consistency checks which can also be supported by a tool (cf. [18]). We already created frameworks for the domains of communication protocol verification [15,16], hazard analysis of technical processes [17], and verification of security properties of component-structured software [14].

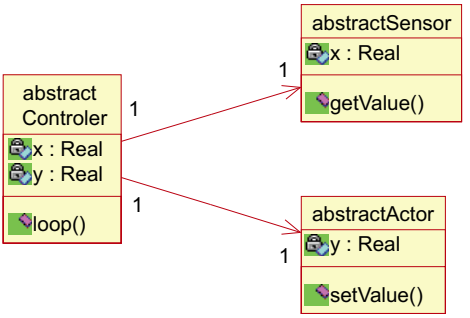

Fig. 1. The class model of the abstract Controller

Besides of the frameworks, which are beyond the scope of this article, one can also perform refinement proofs by means of model checks if the space of the reachable system states does not exceed a certain finite number (cf. [1,24]). The advantage of model checker application to the frameworks is that the proofs can be performed in a highly automated fashion. Since UML-based system descriptions tend to describe systems in a relatively abstract manner, we expect that the modeled state space of many real-life systems can be handled by a model checker. Therefore we decided to apply the powerful checker TLC (Temporal Logic Checker) [24,33], to perform the refinement proofs. Tools for the transformation of UML specifications into cTLA [9,11] and of cTLA specifications into TLA [19] exist. The transformation utilizes the compositional features of cTLA extensively since a UML diagram can be transformed into a separate cTLA process. Our approach complements a lot of other approaches formalizing UML diagrams by formal models (cf. [25,28,30,31]) which, however, do not use compositionality in the way explained in this paper.

We adapted our formalization to the upcoming UML 2.0 standard since the improved action semantics (which was introduced in UML 1.5) and the compositional aspects of sequence diagrams facilitate the transformation into cTLA processes. Based on these new semantical features we redefined our transformation rules from UML diagrams to cTLA processes fostering the generation of very compositional cTLA system descriptions which facilitate the carrying out of suitable refinement proofs by means of the model checker TLC. In order to stay within the boundaries of this paper, we had to restrict ourselves into sketching the process of transforming a UML model into the corresponding cTLA processes and to verify some example properties.

In Secs. 2 to 4, we explain the UML 2.0 statecharts, sequence diagrams, actions, and activities. After an introduction to cTLA in Sec. 5, we focus on the transformation of statechart diagrams and sequence diagrams into cTLA in Secs. 6 and 7. Finally, we sketch the steps of an example proof in Sec. 8.

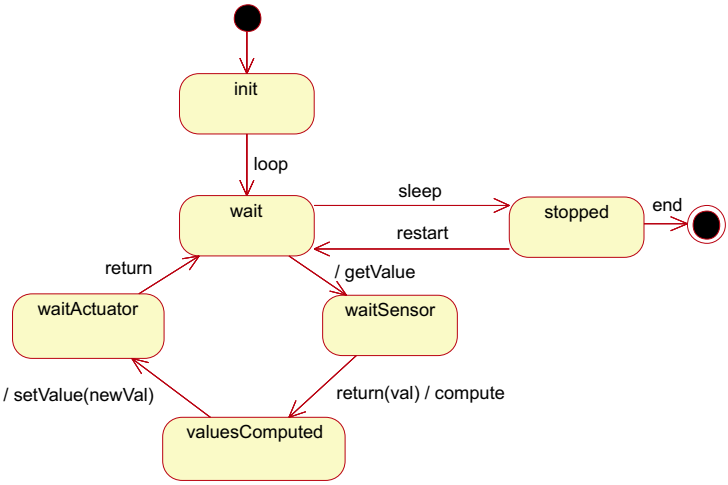

Fig. 2. The statechart of the abstractController class

2 Statecharts in UML 2.0

In UML models, states can be used to define attributes of an object and its behavior in a rather fine-grained way. Here, we apply them in a more abstract fashion in order to model the current situation of an object and its reaction on incoming events. Fig. 1 shows an example class-diagram describing a controller for a technical process which gets input from a sensor and forwards corresponding output values to an actor unit. As depicted in Fig. 2, a state description in UML 2.0 contains an unambiguous name. Moreover, one can define activity identifiers which are accompanied by the keywords *entry*, *exit* or *do* each. An activity (cf. Sec. 4) marked by *entry* is executed when the object enters the state containing the entry statement. Activities accompanied by *exit* are carried out when the state is left, whereas activities marked by *do* are triggered when an object remains in a particular state for a while.

In UML 2.0, transitions can be provided with a statement containing an event name, a guard condition, and an activity identifier. By the event name it is possible to specify a UML change event, call event, send event or completion event. Change, call and send events are triggered by changing an object attribute resp. by execution of a call or a send action whereas a timed event refers to a certain real time constraint. A transition is executed if the specified event occurs and the guard condition specified in the statement holds as well. In contrast, so called completion transitions or triggerless transitions depending on a completion event are carried out without an external trigger. A completion event fires if all transitions and entry resp. state activities in the currently active state are completed. It is preferred against other events in order to prevent deadlocks. Furthermore, one can allow the deferral of events.

If an event cannot be processed in the current state, it is stored in an event queue and can be used later. During the execution of a transition, the activity identified in the transition statement is carried out. Similar to Harel's statechart diagrams [12], one can define so-called composite states from substates which can contain substates as well. A composite state can be a nested state corresponding to the OR-States in a statechart diagram. If an incoming transition of the nested state is fired, exactly one of its substates gets active. The other kind of a composite is a concurrent state which corresponds to an AND-state in a statechart diagram. Here, the substates carry out transitions concurrently. A special class of states are pseudostates which have to be left immediately after being entered. Therefore, pseudostates must not contain *do* activities which are only executed if a state remains active for a while. Well-known pseudo-states are initial states. In contrast, final states are not pseudostates since an object remains in this state after reaching it. Another class are decision nodes which are used to model decision processes between alternating state-transition sequences. In nested states history states (e.g. the state H*) can be applied to store the lastly visited substate of a nested state. By executing an incoming transition of a history state the substate stored by it is reached.

To model the processing of events and correspondingly, the selection of transitions UML 2.0 defines a special state machine which is based on the run-to-completion semantics. According to this semantics only one event may be processed at a point in time and the processing cannot be interrupted by other events. By special state configuration information the state machine describes which state resp. substates are currently active (cf. [10]).

3 Sequence Diagrams in UML 2.0

In UML 2.0, sequence diagrams are strongly influenced by the current Message Sequence Chart (MSC) standard [13]. Sequence diagrams consist basically of object instances and their lifelines. Event occurrences may be found on a lifeline and correspond to actions which are executed in an object. For instance, an action may represent the creation and the reception of a message. The occurrence of messages is described by the aid of traces. A trace consists of sequences of event occurrences $< E_1, E_2 \ldots, E_n >$. In UML 2.0, it is possible to incorporate traces of foreign sequence diagrams into another diagram using the keyword *ref* and an identifier which refers to another sequence diagram. Moreover, one can compose sequence diagrams from other diagrams by application of so-called *CombinedFragments*. A *CombinedFragment* may contain so called *InteractionOperands*. That are special regions in the diagram where events may occur in arbitrary orders defining a set of alternate traces. The

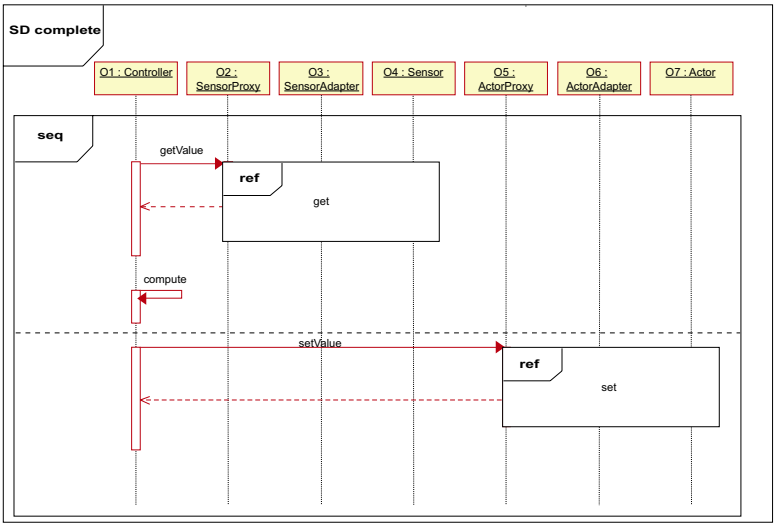

Fig. 3. The sequence diagram of the Controller

mode of interleaving between the events in such a region is determined by a special *InteractionOperator*. In the following, we will introduce two of the currently 11 defined InteractionOperators:

- *seq*: This *InteractionOperator* determines that the *CombinedFragment* provides a weak sequencing of the behavior of its *InteractionOperands*. Weak sequencing is defined on a set of traces which fulfill the following three properties. Firstly, the order of event occurrences of each *InteractionOperand* is preserved in the result. Secondly, event occurrences on different lifelines which stem from different *InteractionOperands* may occur in an arbitrary order. Thirdly, the event occurrences on the same lifeline stemming from different *InteractionOperands* are sequenced with respect to the order of these *InteractionOperands*.
- *strict*: The strict *InteractionOperator* implies that a *CombinedFragment* has a strict order of the event occurrences belonging to each *InteractionOperand*. The strict sequencing, however, is only enforced on the uppermost level of *CombinedFragment* with the operator strict. Thus, if the strict *CombinedFragment* A contains another combined fragment B, the event occurrences of B will not be directly compared with those of A.

In the sequence diagram listed in Fig. 3, an interaction scenario is described. It contains two *CombinedFragments* the first of which is depicted in Fig. 4. The corresponding sequence diagram contains a *CombinedFragment* which is marked by the *InteractionOperator strict*. Thus, the event occurrences of this *CombinedFragment* have to keep a strict order.

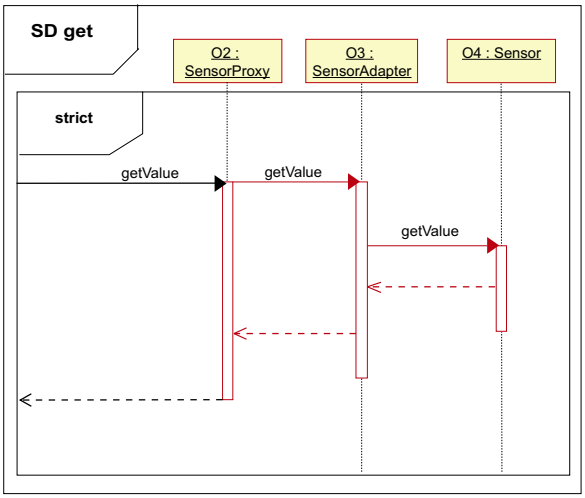

Fig. 4. The reused sequence diagram

4 Actions and Activities in UML 2.0

Another major improvement of UML 2.0 is the new Action Semantics defining a meta-model for action based description languages [32]. In contrast to the traditional Object Constraint Language (OCL) it facilitates the description of dynamic behavior enabling the generation of implementation code from UML models. Actions are the fundamental units of executable functionality. The Action Semantics does not define a particular syntax for action statements but more abstract class definitions which can be realized by applying various different syntaxes. Two actions exchange data and object information via special input and output pins.

The UML 2.0 superstructure [27] distinguishes Invocation Actions, Read Write Actions, and Computation Actions. *Invocation Actions* are used to perform operation calls and transmission of signals between objects. A relevant derived action is the so called *CallOperationAction* which transmits an operation call request to a target object where it invokes an associate behavior. The operation call is completed by a reply transmission which is performed by a *ReplyAction*. A CallOperationAction may be carried out synchronously blocking the caller until receiving the reply transmission. In contrast, if the action is asynchronous, the caller may proceed immediately and acts in concurrence with the invoked action. At the target object the receipt of an operation call triggered by a remote object is handled by an *AcceptCallAction*. This action produces a special output token which is used later to supply the results of the associated behavior to the ReplyAction returned to the caller. Other Invocation Actions are used to send or broadcast signals to target objects in order

to carry out operations without reply transmissions and to transmit objects to target objects.

A *Read Write Action* is used to perform read and write access to structural features like object attributes and properties, to create and destroy objects, and to handle links between objects. Derivations are *Structural Feature Actions* to handle access to structural features, *Object Actions* to manage the life cycle of an object, and *Association Actions* in order to operate on links and associations. An example of the first subclass is *ReadStructuralFeature* retrieving the values of a uniquely identified and not statical structural feature of a certain target object. So called Object actions are responsible for the creation and destruction of objects. A CreateObjectAction is an action which creates an object conforming to a class. The destruction of objects is performed using a DestroyObjectAction.

Finally, *Computation Actions* invoke primitive functions computing output values from input values without reading or writing other system resources.

A CreateObjectAction is an action which creates an object conforming to a class. The destruction of objects is performed using a DestroyObjectAction.

The actions are incorporated into activities which define certain orders of action executions. In our current work, we concentrate on so called *Basic-Activities* which, in contrast to more powerful extensions, do not allow the concurrent execution of different actions. An activity is modelled in a Petri Net-like style containing nodes which are connected by edges forming a complete flow graph for data and control values. An activity model consists of three types of nodes: An *action node* is used to define an action operating on received control and data values and providing control and data to other actions. *Control nodes* are used to coordinate control flow as well as data flow by routing control and data tokens through the graph. Finally, object nodes are applied to store both objects and data temporarily until they can be accepted by an action.

Activity graphs may contain two kinds of directed edges connecting the activity nodes. On the one hand, *control flow edges* permit only the passing of control tokens between two actions. A control flow edge implies that the action at the target end of the edge (arrowhead) cannot start before the source action finishes. On the other hand, *object flow edges* describe the relations between output and input pins of subsequent actions. Only data tokens and objects are permitted to pass along object flow edges.

5 cTLA

Leslie Lamport's Temporal Logic of Actions (TLA, [23]) is a linear time temporal logic which describes safety and liveness properties (cf. [2]) of state tran-

```
PROCESS Adapter(data : ANY;
                processCallFct : SET[data → data];
                processReturnFct : SET[data → data];
                callAdapterEvents : SUBSET(data))
BODY
  VARIABLES
    qu : queue of data;
    sAdapter : {"init", "deqCall", "procCall",
                "called", "deqReturn", "procReturn"};
    val : data;
    sync : {"true","false"};
    ...;
  INIT ≙ sAdapter = "init" ∧ qu = empty ∧ sync = "false" ∧ ...;
  ACTIONS
    getCallProxy(value : data) ≙
      qu′ = insert(qu, value) ∧ sAdapter′ = sAdapter;
    dequeueCall ≙
      sAdapter = "init" ∧ sync = "false" ∧ qu ≠ empty ∧
      head(qu) ∈ callAdapterEvents ∧ sAdapter′ = "deqCall" ∧
      val′ = head(qu) ∧ qu′ = tail(qu) ∧ Unchanged(sync);
    processCall ≙
      sAdapter = "deqCall" ∧ sync = "false" ∧
      sAdapter′ = "procCall" ∧ val′ = processCallFct[val] ∧
      Unchanged(qu,sync);
    callControl(value : data) ≙
      sAdapter = "procCall" ∧ sync = "false" ∧ value = val ∧
      sAdapter′ = "called" ∧ sync′ = "true" ∧ Unchanged(val, qu);
    callControlReply ≙
      sAdapter = "called" ∧ qu ≠ empty ∧
      sAdapter′ = "deqReturn" ∧ val′ = head(qu) ∧
      qu′ = tail(qu) ∧ sync′ = "false";
    ...;
  WF : dequeueCall, processCall, callControl, ...;
END Adapter
```

Fig. 5. cTLA process of the adapter object in the design pattern

sition systems by means of canonical formulas. cTLA (compositional TLA, [16]) is based on TLA, but enables the suitable specification of systems in the notation of processes omitting canonical parts of TLA specifications. In particular, it enables the composition of system descriptions from implementation-oriented processes, constraint-oriented processes, and combinations. Process composition in cTLA has the character of superposition (cf. [3]). Here, a relevant property of a process or a subsystem is also a property of the embedding system. Therefore structured verification is possible (i.e., a proof that a system has a property can be reduced to the verification that a subsystem fulfills the property). A cTLA process acts as a modular specification component and a system can be specified by a set of coupled processes. A process has either the form of a simple process or of a process composition. Simple processes refer directly to state transition systems and can represent implementation parts as well as logical constraints. Fig. 5 depicts the simple cTLA process type `Adapter` which is used in our verification example (cf. Sec. 8). In the header, the process type name and generic module parameters (e.g., `processCallFct`)

are declared. The parameters facilitate the specification of a spectrum of similar but different processes by a single process type. The body of the process type defines the state transition system modelling a process instance. The state space is specified by state variables like `qu`, `sAdapter`, `val`, and `sync`. The initial condition `INIT` refers to state variables and defines the set of initial states (i.e., `sAdapter` initially has the string value `"init"`). The state transitions are specified by actions (i.e., `getCallProxy`). An action is a predicate on a pair of a current and a successor state modelling a set of state transitions. The state variables referring to the current state are described by ordinary variable identifiers (i.e., `sAdapter`) while variables referring to the successor state occur in the so-called primed form (i.e., `sAdapter'`). An action may be supplemented by action parameters (e.g., `var`). The next state relation of the modelled state transition system corresponds to the disjunction of the actions.

Liveness constraints in cTLA are modelled by means of fairness assumptions on actions. The specification in Fig. 5 contains a new construct `WF` listing some action names (e.g., `dequeueCall`). It declares that the corresponding actions have to be executed "weak-fairly" for all defined action parameters (i.e., a weak-fair action has eventually to be performed if, otherwise, it will be continuously enabled for an infinite period of time). Similarly, by the construct `SF` one can define "strong-fair" actions which have even to be executed if they are disabled from time to time. In order to guarantee the superposition property of cTLA, the fairness constructs are conditional. They force an action only if for a certain period of time both the action itself was enabled and the process environment does not block it. Due to these conditional fairness assumptions the fairness properties of local processes cannot be spoiled by the process environment guaranteeing that all composed systems describe a realizable behavior. A disadvantage of this definition, however, is that it is harder to check if the conditional fairness assumptions assure the desired system liveness properties. Nevertheless, one can overcome this problem by adopting a certain specification style guaranteeing that a fair process action is only blocked by the environment in system states if it is disabled locally, too. In our application example, this property holds trivially for nearly all of the fair actions since they are coupled only with other actions which are always enabled. For the remaining cases one has to prove the property by means of an invariant verification.

Systems and subsystems are described as compositions of concurrent processes. As in the ISO/OSI specification language LOTOS [20], a set of processes interact in a rendezvous-like way by performing actions jointly, and the data parameters of the actions can model the communication of values between processes. Each process encapsulates its variables and changes its state by atomic execution of its actions. The system state is the vector of the process

```
PROCESS concreteComposition
          (Sdata : ANY;
           aSprocCallFct : SET[data → data];
           aSprocReturnFct : SET[data → data];
           ...)
  PROCESSES cT : concreteController(...); cS : concreteSensor(...);
            cA : concreteActor(...); pS : Proxy(...);
            aS : Adapter(Sdata, aSprocCallFct, aSprocReturnFct);
            pA : Proxy(...); aA : Adapter(...);
            sd : SequenceDiagram;
  ACTIONS
    getCallSensor(value : Sdata; actionInfo1 : String;
                  actionInfo2 : String) ≙
      aS.callControl(value, actionInfo1) ∧
      cS.getCallAdapter(value, actionInfo2) ∧
      sd.possibleMessageOfSD(actionInfo1, actionInfo2) ∧
      cT.stutter ∧ cA.stutter ∧ pS.stutter ∧ pA.stutter ∧
      aA.stutter;
    getCallSensorReturn(...) ≙
      ...;
    ...;
END
```

Fig. 6. Process type `concreteComposition`

state variables. State transitions of the system correspond to simultaneously executed process actions or to so-called process stuttering steps (i.e., the process does not change its state). Since, moreover, a process participates in a system action either by exactly one process action or by a process stuttering step, one can define a system action by a conjunction of process actions and stuttering steps. In consequence, concurrency is modelled by interleaving while the coupling of processes corresponds to joint actions. The fairness assumption of a process action is inherited by the system action to which it is coupled. The design of cTLA process types as well as the composition of processes to systems and the transformation to TLA is supported by a compiler tool [19].

An example of a compositional cTLA process is depicted in Fig. 6. In the section `PROCESSES`, instances of cTLA process types are listed (e.g., in the example an instance `aS` of the process type `Adapter` introduced in Fig. 5). The coupling of process actions to system actions is defined in the part `ACTIONS`. For instance, the system action `getCallSensor` is coupled from the action `callControl` of process `aS`, the action `getCallAdapter` of process `cS`, and the action `possibleMessageOfSD` whereas the other processes of the system perform so-called stuttering steps in which their local variables are not changed.

6 Transformation of statecharts and actions into cTLA

The transformation of statecharts to cTLA processes was already described in [10]. The basic idea is to create a separate cTLA process for the state machine of each UML class. Instances of the cTLA process specify the behavior of

each object instantiated from the corresponding UML class according to its statechart diagram. Our transformation of a statechart to a cTLA process is based on the concept of run-to-completion semantics (cf. [6]) which assumes that only one thread of control is active in an object at a certain point of time. This concept forces the interleaving of concurrent processes.

The transformation is performed in two steps. At first, we flatten a state chart to another state chart containing only simple states and pseudostates. This is done by application of graph grammars (cf. [7]). In this step, we also consider the actions defined in a state. Actions of entry activities are shifted to every activity of an ingoing transition. In contrast, an action of an exit activity belonging to a state is shifted to every activity of an outgoing transition. Do activities are not supported since they invoke concurrent computations which can be interrupted at any point of time. This, however, violates our assumption of run-to-completion semantics for a state machine.

Moreover, by this transformation, we solve conflicts between transitions in an active state. According to the firing priorities of the UML (cf. [27]), we extend the guards of the lower priority transitions. In consequence, in the flattened statechart a transition t is only enabled if all other transitions, which in the original statechart are in conflict with t and have a higher priority, cannot be executed.

At second, we transform the flattened state chart into a cTLA process. Since this state chart corresponds to a simple state transition system, this transformation is straightforward. The states are modelled by a cTLA variable and the transitions by cTLA actions. As an example of a cTLA process resulting from a transformation, we depicted the process `Adapter` in Fig. 5. The variable `sAdapter` list the various states of the flattened statechart as Strings while the transitions between these states are specified by actions. A cTLA action (e.g., `processCall`) checks the currently active state and guard in the predicate and sets the value of the state variable `sAdapter` to the succeeding state. Non-determinism of UML transitions can be directly mapped to the cTLA actions. If two transitions of the flattened state chart are enabled non-deterministically, their two cTLA action representatives are also enabled in the same state.

Besides of transforming states and transitions, we have to consider other parts of a statechart. Each attribute of a class is transformed into a variable of the cTLA process representing a corresponding object. For instance, an attribute *val* of the statechart is represented by the variable `val` in the cTLA process `Adapter`. Moreover, the cTLA process representing an active object contains a queue `qu` handling concurrent calls. A cTLA action describing a UML transition with a trigger (e.g. `dequeueCall`), has an additional conjunct (`head(qu)` $\in$ `callAdapterEvents`) which checks if the appropriate event is

contained in `qu`. A state variable `sync` used to block an object initiating a synchronized calls. Moreover we introduced the state variable `lifecycle` describing the lifecycle state of an object [3]. Possible values of `lifecycle` are `unborn`, `alive`, and `dead`. Unborn and dead objects are neither able to compute a result nor to take part in an interaction. Moreover, for every pin of an action we added a set of state variables. One variable is used to store the data transferred via the pin. For the sake of simplicity, we restrict the range of pin types to simple data types. Another variable states if the pin is currently filled by a data or control value.

In the UML2.0, an activity is modelled by a kind of Petri Net. Since a Petri Net corresponds with a state transition system, it can be transformed relatively easily to corresponding cTLA variables and actions. As outlined in Sec. 4, we currently restrict ourselves to so called *BasicActivities* (cf. [27]) using actions which are connected by their object and control flow edges permitting no concurrent flow between actions. This basic level of activity modelling supports the description of traditional sequential flow charts including decisions and merges. Moreover, we assume that a well formed activity is completed by an *ActivityFinalNode* which is connected to the final actions of an activity. Thus, in cTLA one can model a transition linked with an activity by two cTLA actions. The first describes the control flow from the previous state of the transition to the initial state of the activity while the other forms the link between the *ActivityFinalNode* and the transition's next state.

The UML actions are transformed within the context of the transformation of their corresponding activities. An action is extended by statements which are responsible for removing data and control values from the state variables representing the input and output pins as well as the control edges of an action (cf. Fig. 5). The transformation of an UML action is handled based on its scope. Read and Write Actions as well as Computation actions have only local scope whereas invocation actions have access to other objects. For each UML action with local scope it is sufficient to introduce a new cTLA action transforming the semantics of the UML action and describing its access to its input and output pins as well as its incoming and outgoing control edges. An example in the cTLA process `Adapter` is the Computation Action `processCall` which modifies the variable `val` coupled with the action's input and output pins.

Read Write Actions reading or writing structural features can also be transformed into a single cTLA action each. This action models the assignment of a value from the corresponding attribute to the corresponding pin and vice versa. Moreover, CreateObjectActions and DestroyObjectActions are supported by

[3] In Fig. 5, we omitted this and the remaining variables in order to save space.

cTLA actions that change the value of the state variable `lifecycle` if the creation and destruction of objects is handled in the corresponding UML model. The cTLA action which creates an object sets the value of the state variable `lifecycle` to `alive` if it has the value `unborn`. An additional action is used to change the value of `lifecycle` to `dead` if an object has to be destroyed and the value of `lifecycle` is `alive`.

Invocation actions have to reflect that they are coupled with other actions in a peer object. In consequence, the corresponding cTLA action will be coupled with another action in a different process modelling the peer behavior. For instance, in a cTLA system specification a cTLA action representing a *CallOperationAction* of a calling object will be coupled with the cTLA action modelling the corresponding *AcceptCallAction* of the called object, which enqueues pending calls to the queue of the called object. For instance, in the system specification `concreteComposition` in Fig. 6 the action `getCallSensor` models an operation call of the adapter to the sensor. Here, the process action `callControl` of the adapter describing a *CallOperationAction* is coupled with the process action `getCallAdapter` in the sensor specifying the corresponding *AcceptCallAction*. The two linked process actions have to carry identical action parameters modelling the arguments of the operation call.

Similar to an operation call, the corresponding operation reply is also modelled by two coupled cTLA actions. In the UML, however, an incoming reply transmission is accepted by the same CallOperationAction which also triggered the initial call. Since this non-atomic behavior cannot be specified by a single cTLA action, we describe CallOperationActions by two cTLA actions modelling the execution of an operation call resp. the acceptance of the reply transmission. In the process `Adapter` the actions `callControl` and `callControlReply` are both used to model a single CallOperationAction.

7 Transformation of UML 2.0 sequence diagrams

Each UML sequence diagram is transformed into one instance of the cTLA process type *SequenceDiagram* which is depicted in Fig. 7. In particular, the transformation tool computes the set of all total traces permitted by the sequence diagram and instantiates the generic module parameter *SetOfTraces* with it. In the cTLA process type, a trace is represented as a sequence of strings. Each string represents an occurrence of an object and is composed of the object name, a dot used for separation and the name of the action associated with the event occurrence (e.g., "O2.getValue" represents the CallOperationAction *getValue* on the lifeline of *O2*). In order to model the sequence diagram's behavior, we, moreover, use the set of all legal partial traces which is specified by the constant *SetOfPossibleTraces*. The set of partial traces cor-

```
PROCESS SequenceDiagram(SetOfTraces : SUBSET(Sequence))
CONSTANT
 setOfPossibleTraces : SUBSET(SetOfTraces) ≜ UNION {
         {SubSeq(d, 1, 2) : d ∈ setOfTraces},
         {SubSeq(d, 1, 3) : d ∈ setOfTraces}, ... };
VARIABLES
 currentTrace : Sequence;

INIT ≜  ∧
        currentTrace = ≪ ≫;
ACTIONS
    possibleMessageOfSD(EventOccurrence1, EventOcurrence2) ≜
        currentTrace ∘
        ≪EventOccurrence1, EventOccurence2≫
        ∈ setOfPossibleTraces ∧
        currentTrace′ = currentTrace ∘
        ≪EventOccurence1, EventOccurence2≫ ∧
END
```

Fig. 7. Process type `SequenceDiagram`

responds to the union of all subsequences of *SetOfTraces* which are described by the cTLA operator *SubSeq*[4]. Moreover, the cTLA process introduces a variable *currentTrace* modelling the partial trace of events which already took place during the system execution. An interaction between two UML objects or between two UML actions of the same object is specified by the action *possibleMessageOfSD*. Its action parameters *EventOccurrence1* and *EventOccurrence2* describe the events taking place during firing the interaction. The enabling condition of *possibleMessageOfSD* states that the action may only be executed if the two events corresponding to the interaction lead to a partial trace of occurred events which is permitted by the sequence diagram. The new trace of events is stored by the variable *currentTrace*.

In the UML, sequence diagrams are used to describe the interaction of objects. Thus, they form a link between the state chart diagrams describing the behavior of the objects. In consequence, we use the cTLA representation of the sequence diagrams in order to model the interaction of the cTLA processes specifying the statecharts. Therefore, in cTLA processes modelling a complete system the cTLA action *possibleMessageOfSD* is coupled with the cTLA actions of the process instances representing the corresponding UML actions and their events in the statecharts. In order to achieve correct compositions, we assume that the state machines are statically bound. Thus, the alphabets of the combined state machines can be mapped into each other.

In the example system outlined in Fig. 6, *possibleMessageOfSD* is coupled with the actions *callControl* of the process modelling the adapter and *getCallAdapter* of the cTLA process specifying the concrete sensor. Each of these

[4] The first argument of *SubSeq* describes the orignal trace d, the second argument indicates which string of d is the first element of the subtrace, and the third argument refers to the length of the subtrace.

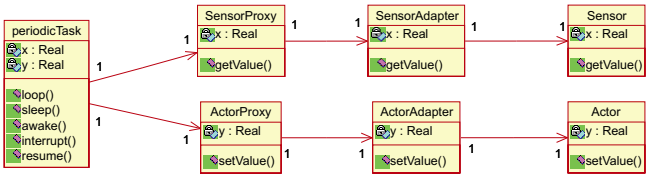

Fig. 8. The class model of the detailed Controller

two actions contains an action parameter *actionInfo1* resp. *actionInfo2* describing the event occurrence related with the specified UML action. Since the action parameters have to be equal to the parameters of action *possibleMessageOfSD*, the two UML actions of the adapter and the concrete sensor may only take place if the related event occurrence can be tolerated by the sequence diagram.

8 Verification

As an example for outlining the refinement verification of systems, we use a refinement pattern for developing a distributed controller of technical processes. A refinement pattern describes the correct refinement of an analysis pattern by adding design-relevant information which is modelled in design patterns and classes. The particular refinement pattern consists of an analysis pattern describing the scenario of a controller, a sensor, and an adapter without including the communication aspects and a design pattern for creating the real controller. The UML class diagram in Fig. 1 describes the analysis pattern. It consists of an abstract controller class which in periodic loops asks an abstraction of a sensor for certain values of the technical process. Based on this value, a setting of the actor, which is also modelled abstractly, is computed and forwarded to the actor. In contrast, the design pattern contains more detailed objects describing the controller, the sensor, and the actor. Moreover, it contains additional objects realizing the distributed communication between the controller and the sensor resp. actuator. At the node executing the controller, the sensor is represented by a special proxy object which maintains the data transfer to the server site. Here, the communication is realized by an adapter object which directly interacts with the proxy. Likewise, the communication with the actor is also performed by means of a proxy and an adapter object. The UML class diagram of the design pattern is listed in Fig. 8.

The goal of our verification is to prove that the refinement pattern performs only model transformations where the behavior of the derived more detailed UML specification does not contradict to the behavior of the original abstract UML description. For this purpose, we transform both UML models to cTLA specifications as outlined in Sec. 6 and 7. Thereafter, we perform

```
qu ≙ IF (pS.qu ≠ empty ∨ pS.sProxy ∈ {"deqCall", "procCall"} ∨
         aS.qu ≠ empty ∨ aS.sAdapter ∈ {"deqCall", "procCall"} ∨
         cS.qu ≠ empty
     THEN ≪SensorCall≫ ELSE empty;
```

Fig. 9. Extract of the refinement mapping for the example proof

a cTLA refinement verification proving that any state sequence modeled by the cTLA representatives of the design pattern is also state sequence specified by the cTLA processes describing the more abstract analysis pattern. This verification corresponds directly to our proof goal.

As an example for a cTLA specification, modeling a part of the design pattern, we use an instance of the cTLA process `Adapter` (cf. Fig. 5) specifying the adapters of the sensor and actor in the design pattern. An adapter run is activated by receiving a call event from the proxy which is enqueued to the message queue (cTLA action `getCallProxy`). Thereafter, the event is read from the message queue and stored in the variable `val` (action `dequeueCall`). Afterwards, the event may be processed (e.g., certain communication-related computations are performed). This is modelled in the action `processCall` by means of the function `processFct` which is defined as a cTLA action parameter. In the next step, the processed value is handed over to the sensor (action `callControl`). Similarly, the other cTLA actions model the handling of the return of the call event which is received from the sensor and sent back to the proxy. The actions modelling UML actions triggered by the adapter itself are provided by weak fairness assumptions in order to guarantee lively behavior. A weak fair action must not be enabled continuously without eventually being executed.

The various cTLA processes are composed to a cTLA system specification Φ of the detailed design pattern objects and Ψ of the abstract analysis pattern objects. Fig. 6 depicts the compositional cTLA process `concreteComposition` specifying Φ. The coupling is performed by defining system actions joint from process actions. For instance, the action `callControl` of the cTLA process instance `aS`, specifying the adapter of the sensor, is coupled with the cTLA action `getCallAdapter` of the cTLA process `cS` modeling the sensor to the system action `getCallSensor` which describes a method call of the adapter to the sensor object. The verification of the refinement pattern is performed by a TLA-based logic deduction proof of the implication $\Phi \Rightarrow \Psi$.

Since the state spaces of Φ and Ψ are different, we have to define a so-called refinement mapping (cf. [1]) from the states of Φ to those of Ψ. Here, the compositional structure of the specifications facilitates the understanding of the system behavior and, in consequence, fosters the detection of a suitable refinement mapping. In particular, the abstract controller, sensor, and actor are similar to their detailed counterparts and the mapping of their respective

variables is straightforward. The mapping of the proxy and adapter variables is a little more subtle since they do not have counterparts in the abstract specification Ψ. A glance into Ψ, however, shows that, for instance, a `getValue` call of the abstract controller leads to the enqueueing of a call event into the message queue of the abstract sensor. In contrast, in Φ the call of `getValue` leads to a sequence of operations in which the call event is forwarded through the proxy and adapter objects before it can finally be enqueued into the message queue of the detailed sensor. Therefore, one should map all states of Φ, in which the proxy and adapter objects are involved in the transmission of the call event to the sensor, to the state of Ψ where the call event is waiting in the message queue of the abstract sensor. Likewise, one can treat the reply of the call event as well as the events for manipulating the actuator. In consequence, the refinement mapping for the variable `qu` describing the message queue of the abstract sensor [5] is defined as outlined in Fig. 9. When in the detailed system a call message is in the queues of the proxy, adapter, or the sensor resp. the proxy or adapter are processing or transmitting the call message (their main states are either in the state `"decCall"` or `"proveCall"`, the message queue `qu` of the abstract sensor contains a call message of type `<<SensorCall>>`. Otherwise, `qu` is empty. By means of the refinement mapping, we can replace the variables of the cTLA processes in the abstract system specification Ψ by those of Φ achieving the modified specification $\overline{\Psi}$. Lamport proved in [23] that the proof $\Phi \Rightarrow \overline{\Psi}$ is sufficient to verify the refinement between Φ and $\overline{\Psi}$.

In the example proof we verify at first, that the safety properties (cf. [2]) of Φ and $\overline{\Psi}$ are consistent. Here, we have to prove that the initial condition of Φ implies the initial condition of $\overline{\Psi}$. Moreover, every system action of Φ has to imply either a system action or a stuttering step of $\overline{\Psi}$. As an example, we look at the system action `adapterDequeueCall` of Φ which is coupled from the process action `dequeueCall` in the adapter (cf. Fig. 6) while the other processes of Φ perform a stuttering step. `adapterDequeueCall` is mapped to a stuttering step in $\overline{\Psi}$ since according to the refinement mapping the variable `abstractSensor` is not altered by the execution of the system action and the other variables of $\overline{\Psi}$ do not depend on the variables of the cTLA process `Adapter`. Since in our example, Φ consists only of a relatively small number of reachable states, the proof of all safety properties could be performed automatically by application of the model checker TLC [24,33] which finished the proof within 4 seconds on a standard PC.

Liveness verifications are the second part of the refinement proof. By these proof steps we guarantee, that the detailed system Φ fulfills the liveness con-

[5] The complete refinement mapping as well as the specification and proof steps of the refinement verification are depicted in the WWW (URL: `ls4-www.cs.uni-dortmund.de/RVS/P-OORT/ExampleProof`).

straints of the abstract system $\overline{\Psi}$. As an example, we look on the system action `abstractSensorDequeueCall` of $\overline{\Psi}$ which models the dequeueing of call events from the message queue in the abstract sensor. This weak-fair action is realized by the action `sensorDequeueCall` in Φ. Since `sensorDequeueCall` is also provided by a weak fairness constraint, we have only to prove that, whenever `abstractSensorDequeueCall` is constantly enabled, also `sensorDequeueCall` will eventually be enabled. Moreover, `sensorDequeueCall` has to be enabled until being executed. As outlined above, however, `abstractSensorDequeue-Call` is enabled in all states when in the design pattern a *getValue* call event is forwarded through the proxy and adapter objects. Thus, we have to prove that the call event passes the proxy and the adapter in a lively manner and will be eventually enqueued in the message queue of the detailed sensor object enabling the action `sensorDequeueCall`. We have to find a sequence of system states which are passed in the transmission process of the call event. For example, after passing the event from the proxy to the adapter, the system in succession passes states in which the variable `sAdapter` gets the values `"deqCall"`, `"procCall"`, and `"called"` (cf. Fig. 5). When it is in the state `"called"`, the action `sensorDequeueCall` is enabled. Finally, we have to prove that Φ may not last forever in a state of the sequence but eventually proceeds to a successor state. Here, we can apply the fairness constraints of Φ. For instance, the weak fairness assumption of the action `dequeueCall` in the process `Adapter` guarantees that the action will eventually be executed if `sAdapter` is in the state `"init"` passing to the state where `sAdapter` contains the value `"deqCall"`. This proof was also performed by TLC but due to the large amount of states to check, the proof run lasts for about 80 seconds. By application of TLC we proved at first that each initial state of Φ is mapped to an initial state of $\overline{\Psi}$ and each state change of Φ corresponds either to a state change or to a stuttering step of $\overline{\Psi}$. Moreover, we proved that each state change in $\overline{\Psi}$ enforced by a fairness assumption is fulfilled by the fairness-enforced state changes of Φ. According to Abadi's and Lamport's refinement mapping proof [1], this is sufficient to verify that Φ implies $\overline{\Psi}$.

9 Concluding Remarks

We discussed that it is possible to formalize UML 2.0 diagrams on the base of cTLA with an emphasis on new features of the upcoming standard. Moreover, we showed how to prove properties of UML models applying the TLA proof rules and the according calculus. Based on the experience of one author in a software development enterprise, we are convinced that the specification language cTLA is a suitable means to formalize actions and the different kinds of diagrams of UML 2.0. Future research will concentrate on the following

goals. Firstly, existing tools which are used for the transformation of UML 1.4 models into cTLA will be adapted (e.g., the adaption to the new action semantics and to sequence diagrams based on MSCs). Furthermore, the formalization of sequence diagrams and activity diagrams using cTLA has to be broadened. The *CombinedFragments* of sequence diagrams supporting several different *InteractionOperators* offer new perspectives for research. Moreover, the composition of actions in an activity should be investigated. We are going to extend our approach to symbolic model proving (cf. [4]) which is a combination of model checking and theorem proving. Model proving seems to be a promising approach for the automatic verification of software. Finally, we will continue to provide analysis, design, and refinement patterns for several application domains.

References

[1] Martín Abadi and Leslie Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991. http://www.research.digital.com/SRC/staff/lamport/bib.html.

[2] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[3] R. J. R. Back and R. Kurkio-Suonio. Decentralization of process nets with a centralized control. *Distributed Computing*, 3(2):73–87, 1989.

[4] Sergey Berezin. *Model Checking and Theorem Proving: A Unified Framework*. PhD thesis, Carnegie Mellon University, 2002.

[5] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Longman, 1999.

[6] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, First International Symposium (FMCO 2002)*, LNCS 2852, pages 71–98, Leiden, 2003. Springer-Verlag.

[7] Martin Gogolla and Francesco Parisi-Presicce. State Diagrams in UML — A Formal Semantics using Graph Transformation. In Manfred Broy, Derek Coleman, Tom Maibaum, and Bernhard Rumpe, editors, *Proceedings of the ICSE'98 Workshop on Precise Semantics of Modeling Techniques (PSMT'98)*, Technical University of Munich, Technical Report TUM-I9803, pages 55–72, 1998.

[8] G. Graw, P. Herrmann, and H. Krumm. Verification of UML-based real-time system designs by means of cTLA. In *Proceedings of the 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC2K)*, pages 86–95, Newport Beach, 2000. IEEE Computer Society Press.

[9] Günter Graw and Peter Herrmann. Verification of xUML Specifications in the Context of MDA. In Jean Bezivin and Robert France, editors, *Workshop in Software Model Engineering (WISMEUML'2002)*, Dresden, 2002.

[10] Günter Graw, Peter Herrmann, and Heiko Krumm. Composing object-oriented specifications and verifications with ctla. In Uwe Nestmann, editor, *Proceedings of the Workshop on Semantics of Objects as Processes (SOAP'99)*, BRICS Notes Series NS–99–2, pages 7–22, Lisbon, June 1999.

[11] Günter Graw, Peter Herrmann, and Heiko Krumm. Constraint-Oriented Formal Modelling of OO-Systems. In *Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 99)*, pages 345–358, Helsinki, June 1999. Kluwer Academic Publisher.

[12] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[13] Øystein Haugen. MSC-2000 Interaction Diagrams for the New Millennium. *Computer Networks*, 35(6):721–732, 2001.

[14] Peter Herrmann. Formal Security Policy Verification of Distributed Component-Structured Software. In Hartmut König, Monika Heiner, and Adam Wolisz, editors, *Proceedings of the 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'2003)*, LNCS 2767, pages 257–272, Berlin, September 2003. Springer-Verlag.

[15] Peter Herrmann and Heiko Krumm. Modular Specification and Verification of XTP. *Telecommunication Systems*, 9(2):207–221, 1998.

[16] Peter Herrmann and Heiko Krumm. A Framework for Modeling Transfer Protocols. *Computer Networks*, 34(2):317–337, 2000.

[17] Peter Herrmann and Heiko Krumm. A Framework for the Hazard Analysis of Chemical Plants. In *Proceedings of the 11th IEEE International Symposium on Computer-Aided Control System Design (CACSD2000)*, pages 35–41, Anchorage, 2000. IEEE CSS, Omnipress.

[18] Peter Herrmann, Heiko Krumm, Olaf Drögehorn, and Walter Geisselhardt. Framework and Tool Support for Formal Verification of High Speed Transfer Protocol Designs. *Telecommunication Systems*, 20(3,4):291–310, 2002.

[19] Carsten Heyl, Arnulf Mester, and Heiko Krumm. ctc — A Tool Supporting the Construction of cTLA-Specifications. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecture Notes in Computer Science, pages 407–411. Springer-Verlag, 1996.

[20] ISO. *LOTOS: Language for the temporal ordering specification of observational behaviour*, International Standard ISO 8807 edition, 1989.

[21] Kennedy Carter. *Action Semantics for the UML*, 2003. Available via WWW: www.kc.com/as_site/home.html.

[22] Anneke Kleppe, Wim Bast, and Jos Warmer. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison Wesley, 2003.

[23] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[24] Leslie Lamport. *Specifying Systems*. Addison-Wesley, 2002.

[25] Johan Lilius and Ivan Porres Paltor. Formalising UML state machines for model checking. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, LNCS 1723, pages 430–445. Springer, 1999.

[26] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.

[27] OMG. *UML: Superstructure v. 2.0 – Third revised UML 2.0 Superstructure Proposal*, omg doc# ad/03-04-01 edition, 2003. Available via WWW: www.u2-partners.org/uml2-proposals.htm.

[28] Carlos Rossi, Manuel Enciso, and Inmaculada P. de Guzmán. Formalization of UML State Machines using Temporal Logic. *Software and Systems Modeling*, November 2003. Online first.

[29] Bernhard Rumpe. Executable Modeling with UML. A Vision or a Nightmare? In *Issues & Trends of Information Technology Management in Contemporary Associations*, pages 697–701. Idea Group Publishing, Hershey, London, Seattle, 2002.

[30] Timm Schäfer, Alexander Knapp, and Stephan Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):13 pages, 2001.

[31] A.J.H. Simons. On the Compositional Properties of UML Statechart Diagrams. In *Proceedings of the Conference of Rigorous Object-Oriented Methods 2000*, 2000. Available via WWW: http://ewic.bcs.org/conferences/2000/objectmethods/papers/paper8.pdf.

[32] U2 Partners. *Unified Modeling Language: Infrastructure Version 2.0*, 2003. Available via WWW: www.kobryn.com.

[33] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA+ Specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME '99)*, Lecture Notes in Computer Science 1703, pages 54–66. Springer-Verlag, 1999.