Modeling and Verification of Manufacturing Systems: A Domain-Specific Formalization of UML

Stephan Flake
C-LAB, Paderborn University, Fuerstenallee 11, 33102 Paderborn, Germany
email: flake@c-lab.de

ABSTRACT
This article presents a formalization of parts of the Unified Modeling Language (UML) w.r.t. the domain of modeling time-dependant manufacturing systems. The formalization approach is based upon a set of identified general formalization steps. We investigate the applicability of UML class diagrams, Statecharts, and UML's textual Object Constraint Language (OCL) to model manufacturing systems and specify required time-dependent system properties. We then define a formal semantics for the utilized UML modeling elements and the extensions we have to make due to the considered domain. Corresponding mappings to formal languages allow to verify whether a domain-specific UML model satisfies required temporal properties.

KEY WORDS
Object Constraint Language (OCL) – UML Statecharts – Formal semantics

1 Introduction

The Unified Modeling Language (UML) provides several kinds of graphical diagrams for *modeling* different views on a system [9]. Furthermore, the textual Object Constraint Language (OCL) is part of UML. OCL is used to formulate restrictions over values of a given model in form of invariants for objects and pre- and postconditions for operations.

In past years, *formal verification methods* like model checking have been successfully applied in some application domains, e.g., electronic systems design and protocol verification. Model checking takes a model specified by means of state transition systems and a property specification expressed by temporal logic formulae as an input. It is then possible to automatically verify whether the model satisfies the required properties. Ongoing research investigates application of this verification technique to models of (large) software systems.

For time-dependent systems, additional timing aspects complicate the task of developing a correct model, and corresponding time-related requirements have to be modeled (w.r.t. UML) and verified (w.r.t. model checking). Both, UML and model checking have already been applied in the domain of time-dependent systems, e.g., UML-RT [15] or real-time model checking with UPPAAL[1].

[1] http://www.uppaal.com

Our approach aims to apply model checking techniques to UML designs in the domain of time-dependent manufacturing systems. In contrast to other approaches that apply model checking to UML models, we build upon *existing OCL concepts and syntax* for specifying requirements that regard the dynamic behavior of UML models.

As a prerequisite, some effort is required to tailor UML's behavior-related diagrams to the domain of time-dependent systems. In our work, we focus on Statechart diagrams that are used to model the reactive behavior of objects. We investigate which of UML's Statechart concepts can be omitted, which concepts need to be newly introduced for our domain, and how a suitable time-based formal semantics can be defined. The whole formalization is embedded into a more general formalization process that is also described in this article.

The remainder of this article is structured as follows. Related work is discussed in Section 2. In Section 3, we present the formalization steps of our approach. In Section 4, the formalization approach for the domain of modeling manufacturing systems is described. Finally, Section 5 concludes the article with an outline of our experiences.

2 Related Work

Related work is found in the areas of (a) formalizing UML Statecharts, (b) behavioral real-time modeling with UML, and (c) formal semantics of standard OCL and temporal extensions. Due to space limitations we can here only list some of the relevant works in each case.

Various approaches have been published to define the semantics of UML Statecharts (see [1] for an overview). As the UML standard leaves open some semantic variation points (e.g., the dispatching mechanism to select an event from the implicit event queue), each formalization has to make choices to provide a unique definition. All approaches that we know of define a formal semantics over a restricted set of Statechart modeling elements. For instance, so-called *elapsed-time events* that trigger a transition after a particular passed time have not been regarded in any formalization yet.

We are less interested in modeling real-time system architectures and therefore do not further discuss well-known approaches like RT-UML or the UML-RT Profile which are concentrating on structural modeling aspects.

Instead, we focus on modeling timed behavior of objects. In this context, the UML standard notation currently provides two ways to specify timing properties, (a) for messages in Sequence Diagrams by timing expressions and (b) for state transitions in Statecharts by elapsed-time events [9, Section 3.77.2]. More recently, the *UML Profile for Schedulability, Performance, and Time* has been adopted by the OMG. Though it provides a common framework of time-related concepts, it also has just limited means concerning the specification of temporal behavior. Independently, a number of extensions of OCL have been proposed to enable modelers to specify temporal properties over occurrences of events and their timing properties, such as deadlines, delay times, and response times, e.g., [2]. Though, state-related temporal properties cannot be defined so far.

There is not yet a formal semantics of OCL provided in UML 1.5, but the latest OCL 2.0 submission [8] has adopted a formal semantics developed by M. Richters [10]. In that document, a metamodel for OCL is defined and a semantics is given by a formal description of class diagrams (so-called *object models*) and a meaning function that maps OCL expressions to a semantic domain, i.e., objects and basic data values. Nevertheless, there are still deficiencies w.r.t. integration of Statecharts. Although there is a standard operation called `oclInState()`, there is no corresponding semantics provided, as object models lack of a Statechart description with states and state configurations.

Formalizations of UML to perform model checking is also addressed by other approaches (e.g., [3]), but these either do not consider time or have only limited means for expressing temporal properties due to the chosen verification tool. We take an approach that enables property specification by means of an extended version of OCL that is consistent with existing OCL concepts and syntax [6].

3 Formalization Steps

Our approach follows an iterative formalization process that takes the particular circumstances concerning UML into account. On the one hand side, UML has so many concepts that a complete formalization is not feasible, and on the other hand side, domain-specific modeling usually requires additional concepts that are not covered in the UML standard. We therefore propose the following stepwise formalization approach. It can also be regarded as a more general guideline to formalizing parts of UML towards a particular domain.

Step 1: Select a subset of UML diagrams. The large number of publications concerning the formalization of (parts of) UML demonstrate that it is already a complex task to give a precise semantics to even a single kind of UML diagram. But eventually, formalization efforts will more and more focus on the relationship and interdependencies of different UML diagrams. We therefore regard the selection of a subset of UML diagrams as a first step of the formalization process. This set of diagrams highly depends on the regarded application domain, e.g., modeling of real-time systems, business process modeling, or consistency analysis among UML diagrams.

Our application domain is modeling of manufacturing systems, and we choose class diagrams for structural and Statecharts for behavioral modeling. For additional property specification, we make use of OCL.

Step 2: Apply syntactical restrictions. As UML provides a variety of modeling concepts within the diagrams chosen in Step 1, further syntactical restrictions have to be applied prior to formalization. By doing this, one might already have a particular semantic target domain in mind, though we do not require this yet in this step (cf. Step 5). The outcome of this step is a (preliminary) source domain referred to as $UML|_{dom}$. Note that at this stage $UML|_{dom}$ has only an informal semantics.

Step 3: Clarify open semantic issues. UML often relies on under-specification and non-determinism. But in many cases, semantic issues are (unintentionally) not considered in the official UML specification. Regardless of intentional or unwanted semantic variation points, all unclear issues in $UML|_{dom}$ have to be identified. In each case, a decision has to be made that clearly describes how to resolve that semantic issue in a unique manner.

Note that this does not imply that non-determinism is completely eliminated. In contrast, non-determinism can even be explicitly allowed. For example, in some domains it might be appropriate for analysis purposes that the event dispatching mechanism in Statecharts is represented by a random function, i.e., a non-deterministic choice out of a set of events.

Precision at this stage is still established by means of (informal) natural language. In the ideal case, the resulting documentation enables modelers to understand the semantics of $UML|_{dom}$ without further knowledge of the actual formalization, i.e., the semantic target domain *as well as* the mapping of $UML|_{dom}$ to that domain. However, this aspect is seen controversially in the formal methods community and cannot generally be assumed. Nevertheless, initiatives like the *precise UML* group[2] aim to provide a semantics of UML that eventually reduces the efforts needed in this formalization step.

Step 4: Introduce domain-specific concepts. Due to some concepts that are special for the regarded domain, the chosen $UML|_{dom}$ might not provide sufficient concepts yet. These have to be added, either by UML "lightweight" extension mechanisms, (i.e., *UML Profiles* with stereotypes, tagged values, and constraints), or by "heavyweight" extensions that introduce completely new language concepts on the UML metalevel (i.e., the language definition or abstract syntax). However, a formal semantics of such new concepts must still be defined in any case.

For instance, for the purpose of model execution analysis, we might want to be able to specify potential timing intervals for operation executions in manufacturing sys-

[2] http://www.puml.org

tems. Though UML allows to define corresponding syntactical extensions, giving an actual semantics to these is out of the scope of UML. Consequently, when $UML|_{dom}$ is to be syntactically extended, Step 3 has to be revisited, as new semantic issues have occurred that need clarification.

Step 5: Mapping to a semantic domain. This step is about the actual formalization, i.e., (a) choosing an appropriate semantic domain $Target$ and (b) defining a mapping (or: meaning function) M_{dom} from $UML|_{dom}$ to $Target$ that complies to the restrictions informally gathered in Step 3. The semantic domain is preferably and most usually an already existing formal language for which analysis tools are available. Note that there is probably no existing formal language that is directly suitable as a semantic domain.[3] In this case, Step 2 has to be revisited to fix that problem.

Note. The proposed steps have to be interpreted as being iterative and (partly) branching. At certain stages, one might identify the need to add or leave out concepts of $UML|_{dom}$ and then has to review the syntactical and semantical effects on $UML|_{dom}$ to adjust the mapping to the semantic domain. Moreover, when the formalization can naturally be divided, the steps might partly be carried out in parallel for distinct parts, especially in early stages. However, these parallel parts are then to be formally integrated, which is a topic out of scope of this article.

4 Domain-Specific Formalization

Figure 1 gives an overview of our formalization approach in the domain of modeling manufacturing systems. We decomposed the whole approach into four different activity parts (indicated by gray boxes). Figure 1 also illustrates the dependencies among the different activities.

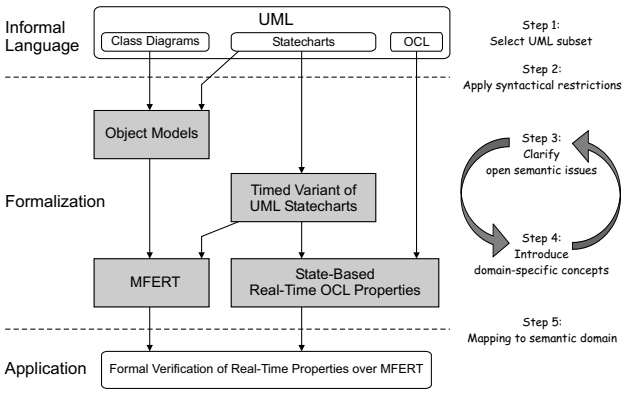

Figure 1. Overview of the Formalization Approach

First, we integrate the notational concepts of UML Statecharts into the existing formal description of class diagrams by M. Richters [10]. Basically, the resulting so-called *extended object models* base upon a set-theoretic definition of the UML metamodel parts for class diagrams and Statecharts. In parallel, we develop a timed variant of UML Statecharts. Note that this activity is positioned in Figure 1 as a more domain-specific task, because timing issues are a non-standard concept of UML Statecharts, while extended object models basically concern standard UML. Integrating these two formal models and applying further restrictions leads to our domain-specific modeling notation called MFERT.

MFERT is an acronym for "Modell der FERTigung" (German for "Model of Manufacturing") and provides means for specification of planning and control assignments in manufacturing processes [13]. For MFERT, we define a mapping M_{MFERT} to the semantic domain of labeled transition systems called *I/O-Interval Structures* [12].

Additionally, we define an extension of OCL (called RT-OCL) that allows for specification of temporal state-based properties. For these OCL expressions, we provide another mapping $M_{RT\text{-}OCL}$ to formulae in a temporal logic called *Clocked Computation Tree Logic* (CCTL) [11]. This means that we actually have two meaning functions, i.e., M_{dom} consists of M_{MFERT} and $M_{RT\text{-}OCL}$ (Fig. 2).

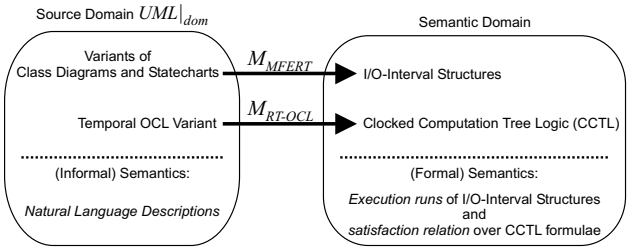

Figure 2. Semantic Domain Mapping

A semantics for the combination of MFERT notation and temporal state-based OCL expressions is then automatically available, as the two formal target languages already have a well-defined formal relationship, i.e., CCTL formulae have a well-defined semantics over *execution runs* of I/O-Interval Structures. In this context, the model checking tool RAVEN [11] is able to verify whether a model (a set of I/O-Interval Structures) satisfies a given property (a CCTL formula).

In the following subsections, we discuss the individual formalizations in some more detail, although we cannot give formal definitions due to a lack of space.

4.1 Extended Object Models

The first formalization part considers UML class diagrams and Statecharts, mainly on the syntactical level. For an appropriate formalization, we have to break the circular UML language definition approach and provide a corresponding mathematical definition. The syntax of class diagrams has already been defined in an appropriate set-theoretical way in [10]. We add some more object-related concepts to that definition, such as signals that instances of a class may receive. But most importantly, we integrate a formal syntax

[3] Other formalization approaches therefore suggest to first select a semantic domain and then formalize UML correspondingly, e.g. [4].

description of Statecharts. Almost all concepts of UML Statecharts are included in that formal description called *extended object models*; we only omit *StubStates* and *SubmachineStates*, as these are directly substitutable by the actual composite states they represent. The complete syntactical definition is straightforward and can be derived from the UML metamodel.

Semantically, an interesting aspect arises from integrating Statecharts into class diagrams, namely, inheritance of behavior specified by Statecharts. Literature distinguishes between different kinds of consistent behavioral inheritance (e.g., weak, strong, or observation consistency [14]). These definitions make use of the dynamic execution of Statecharts by means of *traces*, which are execution runs through (the processes derived from) Statecharts. In contrast, UML provides an informal description of three inheritance policies called subtyping, strict inheritance, and general refinement [9, Section 2.12.5.3]. The latter two are introduced because of coding issues and cannot be compared with the conceptual behavioral inheritance policies as known from corresponding literature.

Nevertheless, one clear notion of behavioral inheritance has to be provided, probably not yet for the general description of extended object models, but at a later stage when the domain-specific modeling language $UML|_{dom}$ is completely defined.

4.2 Time-Based Statecharts

In this subsection, we discuss the behavioral modeling part in more detail, as quite a few design choices have to be made.

First, according to formalization Step 2, we have to select a reasonable sublanguage of the extensive UML Statechart syntax, and for each of the omitted language elements we here justify why we do not support it in our approach. The semantics of the chosen sublanguage is informally described along the lines of the so-called *run-to-completion* step (RTC-step), i.e., one complete reaction of an object on a received and dispatched event (cf. Step 3). On the other hand side, we have to extend the UML Statechart syntax to capture additional timing issues (domain-specific additions, cf. Step 4). In particular, we need concepts to be able to specify non-deterministic timing intervals attached to operations and time events. The main difficulty here is that the UML standard does not make any assumptions about elapsing time. Thus, a new semantics must be provided for RTC-steps that has some underlying inherent notion of time.

We here only provide an overview of our timed Statechart variant. A complete definition and mapping to labeled transition systems (cf. Step 5) can be found in [5].

General Syntactical Restrictions. Among the different state notions within the UML, only *composite states*, *simple states*, *history states*, *final states*, and *initial PseudoStates* are considered. The following kinds of states can generally be simulated by other means and thus do not have to be considered as separate concepts in a formalization. *StubStates* and *SubmachineStates* are just used for syntactical convenience and can be substituted by the actual composite states they represent. *SynchStates* are used to model synchronizations among orthogonal regions. Basically, they can be simulated by additional internal signals. *Junction PseudoStates* are splits or merges of transitions and can simply be replaced by specifying corresponding simple state-to-state transitions. *Choice PseudoStates* are dynamic conditional branches that can be simulated by adding intermediate states, provided that visiting these intermediate states does not take any additional time (as it is possible in the RTC-step semantics). In our timed semantics, though, each transition consumes at least one time unit, and therefore this kind of states cannot be directly supported.

There are some more concepts in UML Statecharts that we can explicitly abstract from in a formalization. *Internal Transitions* do not trigger entry- and exit-actions and are sometimes even seen as unnecessary, as internal transitions are actually modeling behavior that belongs to a substate. Though standard UML allows parameters not only for operation calls but also for asynchronous signals, we do not regard *event parameters*. Note that these can be simulated by specifying a set of parameterless events, built based upon the cross product of the parameters' value sets. *Deferred events* can be simulated by regenerating them as often as they are to be deferred.

Many formalizations of UML Statecharts do not allow *interlevel transitions*, as they do not comply to compositionality. While this might apply to approaches that investigate refinement issues, one cannot on the other hand deny that certain interlevel transitions ease the task of modeling, as they simply reduce graphical complexity. Thus, one approach to still support interlevel transitions is to carefully select and allow certain kinds of interlevel transitions and provide a corresponding semantics. Note that UML already restricts interlevel transitions by means of well-formedness rules specified in OCL. In the same way, additional rules to further restrict the set of allowed interlevel transitions can easily be formulated for a particular domain. Finally, *local variables* of UML Statecharts can be omitted in formalizations, as these can be simulated by attributes defined in the class the Statechart is attached to.

Statechart Syntax Definition. As a result from the previous considerations, the syntax of a UML Statechart can essentially be defined the tuple

$$\langle\ S, H, shallowHistory, deepHistory, init, final, EVTS, GUARDS, ACTS, TR, substates, defaultHistory, entry, exit, doActivity\ \rangle.$$

We here only informally describe the elements of that tuple and refer to [7] for more details.

S is a set of states. It is composed of the disjoint sets of simple and composite states. Composite states in turn are either sequential (Xor-states) or orthogonal (And-states). H is a set of history states, and functions $shallowHistory$ and $deepHistory$ determine for a given composite state its (potential) history state.

$EVTS$, $GUARDS$, and $ACTS$ are sets of events, boolean conditions, and actions, respectively. Appropriate expression languages must be available to formulate events, conditions, and actions. In the latest UML version 1.5, an action semantics and sample well-known action languages such as the Action Specification Language[4] or the Bridge-Point Action Language[5] are described to tackle this issue. Note that the effects of these actions must have a corresponding foundation in the semantics description.

TR represents a set of transitions. Each transition connects two states and may take a triggering event, a guard, and an action. UML does not allow transitions to cross borders of an And-state with a source outside of that And-state. We here additionally do not allow a transition to start in a substate of an And-state x leading to a state outside of x, as this might cause an inconsistency with ongoing activities in an orthogonal state.

Function $substates$ determines all direct substates of a composite state. Function $defaultHistory$ defines the default state to enter for each history state. Functions $entry$ and $exit$ give the actions to take when a state is entered or left, respectively. Function $doActivity$ specifies the activity to take when a state is activated.

Semantics. As the purpose of our formalization is analysis of timing behavior, we may abstract from particular selection algorithms for dispatching events from the event queue and assume that the dispatching mechanism non-deterministically takes one event at a time from the event queue. Non-deterministic choice also applies for conflicting transitions, i.e., when transition selection cannot be uniquely resolved even if all preference rules were applied. Note that in other domains, certain deterministic policies might be required instead.

Concerning timing issues, the UML standard does not make assumptions w.r.t. the RTC-step semantics of Statecharts, e.g., it is possible for transitions to both be instantaneous or to take time [9, Section 3.75.1]. The same holds for states; they can be instantaneous as well as having a notable duration, e.g., when an activity is specified.

In a formal definition of time-based UML Statecharts, we have to provide a precise execution semantics w.r.t. evolving time. So we have to decide how time is evolving and how much time is needed in the affected operational parts of Statecharts. We identify the three operational parts (a) RTC-step, (b) actions, and (c) communication (i.e., calls). For each of these, we now briefly discuss their time-related behavior under the premises of the domain of manufacturing systems.

[4] http://www.kc.com

[5] http://www.projtech.com

Time-based RTC-step. In the first part of an RTC-step, an event is chosen from the event queue, i.e., some dispatching mechanism has to select an event. We restrict on clock-synchronous semantics, i.e., an object dispatches a new event from its event queue to be processed by the corresponding Statechart only at the tick of the (global) clock in the moment when the previous RTC-step is completed.

In the second part of each RTC-step, based on the dispatched event, the transition(s) to take have to be determined by evaluating guards and considering priority rules, and the actual transition execution has to take place, i.e., some actions might have to be executed when the transition(s) fire(s). Basically, the system will change from one stable source configuration denoting the current status of the object to a destination configuration denoting the subsequent stable state configuration right after the RTC-step.

When considering evolving time as an inherent characteristic of the system, two subsequent stable state configurations cannot occur at the same point of time. Thus, we assume at least a minimal expired time of one time unit between two stable state configurations.

We also have to decide at which point of time the state configuration actually changes. We assume that the source configuration becomes inactive when all exit actions and the transition action have been completed.

Timed Actions. Though UML assumes that actions do not take time (as opposed to "activities"), this is no longer valid in the considered domain, as we assume that at least a minimal time unit is elapsed when executing an action. Thus, a timing scheme for actions has to be defined, e.g., that an assignment to a variable takes one time unit, an asynchronous signal call takes one time unit, and a synchronous operation call takes until a corresponding response is received (i.e., it depends on a timing scheme for communication).

Communication. We assume a perfect underlying communication technology, i.e., none of the communicated events will be lost. Without that assumption, the task of formally analyzing the model becomes even more complex, as at any point of time, a synchronous call event may be lost and thus the Statechart may block.

From our point of view, synchronous operation call events sent to *remote objects* cannot be seen as actions *with negligible time* as described in the UML standard, as time evolves when waiting for an response on that operation call. Thus, we do not allow such operation call events as exit- or entry-actions of states. Nevertheless, we allow them as transition actions. But this raises the question, what states are activated during execution of that transition operation, as the source state(s) are left and the target state(s) are not yet entered when waiting for the operation to finish. This is a typical example for going back from formalization Step 4 to Step 3 in order to clarify new semantic issues.

Furthermore, we have to extend UML by timing annotations on operations, but note that operations invoked as exit- or entry-actions for the object itself must not have a specified execution time larger than one time unit.

4.3 MFERT

The third main activity of the formalization focuses on a UML Profile for modeling manufacturing systems and a corresponding formal semantics with a notion of time. In this context, we make use of the graphical MFERT notation that is dedicated to model manufacturing systems [13].

In MFERT, nodes represent either production processes or storages for production elements. *Production Process Nodes* (PPNs) represent logical locations where material is transformed and are drawn as annotated shaded rectangles. *Production Element Nodes* (PENs) are used to model logical storages of material and resources and are drawn as annotated shaded triangles. PENs and PPNs are composed to a bipartite graph connected by *directed edges* which define the flow of production elements. MFERT graphs establish both a static and dynamic view of a manufacturing system. On the one hand, the nodes are statically representing the participating production processes and element storages. On the other hand, directed edges represent the dynamic flow of production elements (i.e., material and resources) within the manufacturing system.

A corresponding UML Profile for MFERT and the actual semantic mapping M_{MFERT} to I/O-Interval Structures can be found in [5]. Basically, we integrate the timed Statechart variant (cf. Subsection 4.2) into extended object models (cf. Subsection 4.1), such that main structural system parts (i.e., MFERT nodes) are coupled with corresponding behavioral descriptions (i.e., Statecharts).

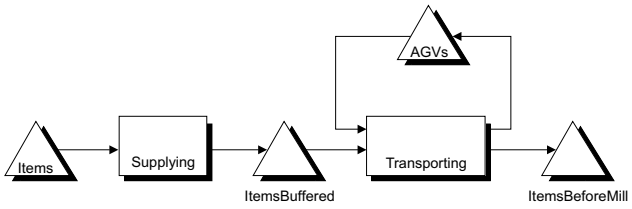

Figure 3. Sample MFERT Graph

Example. A sample MFERT graph is shown in Figure 3, where transportation of items by means of automated guided vehicles (AGVs) between processing steps is illustrated. This is a small outtake of a model that is composed of different manufacturing stations and transport vehicles that transport items between stations.

The sample UML Statechart in Figure 4 shows parts of the behavior specification of PPN Transporting – details of the negotiation part for accepting transportation orders are left out here for brevity reasons. The transport part basically consists of a chain of activities to perform – an instance of PPN Transportation is thus controlling the activities of an AGV object. The activities are initiated by operation calls on AGVs, such as `move()`, `load()`, and `unload()`. Recall that we allow to associate (estimated) execution times to these operations in class diagrams or MFERT graphs, respectively. E.g., operations `load()` and `unload()` might take between 5 and 10 time units.

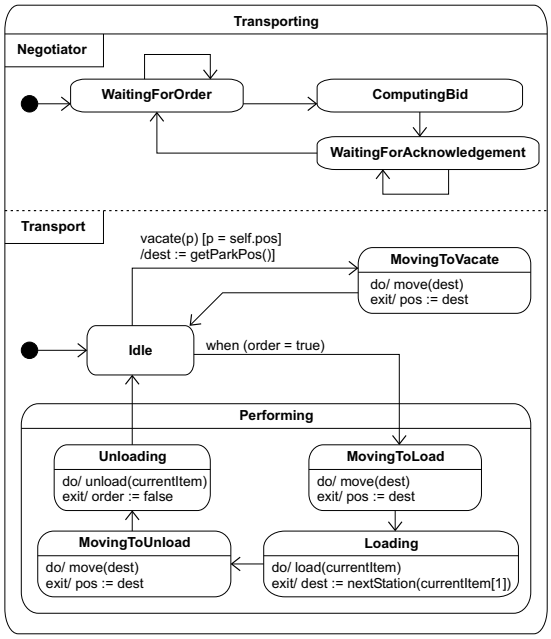

Figure 4. Parts of the Transporting Statechart

4.4 Property Specification with OCL

In addition to the actual model, we want to specify non-functional properties a model should fulfill. UML already provides (non-diagrammatic) means to specify constraints over a given model by OCL. But currently, OCL lacks means to specify constraints over the *dynamic behavior* of a UML model, i.e., consecutiveness of states and state transitions as well as time-bounded constraints. However, it is essential to specify such constraints to guarantee correct system behavior, e.g., for modeling real-time systems.

Temporal extensions of OCL usually introduce temporal logic operators (e.g., eventually, always, or never) that enable modelers to specify required occurrences of actions and events, e.g., [2]. Unfortunately, the resulting syntax of these extensions often do not combine well with current OCL concepts, as e.g., temporal expressions are very similar to rather cryptic temporal logic formulae.

In contrast to these approaches, we extend OCL with only minor modifications on the language metalevel. Our extension is defined by a UML Profile based on the metamodel proposed for OCL 2.0 in [8]. The formal semantics of our extension is given by a direct correspondence to time-annotated temporal logic formulae in CCTL [6].

In OCL, it is already possible to specify constraints concerning the current state of an object, using the predefined operation `oclInState()`, but a precise semantics of this operation is missing so far. We therefore first formalized Statechart state configurations and could then define a corresponding semantics for that operation [7]. Note that this is an essential prerequisite for completing the formal semantics of the OCL 2.0 proposal and for defining a mapping $M_{RT\text{-}OCL}$ from temporal state-related OCL expressions to CCTL.

For example, the following invariant requires that for each instance of `Transporting`, at each point of time, the states `Idle` and `Performing` must be subsequently entered within the next 100 time units:

```
context Transporting
inv: self@post(1,100)->forAll(p:OclPath |
         p->includes(Sequence{Idle,Performing}))
```

This notation is compliant with existing OCL syntax. Operation `@post()` is newly introduced and can be used for all user-defined types. It extracts the set of possible future execution paths, optionally restricted by a timing interval.

5 Conclusion

Experiences with our formalization approach regard (a) the UML abstract syntax and its informal semantics with clarification of open issues, (b) tailoring towards a particular domain by selecting and introducing model elements (in our case time-dependent models) and defining a mapping to formal languages, and (c) the actual application of our formalized UML parts for verification purposes.

Firstly, many syntactical UML elements can be directly simulated, i.e., they do not need to be formalized separately. Semantic variation points – once identified – can be fixed for a given domain. Additional domain-specific concepts can either be introduced by stereotypes in UML Profiles or as heavyweight extensions on the metalevel. However, from a formalization point of view in both cases a formal semantics must still be provided separately.

Secondly, concerning the formalization for domain-specific modeling, we considered modeling of manufacturing systems. We defined a restricted UML syntax by means of a UML Profile. Note that some major restrictions were necessary to be able to define a mapping to the semantic domain of I/O-Interval Structures, e.g., dynamic object creation and deletion and infinite value domains could not be considered. On the other hand, we needed to introduce new modeling concepts to capture timing aspects.

Finally, we applied our state-oriented temporal OCL extension for specification of real-time properties [6]. The provided mappings allow to transform MFERT models and OCL properties into corresponding input for the real-time model checker RAVEN [11] that automatically verifies whether given properties are satisfied for a particular model.

Nevertheless, additional techniques such as decomposition or abstraction are necessary to efficiently perform model checking on models of reasonable size. We currently investigate how MFERT models can be decomposed into submodels to be able to perform model checking on models of smaller size (compositional model checking). A tool that automatically translates models in MFERT notation into I/O-Interval Structures is currently being implemented.

Acknowledgements

The work described in this article receives funding by the DFG project GRASP within the DFG Priority Programme 1064 “Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen”.

References

[1] M. v. d. Beeck. A Structured Operational Semantics for UML-Statecharts. *Software and Systems Modeling (SoSyM), Springer*, 1(2):130–141, December 2002.

[2] M. Cengarle and A. Knapp. Towards OCL/RT. In *Formal Methods – Getting IT Right*, volume 2391 of *LNCS*, pages 389–408. Springer, July 2002.

[3] A. David, M. Möller, and W. Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In *FASE 2002, Grenoble, France*, volume 2306 of *LNCS*, pages 218–232. Springer, 2002.

[4] A. Evans, R. France, K. Lano, and B. Rumpe. Developing the UML as a Formal Modelling Notation. In *UML'98 – Beyond the Notation. Mulhouse, France, June 1998*, pages 297–307, 1998.

[5] S. Flake and W. Müller. A UML Profile for MFERT. Technical Report 04/2002, C-LAB, Paderborn, Germany, March 2002.

[6] S. Flake and W. Müller. A UML Profile for Real-Time Constraints with the OCL. In *UML 2002. Model Engineering, Languages, Concepts, and Tools. Dresden, Germany*, volume 2460 of *LNCS*, pages 179–195. Springer, 2002.

[7] S. Flake and W. Müller. Semantics of State-Oriented Expressions in the Object Constraint Language. In *SEKE 2003, San Francisco, CA, USA*, pages 142–149. Knowledge Systems Institute, July 2003.

[8] A. Ivner, J. Högström, S. Johnston, D. Knox, and P. Rivett. Response to the UML2.0 OCL RfP, Version 1.6. OMG Document ad/03-01-07, January 2003.

[9] Object Management Group. Unified Modeling Language 1.5 Specification. OMG Document formal/03-03-01, March 2003.

[10] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Bremen, Germany, 2001.

[11] J. Ruf. RAVEN: Real-Time Analyzing and Verification Environment. *Journal on Universal Computer Science (J.UCS), Springer*, 7(1):89–104, February 2001.

[12] J. Ruf and T. Kropf. Modeling and Checking Networks of Communicating Real-Time Systems. In *Correct Hardware Design and Verification Methods (CHARME 99)*, pages 265–279. Springer, September 1999.

[13] U. Schneider. *Ein formales Modell und eine Klassifikation für die Fertigungssteuerung*. PhD thesis, Heinz Nixdorf Institut, HNI-Verlagsschriftenreihe, Band 16, Paderborn, Germany, 1996. (in German).

[14] M. Schrefl and M. Stumptner. Behavior Consistent Specialization of Object Life Cycles. *ACM Transactions of Software Engineering and Methodology (ACM TOSEM), ACM Press*, 11(1):92–148, January 2002.

[15] B. Selic and J. Rumbaugh. Using UML for Modeling Complex Real-Time Systems. White Paper, 1998. http://www.rational.com/media/whitepapers/umlrt.pdf.