Specification and Formal Verification of Temporal Properties of Production Automation Systems

Stephan Flake[1], Wolfgang Müller[1], Ulrich Pape[2], and Jürgen Ruf[3]

[1] Universität Paderborn
Fürstenallee 11, 33102 Paderborn, Germany

[2] Heinz Nixdorf Institut
Fürstenallee 11, 33102 Paderborn, Germany

[3] IBM Deutschland Entwicklung GmbH
Schönaicher Str. 220, 71032 Böblingen, Germany

Abstract

This article describes our approach for the specification and verification of production automation systems with real-time properties. We focus on the graphical MFERT notation and RT-OCL (Real-Time Object Constraint Language) for the specification of state-oriented real-time properties. RT-OCL is an extension of the Object Constraint Language (OCL) that is part of the Unified Modeling Language (UML). We introduce the formal semantics of RT-OCL based on a formal model of UML Class and State Diagrams and provide a mapping to temporal logics. The applicability of our approach is demonstrated by the case study of a manufacturing system with automated guided vehicles.

1 Introduction

In early stages of development of production automation systems, system model behavior is most frequently analyzed by quantitative and qualitative simulation. However, due to the complexity of those systems, simulation can never provide coverage for complete verification for those systems. In recent years, formal verification with equivalence and model checking has received a wide acceptance in the domain of digital circuit and communication protocol design. A model checker verifies a property specification for a given state-oriented model of a system, typically given as a Kripke Structure. The model checker returns either 'true' or generates a counter example in cases when the model does not satisfy the property. The counter example demonstrates an execution of the model that leads to a situation which falsifies the property. This can be most helpful for detailed error analysis. The most remarkable advantage of model checking is that the task of verifying is fully automated. However, model checking has two main obstacles in practical application. The first one is the *state explosion problem* in

dependence to the number of possible inputs. The second one is due to the specification of properties in *temporal logics*, since it often turns out that designers and programmers are not familiar with formal methods and regard it as a task too cumbersome to specify and understand properties in temporal logics.

For production automation systems, the correct *time-critical behavior* of required properties is of particular interest. This is already important in early phases of development to avoid expensive and time-consuming changes to the system under development at later stages. Though classical model checking is mainly for the verification of cycle-accurate behavior *without* timing properties, there are a few tools like the RAVEN model checker [36] that support the formal verification of time-annotated system models and additionally provide basic timing analysis.

In this article, we present the GRASP[1] approach to formal verification of production automation systems. The GRASP approach covers the design flow for the modeling and formal verification of production automation systems by means of the domain-specific modeling language MFERT[2] with complementary visualization through animation of a virtual 3D model (cf. Figure 1). MFERT is a methodology and graphical language for the description and analysis of production automation systems. For analysis, GRASP focuses on model checking and integrates a model checker by seamlessly embedding it into a graphical environment with 3D animation for virtual prototyping, in particular for the animation of counter examples. The main idea was that in a first step the designer specifies a model in a graphical specification language, namely MFERT. The model, i.e., the MFERT description, is then translated into an annotated state machine-based formalism (i.e., I/O-Interval Structures [38]) for model checking. Additionally, properties are specified and translated into temporal logics (i.e., Clocked Computation Tree Logic, CCTL [37]) for formal verification with the RAVEN model checker. Once the objects in the virtual prototype are associated with the system model, the execution of counter examples can be observed in the virtual prototype animation.

One of the main visions of the GRASP approach was to provide practical means to designers with programming skills to facilitate property specification. We investigated existing related work and developed a pattern-based approach in the early phases of the GRASP project (see Section 2.1 for more details). In a second step, we decided to integrate an extension of the Object Constraint Language (OCL) [42, 43] with MFERT for the specification of required properties for production automation systems. OCL was introduced as a language for the specification of constraints in the context of the Unified Modeling Language (UML) de-facto standard [29], focusing on Class Diagrams and on guards in be-

---

[1] GRASP (GRAphical Specification and Real-Time Verification for Production Automation Systems) is a project within the DFG Priority Programme 1064 'Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen'.

[2] MFERT is short for 'Modell der FERTigung' (German for 'Model of Manufacturing').

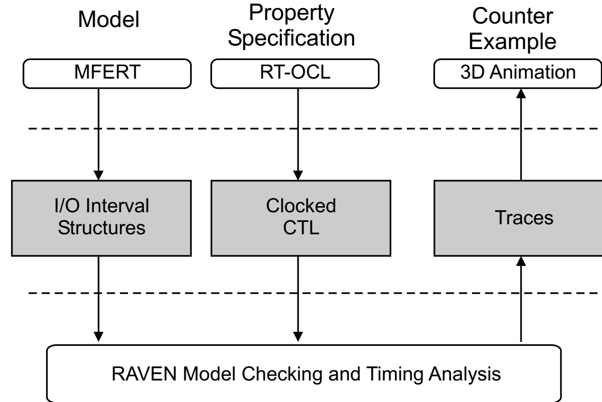

Fig. 1. GRASP Approach to Verification of Production Automation Systems

havioral diagrams. The syntax of OCL comes in a 'programmer-friendly' style using dot-notation and operation calls as known from object-oriented languages. With the wide acceptance of UML, OCL has also received a considerable visibility. However, OCL currently lacks sufficient means to specify constraints over the *temporal behavior*, i.e., the evolution of state activations and state transitions as well as timing constraints. Since it is essential to be able to specify such timing constraints for time-dependent systems to guarantee correct system behavior, we developed an OCL extension, i.e., RT-OCL, that overcomes this limitation and at the same time keeps compliant with the syntax and semantics of the latest version of OCL, i.e., Version 2.0 [28].

To seamlessly integrate RT-OCL with the domain-specific language MFERT, we defined UML Profiles for both MFERT and RT-OCL and defined mappings to the formal means of I/O-Interval Structures and CCTL, respectively. CCTL was introduced by Ruf and Kropf in [37] for the specification of properties over I/O Interval Structures. CCTL formulae are composed from propositions denoting predicates in combination with Boolean connectives and time-annotated temporal operators. The temporal CCTL operators build upon the common CTL operators and are annotated by timing intervals, such as `AF[a,b]`, where `A` is a path quantifier ('*on all paths*') and `F` is the temporal CTL operator ('*eventually in the future*') that is further limited by the timing interval `[a,b]`. For further details of CCTL and its application for real-time model checking we refer to [35, 39].

The remainder of this article is structured as follows. The next section discusses related work in the domain of (real-time) property specifications. Section 3 gives an introduction to MFERT with an example. Section 4 introduces syntax and semantics of our temporal OCL extension RT-OCL. Section 5 demonstrates the application of RT-OCL in the context of a production automation scenario with automated guided vehicles. Finally, Section 6 concludes the article and gives an outlook on future work.

2 Related Work

There are already several approaches that make use of graphical captures for model checking, e.g., with Petri Nets or StateCharts [6, 3]. Those means are used to define the system by a model. Behavioral property specification, however, is usually still performed by means of temporal logics formulae, mainly in Computation Tree Logic (CTL) or Linear Time Logic (LTL). The most prominent formal method that investigates whether a given model satisfies such property specifications is *model checking* [8]. Model checking takes a set of synchronous finite state machines as a model and (a set of) temporal logic formulae as the specification of required properties. In application, the main obstacles for model checking lie in the *state explosion problem* and in the *adequate specification* of properties by means of temporal logic formulae [33]. Several approaches to support property specification have been developed. In the following subsections, we distinguish the areas of (a) pattern-based specification, (b) graphical property languages, and (c) temporal extensions of OCL.

2.1 Pattern-Based Property Specification

To support temporal logic property specification, some approaches identify *patterns* which provide the user with structured application of formulae. First attempts in pattern classification led to taxonomies that coarsely distinguish between safety and liveness properties. A detailed pattern-based classification is published by Dwyer et al. in [14]. That pattern system is based on the investigation of more than 500 examples for property specification and presents a semantically ordered hierarchy of property patterns. For instance, absence, eventual existence, and global existence of states/events are combined as so-called occurrence patterns.

The idea of patterns was adopted not just to *classify* but also to *construct* specifications for finite state verification. For example, the Testbed Studio is a framework for business process modeling that provides a small set of templates in natural English language for verification with the SPIN model checker [26]. These templates are also denoted as patterns, although they refer to concrete specifications in contrast to the previously mentioned classification patterns. In a more general approach of natural language oriented specification, the PROSPER project aims at the specification through an English language subset [24].

In the early phases of the GRASP project, we developed an interactive visual framework that employs structured English sentences [23] as given in Figure 2. Compared to other pattern-based approaches, we provide a richer set of specifications, in particular, as we additionally cover explicit timing annotations. In contrast to temporal logic formulae-based approaches, non-experts can more easily capture the final sentence in structured English than just by CTL or LTL. Compared to unstructured English, the available structured English fragments give the uneducated user a better guidance through the allowable and non-allowable specifications with less iterations. However, it turned out that this

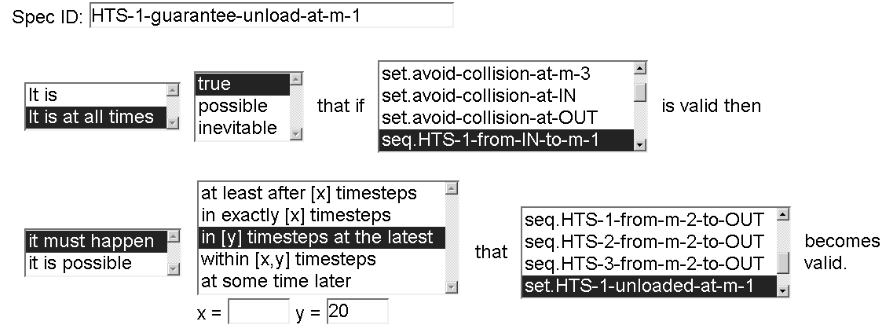

Fig. 2. Specification with Structured English Sentences

approach leads to quite long sentences and remains too cumbersome for more complex applications, so that we started to investigate alternative approaches.

2.2 Graphical Property Specification

Regarding property specification by visual means we can distinguish two different kinds of approaches. The first ones are still syntactically based on CTL or LTL specifications. Those frameworks provide support to visually compose segments of specifications, e.g., by enabling and disabling parts of specifications during the development process. In UPPAAL, invariants and reachability properties can be specified using a very limited subset of LTL formulae [27]. To create specifications with this approach, the user must know how to apply and to control temporal logic formulae. Other approaches have an abstract graphical notation, which is translated to temporal logic formulae before checking. Examples are *Symbolic Timing Diagrams (STDs)* [15] and *Life Sequence Charts (LSCs)* [10], which are for StateChart verifications.

2.3 OCL-Based Property Specification

As an alternative to the previous approaches, we investigated UML in combination with OCL for model checking. OCL was originally developed complementary to UML to restrict values of (parts of) a model, e.g., attributes or associations, but has recently been extended towards a more general query and expression language [28]. Several non-commercial OCL tools are currently available that implement syntax and type checks, dynamic constraint validation, test automation, and code generation of OCL constraints. An overview can be found in [31, 32][3]. OCL constraints are frequently used in the UML specification documents at the UML metamodel level (M2 layer) to define the static semantics

[3] See http://www.klasse.nl/ocl and http://www.um.es/giisw/ocltools for updated lists of available OCL tools.

of UML diagrams. Those so-called *well-formedness constraints* specify syntactical restrictions on diagrammatic model elements. There are several approaches that either extend OCL for temporal constraints specification or introduce alternative UML-based means to express behavioral real-time constraints for UML diagrams.

Ramakrishnan et al. [30] extend the OCL syntax by additional grammar rules with unary and binary future-oriented temporal operators (e.g., `always` and `never`) to specify safety and liveness properties. Ziemann and Gogolla [45] introduce similar temporal operators based on a finite linear temporal logic. Therein, Richter's formal object model [31] is extended to provide a formal definition of system state sequences. However, it is left open *how* system state sequences are exactly derived. A similar approach has been published by Conrad and Turowski in the area of business modeling [9]. Their approach additionally considers past-temporal operators; a formal semantics is not provided.

Distefano et al. [13] define *Object-Based Temporal Logic* (BOTL) to facilitate the specification of static and dynamic properties. BOTL is not directly an extension of OCL. It rather maps a subset of OCL into object-oriented CTL. Bradfield et al. [2] extend OCL by useful causality-based templates for dynamic constraints. A template consists of two clauses, i.e., the *cause* and the *consequence*. The cause clause starts with the keyword `after` followed by a Boolean expression, while the consequence is an OCL expression prefaced by `eventually`, `immediately`, `infinitely`, etc. The templates are formally defined by a mapping to *observational μ-calculus*, a two-level temporal logic with OCL on the lower level.

In the domain of real-time systems modeling, we can find mainly three approaches for temporal constraint specification. Roubtsova et al. [34] define a UML Profile with stereotyped classes for dense time as well as parameterized specification templates for deadlines, counters, and state sequences. Each of the templates has a structural-equivalent dense time temporal logics formula in Timed Computation Tree Logic (TCTL). Sendall and Strohmeier [41] introduce timing constraints on state transitions in the context of a restricted form of UML protocol state machines that define the temporal ordering between operations. Five time-based attributes on state transitions are proposed, e.g., (absolute) completion time, duration time, or frequency of state transitions. Cengarle and Knapp [5] present OCL/RT, a temporal extension of OCL with modal operators `always` and `sometime` over event occurrences. They specify deadlines and timeouts of operations and reactions on received signals. Events are equipped with time stamps by introducing a metaclass `Time` with attribute `now` to refer to the time unit at which an event occurs. In turn, each object can access the set of currently queued events at each point in time.

In contrast to the event-based temporal extensions of OCL, we focus on *state-oriented properties* due to the intended application domain of state-based modeling of production automation systems with MFERT. Note that it is already possible to refer to the states of UML State Diagrams in standard OCL, i.e., the operation `oclInState(stateName)` returns a Boolean value that indicates

whether a given state is currently activated or not. However, OCL does not yet integrate the notion of State Diagram states on the language definition level, i.e., the semantics of State Diagram states in the context of OCL expressions is not sufficiently defined so far. To overcome this deficiency, we provided a formal semantics for state-oriented OCL expressions for application with UML State Diagrams in [22].

3 MFERT

Our approach is based on MFERT as the basis for modeling of production automation systems. MFERT is a language and a methodology for the specification and implementation of planning and control assignments in production processes. MFERT is basically a universal approach which has been successfully applied in various projects with different industrial partners [12, 11], additionally acknowledged by the German science award of logistics [40]. An MFERT model is based on *production elements* and *production processes*. Production elements represent objects whose properties are changed by processes and transformations. Properties of production elements are described by attributes. A production element obtains its own identity, composed out of the description and the element's correlative status. Using this identity, the different states during the production can be associated to production elements. An MFERT model is a directed bipartite graph of *E-nodes* for elements and *P-nodes* for processes. The graphical notation of an E-node is a triangle. P-nodes are represented as rectangles. Each E-node represents a specific state and can be seen as a container for elements in the respective state. P-nodes represent transformations on elements, performed by the corresponding processes. Nodes are connected by edges that describe exchange relations between two nodes. Edge annotations define if a predecessor is bringing, providing, or waiting for elements and processes, as well as that a successor is fetching, receiving, or waiting for elements. Interface edges are for connecting different levels of hierarchy. They additionally allow the coupling to a real production environment. MFERT-Elements and MFERT-Processes are in certain states, which are characterized by attributes. An attribute denotes a property of an element and assigns a value to a relation. Constructors for the definition of discrete time modes are available. In practice, different time models are required for the definition of production assignments, e.g., the provision of the source materials for an assembly line may take place non-recurringly at the beginning of a shift, while the mounted end products are transported every hour to the delivery store. For implementation, model nodes are equipped with functions and their process control is carried out by means of message exchange between nodes and by a so-called *global manager* that coordinates the computations in the model.

Figure 3 gives the MFERT example of a subsystem for the production of engines with processing steps `Milling`, `Drilling`, and `Washing`. The primary input of the example is modeled by the E-nodes `RawEngines` and `RawShafts`. Corresponding processes are used to supply these items into E-nodes `EnginesSupplied`

and ShaftsSupplied. Input and output buffers of processes are modeled by the corresponding E-nodes like ItemsBeforeMill, ItemsAfterMill etc. The transport between stations and the primary output is modeled by different transport processes like TransportingToMill and TransportingToOutput. Automated guided vehicles (AGVs) transport items between the different production steps, where the AGV resource management is modeled as a separate E-node.

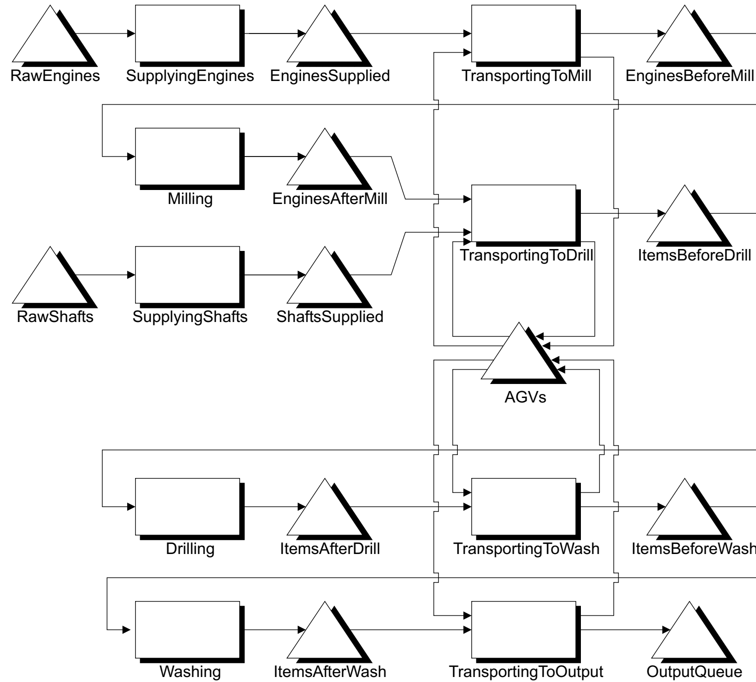

Fig. 3. MFERT Graph of the Case Study

The input/output behavior of P-nodes is basically defined as time-annotated finite state machines whose graphical presentation is not given in MFERT. For P-node representation, we thus developed a variant of timed UML State Diagrams in [17] to define the local functionality of P-nodes.

In order to be able to give a formal semantics, we have to limit the set of actions and activities of standard UML State Diagrams. We only consider actions and activities that perform (a) requests of P-nodes to put and get elements to and from E-nodes, (b) transfers of production elements between MFERT nodes, and (c) local transformations with a duration. Due to the limited space, we give just a a small example and refer to [17] for more details about the graphical notation, the formal model, and the dynamic semantics of MFERT.

Consider the P-node TransportingToMill and its corresponding State Diagram given in Figure 4. The diagram specifies that a transport requires an AGV

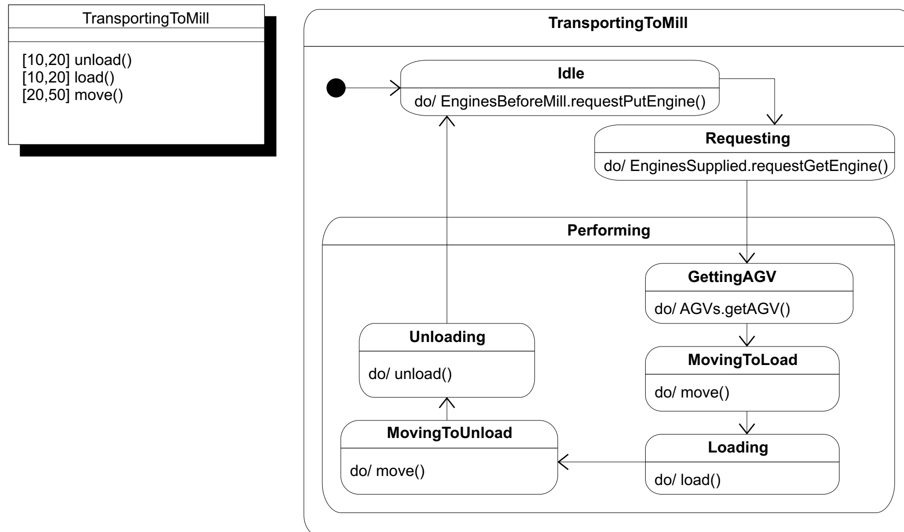

Fig. 4. P-node `TransportingToMill` and its State Diagram

and either an `Engine` or a `Shaft`, where the movement between stations is expected to be executed within 20 to 50 time units. Such basic timing declarations are based upon information gained from the actual physical alignment of the production system.

4 RT-OCL

Figure 5 gives an overview of our formalization approach in the domain of modeling production automation systems. We decomposed the whole approach into four different activity parts. Figure 5 also illustrates the dependencies among the different activities.

First, we integrated the notational concepts of UML State Diagrams into the existing formal description of Class Diagrams by Richters [31]. Basically, the resulting so-called *extended object model* is based upon a set-theoretic definition of the UML metamodel parts for Class Diagrams and State Diagrams. In parallel, we developed a timed variant of UML State Diagrams. Note that this activity is at a lower position in Figure 5 to indicate it as a more domain-specific task, because timing issues are a non-standard concept of UML State Diagrams, while the extended object model basically concerns standard UML. Integrating the two formal models and applying further restrictions leads to our domain-specific notation MFERT, which is provided as a domain-specific UML Profile with stereotypes, e.g., for P-nodes and E-nodes [22]. Additionally, we defined a mapping to the semantic domain of *I/O-Interval Structures* [17].

We have also defined an extension of OCL called RT-OCL that allows for specification of temporal state-oriented properties. For such OCL expressions,

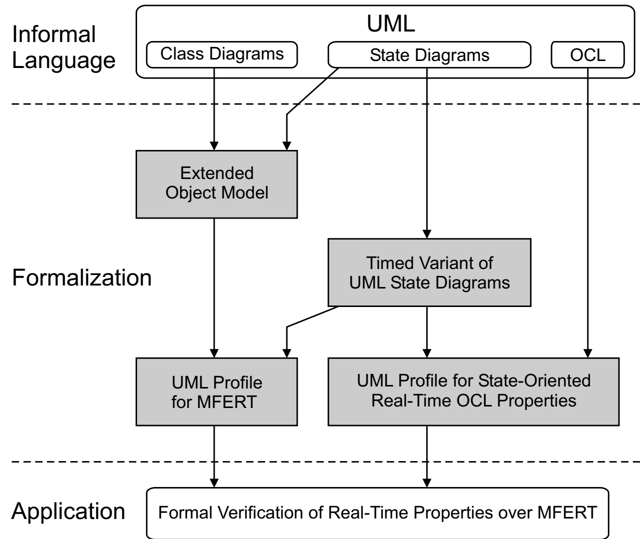

Fig. 5. Overview of the Formalization Approach [16]

we provide a mapping to CCTL formulae. The semantics of the combination of MFERT notation and temporal state-oriented 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 that context, recall that the model checker RAVEN is able to verify whether a model (a set of I/O-Interval Structures) satisfies a given property (a CCTL formula) [36].

The reminder of this section sketches some details of the definition of RT-OCL and its semantics. But first, we give an introduction to standard OCL in the next subsection.

4.1 OCL

The Object Constraint Language is an integral part of UML [29, Chapter 6]. OCL constraints are defined over a given UML model to restrict the values of object properties. OCL is mainly applied to define invariants for classes and pre- and postcondition of operations. As OCL is a declarative expression-based language, evaluation of OCL expressions does not have side effects on the corresponding UML model.

Each OCL expression has a type. Beyond user-defined model types (e.g., classes or interfaces) and predefined basic types (e.g., `String`, `Integer`, `Real`, or `Boolean`), OCL has a notion of object collection types, i.e., sets, ordered sets, sequences, and bags. Collection types are homogeneous in the sense that all elements of a collection have a common type. A standard library is available with operations to select, access, and manipulate values and objects.

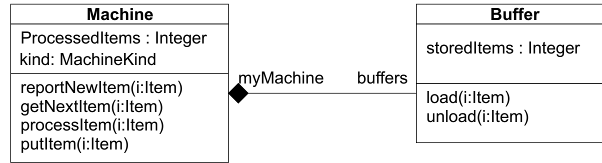

Fig. 6. Sample Class Diagram

Assume, for example, that we have a model with classes `Machine` and `Buffer` and an association between these classes (cf. Figure 6). The following invariant defines that each instance of class `Machine` has at least one associated buffer:

```
context Machine
inv: self.buffers->notEmpty()
```

We briefly outline how to read this OCL constraint. The identifier following the `context` keyword specifies the class for which the constraint is defined. The keyword `inv` specifies that this is an invariant, i.e., for each object of the context class the following expression must evaluate to true 'at any time'[4]. The (optional) keyword `self` refers to the object for which the constraint is evaluated. Attributes, operations, and associations can be accessed by dot notation, e.g., evaluation of `self.buffers` results in a (possibly empty) set of instances of `Buffer`. The arrow operator indicates that a collection of objects is manipulated by one of the predefined OCL collection operations. For example, operation `notEmpty()` returns 'true' if the accessed set is not empty.

Standard OCL currently lacks means to specify constraints over the dynamic behavior of a UML model. Constraints covering the consecutiveness of states and state transitions as well as time-bounded constraints cannot be defined. However, since those are essential to specify a correct system behavior, we have developed a temporal OCL extension that enables modelers to specify state-oriented real-time constraints [19, 20]. Because the official UML 1.5 specification [29] did not come with an OCL metamodel, our first idea was to develop an extension of the OCL type metamodel that was presented by Baar and Hähnle in [1]. More recently, in reply to the OMG's OCL 2.0 Request for Proposals, the extensive OCL 2.0 language proposal by Ivner et al. [25] became available, which addresses a seamless integration of OCL to relevant parts of UML. In October 2003, this proposal has been adopted by the OMG as the official OCL 2.0 Specification [28]. Based on the metamodel provided in these documents, we developed a more lightweight approach by defining a UML Profile for our temporal OCL extension [22]. The syntax and semantics of this extension are briefly described in the following two subsections.

[4] Note that an invariant may be violated during execution of an operation. The term 'at any time' has therefore to be refined by a dynamic OCL semantics. See our proposal in [18].

4.2 OCL Syntax Extension

The concrete syntax of OCL 2.0 is defined by an attributed grammar in EBNF (Extended Backus-Naur Form) with inherited and synthesized attributes as well as disambiguating rules. *Inherited attributes* are defined for elements on the right hand side of production rules. Their values are derived from attributes defined for the left hand side of the corresponding production rule. For instance, each production rule has an inherited attribute `env` (environment) that represents the rule's namespace. *Synthesized attributes* are used to keep results from evaluating the right hand sides of production rules. For instance, each production rule has a synthesized attribute `ast` (abstract syntax tree) that constitutes the formal mapping from the concrete to the abstract syntax. *Disambiguating rules* allow to uniquely determine a production rule in the case of syntactically ambiguous production rules.

The following rule gives the main production rule for temporal expressions we introduced for our RT-OCL extension. The idea is to interpret a future-oriented temporal expression as a kind of operation call. Future temporal OCL expressions map to a new stereotype `FutureTemporalExp` that is a specialization of `OperationCallExp` on the abstract syntax level (i.e., the OCL metamodel). For temporal OCL expressions, we introduce a temporal operator '@' to distinguish temporal expressions from OCL's common dot and arrow notation for accessing attributes, operations, and associations.

```
FutureTemporalExpCS ::= OclExpressionCS '@'
                                        simpleNameCS '(' argumentsCS? ')'
Abstract Syntax Mapping:
   FutureTemporalExpCS.ast : FutureTemporalExp
Synthesized Attributes:
   FutureTemporalExpCS.ast.source            = OclExpressionCS.ast
   FutureTemporalExpCS.ast.arguments         = argumentsCS.ast
   FutureTemporalExpCS.ast.referredOperation =
      OclExpressionCS.ast.type.lookupOperation(
                                   simpleNameCS.ast,
                                   if argumentsCS->notEmpty()
                                     then argumentsCS.ast->collect(type)
                                     else Sequence{}
                                   endif )
Inherited Attributes:
   OclExpressionCS.env = FutureTemporalExpCS.env
   argumentsCS.env     = FutureTemporalExpCS.env
Disambiguating Rules:
   -- Operation name must be a (future-oriented) temporal operator.
  [1] Set{'post'}->includes(simpleNameCS.ast)
   -- The operation signature must be valid.
  [2] not FutureTemporalExpCS.ast.referredOperation.oclIsUndefined()
```

Note that an operation call in the abstract syntax has a source, a referred operation, and operation arguments. In this case, the variable `ast` is re-typed

to `FutureTemporalExp` and thus inherits the features `source`, `arguments`, and `referredOperation` from the metatype `OperationCallExp`. These features get the evaluation results of the corresponding parts of the right-hand side of the production rule (cf. the section '`Synthesized Attributes`' above).

Additional temporal operations can easily be introduced at a later point of time, as just the disambiguating rule `[1]` has to be modified in such cases. For instance, `next()` can be introduced as a shortcut for `post(1,1)`, or `post()` as shortcut for `post(1,'inf')`.

4.3 Semantics

While the syntactic integration of the temporal OCL extension is straightforward, the definition of the semantics needs more investigation. The OCL 2.0 specification provides extensive semantic descriptions by both a metamodel-based as well as a formal mathematical approach, but unfortunately, those are currently neither consistent nor complete [18].

In the metamodel-based approach, the semantics of an OCL expression is given by associations between the different modeling layers M1 (user model layer) and M2 (metamodel layer). On the one hand, each value defined in the semantic domain on layer M1 is associated with a type defined in the abstract syntax on layer M2. On the other hand, each evaluation is associated with an expression on the abstract syntax. Given a snapshot of a running system, the associations yield to a unique value for an OCL expression, which determines the result value of expression evaluation.

The second approach gives the *formal semantics* of OCL and is based on set-theory using the notion of an *object model* [28]. An object model is a tuple

$$\mathcal{M} = \langle\ CLASS, ATT, OP, ASSOC, \prec, associates, roles, multiplicities\ \rangle$$

with a set $CLASS$ of classes, a set ATT of attributes, a set OP of operations, a set $ASSOC$ of associations, a generalization hierarchy $\prec$ over classes, and functions *associates*, *roles*, and *multiplicities* that give for each $as \in ASSOC$ its dedicated classes, the classes' role names, and multiplicities, respectively.

The formal semantics of that object model, however, lacks descriptions of ordered sets, global OCL variable definitions, OCL messages, and states of UML State Diagrams. Especially the latter are needed for our RT-OCL semantics. In the remainder, we call an instance of an object model a *system*. A system changes over time, i.e., the (number of) objects, their attribute values, and other characteristics change during system execution. This information is stored in *system states*, i.e., a system state represents a snapshot of the running system that is used to evaluate OCL expressions.

In OCL 2.0, a system state $\sigma(\mathcal{M})$ is formally defined as a triple $\sigma(\mathcal{M}) = \langle \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{ASSOC} \rangle$ with a set Σ_{CLASS} of currently existing objects, a set Σ_{ATT} of attribute values of the objects, and a set Σ_{ASSOC} of currently established links that connect the objects. However, this information is not sufficient to evaluate OCL expressions, as system states do not comprise currently

activated states and messages that have been sent. We therefore have to extend the formal model and system states accordingly, such that the resulting *extended object model* $\mathcal{M}$ with

$$\begin{aligned}\mathcal{M} = \langle\ & CLASS, ATT, OP, paramKind, isQuery, SIG, \\ & SC, ASSOC, \prec, \prec_{sig}, associates, roles, multiplicities\ \rangle\end{aligned}$$

additionally includes

- functions that give a parameter kind $\in \{in, inout, out\}$ for each operation parameter,
- functions that indicate whether an operation is a query operation without side-effects or not,
- signal receptions for classes with corresponding well-formedness rules, and
- State Diagrams and their association with classes[5].

The formalization of the extended object model is completed by a formal definition of state configurations[6] and an extension of the formal descriptor of a class.

Furthermore, the following information has to be added to system states to be able to evaluate OCL expressions that make use of state-related and OCL message-related operations:

- for each object, the input queue of received signals and operation calls that are waiting to be dispatched[7],
- the state configurations of all currently existing active objects,
- the currently executed operations, and
- for each currently executed operation, the messages sent so far.

The resulting tuple of a *system state* over an extended object model $\mathcal{M}$ is

$$\begin{aligned}\sigma(\mathcal{M}) = \langle\ & \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{ASSOC}, \Sigma_{CONF}, \Sigma_{currentOp}, \Sigma_{currentOpParam}, \\ & \Sigma_{sentMsg}, \Sigma_{sentMsgParam}, \Sigma_{inputQueue}, \Sigma_{inputQueueParam}\ \rangle\ .\end{aligned}$$

With those extensions, it is possible to define *execution traces* that capture all of those system changes that are relevant to evaluate OCL constraints (see [18] for details). The final formal semantics for our temporal OCL expressions is given in [22, 17]. While those articles also provide a general mapping to CCTL formulae, we here give some typical specification examples in the next section.

---

[5] Note that no specific execution semantics for State Diagrams has to be assumed here.

[6] UML only informally defines *active state configurations*. This results in some shortcomings, e.g., it is not considered that final states can be part of state configurations.

[7] As we only need to consider those events that are *relevant* for the evaluation of OCL constraints, implicit events such as completion events generated by State Diagram executions do not have to be considered in this context.

5 Application

We applied the GRASP approach to the case study of a Holonic Manufacturing System (HMS). The HMS case study was introduced by the IMS Initiative in [44]. The HMS is composed of a set of different manufacturing stations and a transport system as it is illustrated by the virtual 3D model in Figure 7. The different manufacturing stations transform items, e.g., by milling, drilling, or washing. Additional input and output storages are for primary system input and output. The flexible transport system consists of a set of automated guided vehicles (AGVs), i.e., autonomous vehicles that carry items between stations. We considered that stations have an input buffer for incoming items and that each AGV can take only one item at a time.

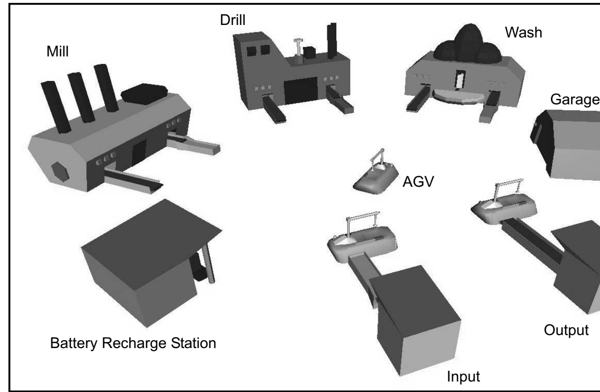

Fig. 7. Virtual 3D Model of the Holonic Manufacturing System

In the following, we provide some typical time-bounded constraints that refer to the MFERT model given in Section 3. We here just refer to the P-node `TransportingToMill`; similar constraints can be defined for other P-nodes. To outline the mapping, the following also gives the corresponding temporal logic formula in CCTL [35] for each RT-OCL constraint.

1. When `TransportingToMill` is in state `Idle`, we require that it gets a grant to put an engine into the subsequent E-node `EnginesBeforeMill` within the next 100 time units.

```
context TransportingToMill
inv: self.oclInState(TransportingToMill::Idle)
     implies
     self@post(1,100)->forAll(p:Sequence(OclState) |
                         p->includes(TransportingToMill::Requesting))
```

```
// CCTL formula:
AG ( (TransportingToMill.state==TransportingToMill.idle)
    -> AF[1,100](TransportingToMill.state==
                                      TransportingToMill.requesting))
```

2. A transport – once started after the acknowledgments have been received – has to be completed within 300 time units.

```
context TransportingToMill
inv: self.oclInState(TransportingToMill::Performing)

     implies
     self@post(1,300)->forAll(p:Sequence(OclState) |
                p->exists(s:OclState | s = TransportingToMill::Idle))

// CCTL formula:
AG ( (TransportingToMill.state==TransportingToMill.performing)
    -> AF[1,300](TransportingToMill.state ==
                                      TransportingToMill.idle))
```

3. The acknowledgment for an available AGV within composite state `TransportingToMill::Performing` must be received within 150 time units.

```
context TransportingToMill
inv: self.oclInState(TransportingToMill::Performing::GettingAGV)
     implies
     self@post(1,150)->forAll(p:Sequence(OclState) |
         p->exists(s:OclState |
                   s <> TransportingToMill::Performing::GettingAGV))

// CCTL formula:
AG ( ((TransportingToMill_performing.state==
                           TransportingToMill_performing.gettingAGV)
       & (TransportingToMill_performing.activated))
    -> AF[1,150]( (TransportingToMill_performing.state==
                         TransportingToMill_performing.movingToLoad)
                   & TransportingToMill_performing.activated
                )
   )
```

   Note here, that for technical matters, the activation of composite substate `Performing` has to be considered explicitly in the CCTL formula, as explained in [17, Section 7.3].

4. To enforce the production flow, it has to be guaranteed that the mill station is continuously served, i.e., at each point of time, state `Performing` will eventually be entered, and at each point of time, state `Idle` will eventually be entered. (The latter condition guarantees that state `Performing` is eventually left again.)

```
context TransportingToMill
inv: self@post()->forAll(p:Sequence(OclState) |
                      p->includes(TransportingToMill::Performing))
     and
     self@post()->forAll(p:Sequence(OclState) |
                         p->includes(TransportingToMill::Idle))

// CCTL formula:
AG AF (TransportingToMill.state==TransportingToMill.performing)
&
AG AF (TransportingToMill.state==TransportingToMill.idle)
```

Further examples of time-bounded state-oriented OCL constraints in the context of other UML and MFERT models can be found in [20, 4, 17]. In [21], we additionally demonstrated that it is possible to express the property specification patterns of Dwyer et al. [14].

For formal verification of MFERT models and RT-OCL/CCTL specifications, we apply the RAVEN model checker. In RAVEN, additional timing analysis queries help users to extract important time bounds from formal system descriptions. For instance, one might be interested in the maximal number of time steps an item is waiting until it is processed. Other typical problems are minimal and maximal delay times between events, e.g., the maximal time until the first item leaves the process. For intuitive interpretation of counter examples, RAVEN generates execution runs to give the example of a violating path in the state transitions. Additionally, we have extended RAVEN to automatically generate traces which trigger the animation of the virtual 3D model (cf. Fig. 7).

6 Conclusion

We have presented the GRASP approach for the specification and verification of production automation systems combining the domain-specific language MFERT and a temporal OCL extension, i.e., RT-OCL. For application in the context of model checking with the real-time model checker RAVEN, we have defined the semantics of RT-OCL by means of a mapping to Clocked Computational Tree Logic. The approach demonstrates that an OCL extension by means of a UML Profile towards temporal real-time constraints can be seamlessly applied on the M2 layer of UML, i.e., the OCL metamodel. Nevertheless, some extensions have to be made also on the user model level (i.e., M1 layer) in order to enable modelers to use our temporal OCL extensions. The presented extensions are based on a future-oriented temporal logic. However, current work additionally investigates the extension to past-oriented and additional logics.

We have implemented an editor and simulator for MFERT as given in Figure 8. Efficient code generation for RAVEN is currently investigated and under implementation. The code is generated with respect to efficient runtime in BDD composition and model checking considering optimized module and variable orders. The temporal OCL extensions as presented here are integrated into our

OCL parser and type checker, which translates constraints with temporal operations to CCTL formulae.

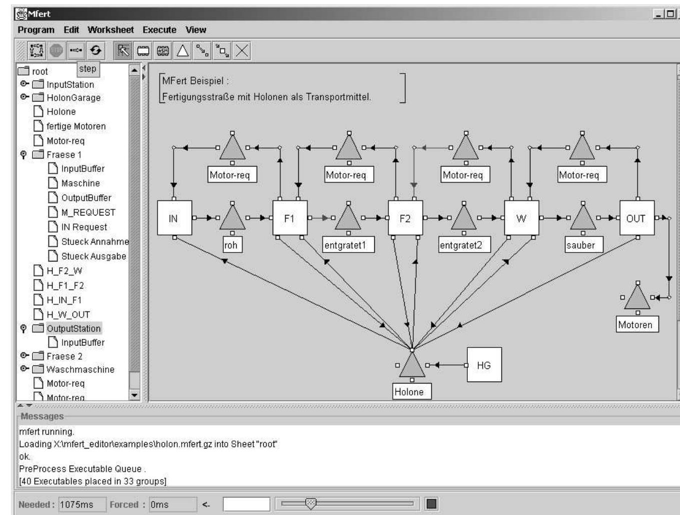

Fig. 8. MFERT Editor and Simulator

Acknowledgements

We thank Henning Zabel for his work on the MFERT editor and simulator.

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'. In the final phase, the GRASP project received partial contributions from the DFG Special Research Initiative 614 'Selbstoptimierende Systeme des Maschinenbaus'.

References

1. T. Baar and R. Hähnle. An Integrated Metamodel for OCL Types. In R. France, B. Rumpe, J.-M. Bruel, A. Moreira, J. Whittle, and I. Ober, editors, *OOPSLA'2000 Workshop Refactoring the UML: In Search of the Core, Minneapolis, MN, USA*, 2000.
2. J. C. Bradfield, J. Küster Filipe, and P. Stevens. Enriching OCL Using Observational Mu-Calculus. In R.-D. Kutsche and H. Weber, editors, *5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002). Part of the Joint European Conferences on Theory and Practice of Software (ETAPS 2002), Grenoble, France, April 2002*, volume 2306 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2002.

3. U. Brockmeyer and G. Wittich. Tamagotchis Need Not Die — Verification of STATEMATE Designs. In B. Steffen, editor, *Proceedings of TACAS '98, Lisbon, Portugal, March/April 1998*, volume 1384 of *Lecture Notes in Computer Science*, pages 217–231. Springer, 1998.
4. S. Burmester, S. Flake, H. Giese, W. Schäfer, and M. Tichy. Towards the Compositional Verification of Real-Time UML Designs. In P. Inverardi and J. Paakki, editors, *Joint 9th European Software Engineering Conference (ESEC) and 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11), Helsinki, Finland, September 2003*, pages 38–47. ACM Press, 2003.
5. M. V. Cengarle and A. Knapp. Towards OCL/RT. In L.-H. Eriksson and P. Lindsay, editors, *11th International Symposium of Formal Methods Europe (FME 2002), Formal Methods: Getting IT Right, Copenhagen, Denmark, July 2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 389–408. Springer, 2002.
6. A. Cheng. Petri Nets, Traces, and Local Model Checking. In *Algebraic Methodology and Software Technology*, pages 322–337, 1995.
7. T. Clark and J. Warmer, editors. *Object Modeling with the OCL. The Rationale behind the Object Constraint Language*, volume 2263 of *Lecture Notes in Computer Science*. Springer, 2002.
8. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
9. S. Conrad and K. Turowski. Temporal OCL: Meeting Specifications Demands for Business Components. In K. Siau and T. Halpin, editors, *Unified Modeling Language: Systems Analysis, Design, and Development Issues*, pages 151–165. IDEA Group Publishing, 2001.
10. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
11. W. Dangelmaier and H.-J. Warnecke. *Fertigungslenkung: Planung und Steuerung des Ablaufs der diskreten Fertigung*. Springer, 1997.
12. W. Dangelmaier and H. Wiedenmann. *Modell der Fertigungssteuerung*. Beuth Verlag GmbH, Berlin, Wien, Zürich, 1st edition, 1993.
13. D. Distefano, J.-P. Katoen, and A. Rensink. On a Temporal Logic for Object-Based Systems. In S. Smith and C. Talcott, editors, *Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000), Stanford, CA, USA, September 2000*, pages 305–326. Kluwer Academic Publishers, 2000.
14. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *21st International Conference on Software Engineering (ICSE 99), Los Angeles, CA, USA, May 1999*, pages 411–420. ACM Press, 1999.
15. K. Feyerabend and B. Josko. A Visual Formalism for Real Time Requirement Specifications. In *4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software (ARTS'97)*, Lecture Notes in Computer Science, pages 156–168. Springer, 1997.
16. S. Flake. Modeling and Verification of Manufacturing Systems: A Domain-Specific Formalization of UML. In M. Hamza, editor, *7th IASTED International Conference on Software Engineering and Applications (SEA 2003), Los Angeles, CA, USA, November 2003*, pages 580–586. ACTA Press, Calgary, Canada, 2003.
17. S. Flake. *UML-Based Specification of State-oriented Real-time Properties*. PhD thesis, Faculty of Computer Science, Electrical Engineering and Mathematics, Paderborn University, Shaker Verlag, Aachen, Germany, December 2003.

18. S. Flake. Towards the Completion of the Formal Semantics of OCL 2.0. In V. Estivill-Castro, editor, *27th Australasian Computer Science Conference (ACSC 2004), Dunedin, New Zealand, January 2004*, volume 26 of *Australian Computer Science Communications*, pages 73–82. Australian Computer Science Society, Sydney, Australia, 2004.
19. S. Flake and W. Müller. An OCL Extension for Real-Time Constraints. In Clark and Warmer [7], pages 150–171.
20. S. Flake and W. Müller. Specification of Real-Time Properties for UML Models. In R. Sprague, Jr., editor, *35th Hawaii International Conference on System Sciences (HICSS-35), Big Island, HI, USA, January 2002*. IEEE Computer Society Press, 2002.
21. S. Flake and W. Müller. Expressing Property Specification Patterns with OCL. In *The 2003 International Conference on Software Engineering Research and Practice (SERP'03), Las Vegas, NV, USA, June 2003*, pages 595–601. CSREA Press, Las Vegas, NV, USA, 2003.
22. S. Flake and W. Müller. Formal semantics of static and temporal state-oriented OCL constraints. *Software and Systems Modeling (SoSyM), Springer*, 2(3):164–186, October 2003.
23. S. Flake, W. Müller, and J. Ruf. Structured English for Model Checking Specification. In K. Waldschmidt and C. Grimm, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, Frankfurt/M., Germany, February 2000*, pages 251–262. VDE Verlag, Berlin, Germany, 2000.
24. A. Holt and E. Klein. A Semantically-Derived Subset of English for Hardware Verification. In *37th Annual Meeting of the Association for Computational Linguistics (ACL'99), University of Maryland, College Park, MD, USA*, pages 451–456, June 1999.
25. A. Ivner, J. Högström, S. Johnston, D. Knox, and P. Rivett. Response to the UML2.0 OCL RfP, Version 1.6 (Submitters: Boldsoft, Rational, IONA, Adaptive Ltd., et al.). OMG Document ad/03-01-07, January 2003. ftp://ftp.omg.org/pub/-docs/ad/03-01-07.pdf.
26. W. Janssen, R. Mateescu, S. Mauw, P. Fennema, and P. van der Stappen. Model Checking for Managers. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 1999, and Toulouse, France, September 1999*, volume 1680 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 1999.
27. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1+2), 1997.
28. OMG, Object Management Group. UML 2.0 OCL Final Adopted Specification. OMG Document ptc/03-10-14, October 2003. ftp://ftp.omg.org/pub/docs/ptc/-03-10-14.pdf.
29. OMG, Object Management Group. Unified Modeling Language 1.5 Specification. OMG Document formal/03-03-01, March 2003. ftp://ftp.omg.org/pub/-docs/formal/03-03-01.pdf.
30. S. Ramakrishnan and J. McGregor. Extending OCL to Support Temporal Operators. In *21st International Conference on Software Engineering (ICSE 99), Workshop on Testing Distributed Component-Based Systems, Los Angeles, CA, USA*, May 1999.

31. M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Bremen, Germany, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.
32. M. Richters and M. Gogolla. OCL: Syntax, Semantics, and Tools. In Clark and Warmer [7], pages 42–68.
33. D. Rosenblum. Formal Methods and Testing: Why State-Of-The-Art is not State-Of-The-Practise. In *ISSTA'96/FMSP'96 Panel Summary*, ACM SIGSOFT Software Engineering Notes, 21(4), July 1996.
34. E. E. Roubtsova, J. van Katwijk, W. Toetenel, and R. C. de Rooij. Real-Time Systems: Specification of Properties in UML. In *7th Annual Conference of the Advanced School for Computing and Imaging (ASCI 2001), Het Heijderbos, Heijen, The Netherlands, May/June 2001*, pages 188–195, 2001.
35. J. Ruf. *Techniken zur Modellierung und Verifikation von Echtzeitsystemen*. PhD thesis, Universität Karlsruhe, Karlsruhe, Germany, March 2000. (in German).
36. J. Ruf. RAVEN: Real-Time Analyzing and Verification Environment. *Journal on Universal Computer Science (J.UCS), Springer*, 7(1):89–104, February 2001.
37. J. Ruf and T. Kropf. Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In E. Cerny and D. Probst, editors, *Correct Hardware Design and Verification Methods (CHARME'97), 9th IFIP WG 10.5 Advanced Research Working Conference, Montreal, Canada, October 1997*, pages 146–166. Chapman and Hall, 1997.
38. J. Ruf and T. Kropf. Modeling and Checking Networks of Communicating Real-Time Systems. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME'99), 10th IFIP WG 10.5 Advanced Research Working Conference, Bad Herrenalb, Germany, September 1999*, pages 265–279. Springer, 1999.
39. J. Ruf, T. Kropf, and R. Weiss. Modeling and Formal Verification of Production Automation Systems. (in this volume).
40. U. Schneider. *Ein formales Modell und eine Klassifikation für die Fertigungssteuerung – Ein Beitrag zur Systematisierung der Fertigungssteuerung*. PhD thesis, Heinz Nixdorf Institut, HNI-Verlagsschriftenreihe, Band 16, Paderborn, Germany, 1996. (in German).
41. S. Sendall and A. Strohmeier. Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML. In M. Gogolla and C. Kobryn, editors, *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference. Toronto, Canada. October 2001*, volume 2185 of *Lecture Notes in Computer Science*, pages 391–405. Springer, 2001.
42. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.
43. J. Warmer and A. Kleppe. *The Object Constraint Language – Getting Your Models Ready for MDA*. Object Technology Series. Addison-Wesley, 2nd edition, 2003.
44. E. Westkämper, M. Höpf, and C. Schaeffer. Holonic Manufacturing Systems (HMS) – Test Case 5. In *Proceedings of Holonic Manufacturing Systems, Lake Tahoe, CA, USA*, February 1994.
45. P. Ziemann and M. Gogolla. An Extension of OCL with Temporal Logic. In J. Jürjens, M. Cengarle, E. Fernandez, B. Rumpe, and R. Sandner, editors, *Critical Systems Development with UML – Proceedings of the UML'02 Workshop*, pages 53–62. Technische Universität München, Institut für Informatik, Munich, Germany, 2002.