A UML Profile for Real-Time Constraints with the OCL

Stephan Flake and Wolfgang Mueller

C-LAB, Paderborn University, Fuerstenallee 11, 33102 Paderborn, Germany
{flake, wolfgang}@c-lab.de

Abstract. This article presents a UML profile for an OCL extension that enables modelers to specify behavioral, state-oriented real-time constraints in OCL. In order to perform a seamless integration into the upcoming UML2.0 standard, we take the latest OCL2.0 metamodel proposal by Warmer et al. [22] as a basis. A formal semantics of our temporal OCL extension is given by a mapping to time-annotated temporal logics formulae.
To give an example of the applicability of our extension, we consider a modeling approach for manufacturing systems called MFERT. We present a corresponding UML profile for that approach and combine both profiles for formal verification by real-time model checking.

1 Introduction

The Object Constraint Language (OCL) is part of the UML since version 1.3. It is an expression language that enables modelers to formulate constraints in the context of a given UML model. OCL is used to specify invariants attached to classes, pre- and postconditions of operations, and guards for state transitions [13, Chapter 6]. But currently, OCL misses 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.

In previous works, we have presented an OCL extension that enables modelers to specify state-oriented real-time constraints [9, 10]. Due to a missing OCL metamodel in the current official UML1.4 specification [13], we took an OCL type metamodel presented in [1] and performed a rather heavyweight extension by directly extending that metamodel. More recently, in reply to the OMG's OCL 2.0 Request for Proposals, an extensive OCL metamodel proposal has been published by Warmer et al. [22] that addresses a better integration of OCL with other parts of UML. Though that proposal has not been adopted by the OMG yet, we apply that metamodel since it comprises the work of several significant contributions concerning the development of OCL in recent years. Based on that metamodel, we present a 'lightweight' approach by defining a UML profile for our temporal OCL extension. For semantics, we present a mapping of (future-oriented) temporal OCL expressions to time-annotated formulae, expressed in a discrete temporal logics called Clocked CTL [18].

As an application example, we consider a manufacturing system. We take an existing notation called MFERT [20] which is dedicated to analysis and design of manufacturing systems, present a UML profile for that notation, and define the semantics for a

UML subset by a mapping to time-annotated Kripke structures, so-called I/O-Interval Structures [19].

The two profiles are integrated by the relation of Clocked CTL formulae and I/O-Interval Structures. It is then possible to apply real-time model checking, i.e., a given model in the MFERT profile notation is checked if it satisfies required real-time properties specified by state-oriented temporal OCL expressions.

2 Related Work

This section gives an overview of approaches that (a) extend OCL for temporal constraints specification or (b) investigate alternative means to express behavioral real-time constraints for UML diagrams.

Ramakrishnan et al. [15] extend OCL by additional rules with unary and binary future-oriented temporal operators (e.g., *always* and *never*) to specify safety and liveness properties. A very similar approach in the area of business modeling that additionally considers past-temporal operators is published by Conrad and Turowski [4]. Kleppe and Warmer [12] introduce a so-called *action clause* to OCL. Action clauses enable modelers to specify required (synchronous or asynchronous) executions of operations or dispatching of events. Similarly, the latest OCL2.0 metamodel proposal introduces *message expressions* [22]. Distefano et al. [6] define *Object-Based Temporal Logic* (BOTL) in order 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 Computational Tree Logic (CTL). Bradfield et al. [3] extend OCL by useful causality-based templates for dynamic constraints. Basically, 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 mu-calculus*, a two-level temporal logic, using OCL on the lower level.

In the domain of real-time specification, there exist two approaches. Roubtsova et al. [16] define a UML profile with stereotyped classes for dense time as well as parameterized specification templates for deadlines, counters, and state sequences. Each of these templates has a structural-equivalent dense-time temporal logics formula in Timed Computation Tree Logic (TCTL). Sendall and Strohmeier [21] introduce timing constraints on state transitions in the context of a restricted form of UML protocol state machines called *System Interface Protocol* (SIP). A SIP defines 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.

All approaches that provide a formal semantics are due to formal verification by model checking. Roubtsova et al., though, do not use OCL for constraint specification in their formal approach, as they argue that *"Any extension of OCL to present properties of computation paths breaks the idea of the language and makes it eclectic"*. In contrast to this, we think that the notion of execution paths can very well be introduced to OCL, as shown in our previous work and this article.

3 UML Profile for Real-Time Constraints with OCL

Due to space limitations, we cannot give a detailed description of OCL in this article. We presume that the reader has basic knowledge of OCL and refer to the *Response to the UML2.0 OCL Request for Proposal, Version 1.5* by Warmer et al. [22] for further information about the language and the metamodel we apply. In the following, we refer to that metamodel as the *OCL2.0 metamodel proposal*. In contrast to current OCL1.4, *nested collections* are now supported in the OCL2.0 metamodel proposal, and our approach significantly relies on this new language feature.

The remainder of this section is organized as follows. In 3.1, we extend the abstract OCL syntax. In particular, new elements are introduced to the OCL2.0 metamodel proposal using the common UML extension mechanisms, i.e., stereotypes, tagged values, and constraints. To support modeling, a concrete syntax and operations have to be defined for this extension on the *M1 (Domain Model) layer* in the UML 4-layer architecture. The grammar of the concrete syntax of [22] is introduced in 3.2 and some new production rules are added, keeping the compliance with existing concrete OCL syntax. Note that we cannot avoid the overlap with the M1 layer in an OCL profile, since OCL pre-defines types and operations on that level. As the concrete OCL syntax only partly provides the operations that are defined in OCL expressions, a standard library of pre-defined OCL operations is specified in [22, Chapter 6]. Correspondingly, we define operations in the context of temporal expressions in 3.3. The semantics of our temporal OCL extension is finally described in Subsection 3.4.

3.1 OCL Metamodel Extensions

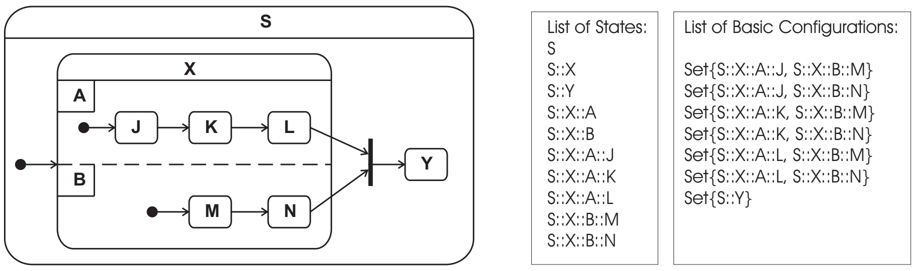
The OCL2.0 metamodel proposal distinguishes two packages. The *OCL type metamodel* describes the pre-defined OCL types and affiliated UML types, while the *OCL expression metamodel* describes the structure of OCL expressions. In the next paragraphs, we further investigate states, state configurations, and sequences of state configurations and introduce respective stereotypes for these concepts.

States. OCL already supports retrieval of states from Statechart diagrams that are attached to user-defined classes. In the OCL type metamodel on layer M2, the respective metaclass is `OclModelElementType`. Generally, `OclModelElementType` represents the types of elements that are `ModelElements` in the UML metamodel. In that case, the model elements are states (or more precisely, instances of a concrete subclass of the abstract metaclass `State`), and the corresponding instance of `OclModelElementType` on layer M1 is `OclState`. For each state, there implicitly exists a corresponding enumeration literal accessible in `OclState`, i.e., `OclState` is seen as an enumeration type on the M1 layer, accumulating the state names of all Statechart diagrams attached to user-defined classes.

The only operation in OCL to access states is of form `obj.oclInState(s)`, with `obj` being an object and `s` being a state name of type `OclState`. The operation returns a Boolean value that indicates whether `s` is currently active in the Statechart for `obj`.

Configurations. The building blocks of Statecharts are hierarchically ordered states. Note that we do not regard pseudo states (like synch, stub, or history states) in this context. A *composite state* is known as a state that has a set of sub-states. A composite state can be a *concurrent state*, consisting of orthogonal regions, which in turn are (composite) states. *Simple states* are non-pseudo, non-composite states.

In a Statechart with composite and concurrent states, a 'current state' cannot be exactly identified, as more than one state can be active at the same time. Consequently, UML1.4 provides the notion of *active state configurations* [13, Section 2.12.4.3] as follows. If a Statechart is in a simple state which is part of a composite state, then all the composite states that (transitively) contain that simple state are also active. As composite states in the state hierarchy may be concurrent, currently active states are represented by a tree of states starting with a unique root state and with individual simple states at the leaves. However, to uniquely identify an active state configuration, it is sufficient to list the comprising simple states, which we denote as a *basic configuration* or *valid configuration*. Figure 1 gives a Statechart example with corresponding basic configurations.



List of States:
S
S::X
S::Y
S::X::A
S::X::B
S::X::A::J
S::X::A::K
S::X::A::L
S::X::B::M
S::X::B::N

List of Basic Configurations:

Set{S::X::A::J, S::X::B::M}
Set{S::X::A::J, S::X::B::N}
Set{S::X::A::K, S::X::B::M}
Set{S::X::A::K, S::X::B::N}
Set{S::X::A::L, S::X::B::M}
Set{S::X::A::L, S::X::B::N}
Set{S::Y}

Fig. 1. Statechart Example with Lists of States and Basic Configurations

For valid configurations, we introduce the new collection type `ConfigurationType` in the context of the OCL type metamodel, as illustrated in Figure 2a[1]. Instances of `ConfigurationType` are restricted sets on the M1 layer. Such sets are restricted to have elements that are enumeration literals of type `OclState`. A possible corresponding formula to express this property in OCL is

```
context ConfigurationType inv:
self.oclIsKindOf(SetType) and self.elementType.name = 'OclState'
```

Execution Paths. Execution paths of Statecharts can be represented by sequences of valid configurations. To model execution paths, we introduce `PathType` in Figure 2a, whose instances are sequences on the M1 level that are restricted to have a certain element type. In that case, the element type must be an instance of `ConfigurationType`. Written as an OCL formula, we require:

```
context PathType inv:
self.oclIsKindOf(SequenceType) and self.elementType.name ='Set(OclState)'
```

[1] For our stereotype definitions, we make use of the graphical notation suggested in the official UML 1.4 specification [13, Sections 3.17, 3.18, 3.35 and 4.3].

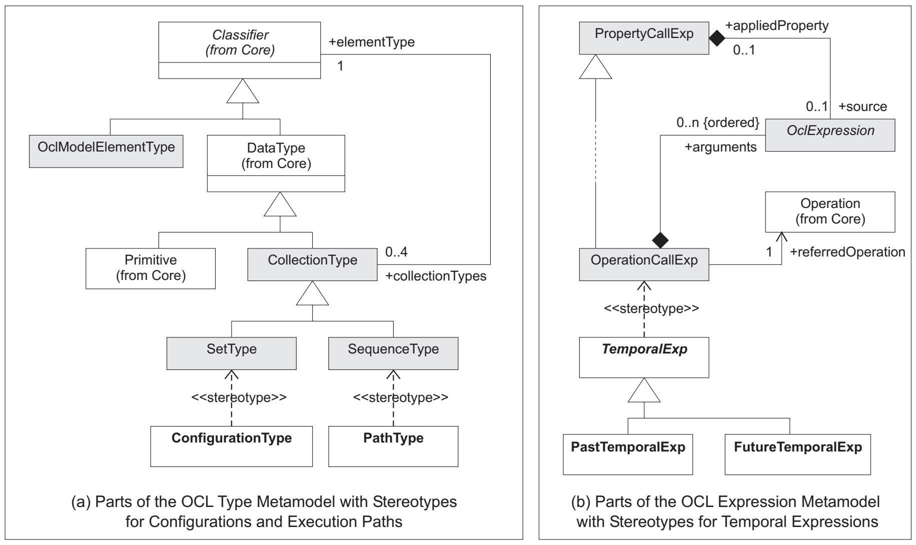


Fig. 2. New OCL Types and Expressions – Gray boxes are taken from the metamodel in [22]

Temporal Expressions. Concerning the OCL expression metamodel, we introduce a new kind of operation call (cf. Figure 2b): `TemporalExp` represents a temporal expression that refers to execution paths. It is the abstract superclass of `PastTemporalExp` and `FutureTemporalExp` for past- and future-oriented temporal expressions. We need these two stereotypes in order to define a semantics for according temporal operations (see Section 3.4).

Execution Path Literals. As we want to reason about execution paths by means of states and configurations, we also need a mechanism to explicitly specify execution paths with annotated timing intervals by literals. We therefore define the stereotype `PathLiteralExp`, as illustrated in Figure 3. The following restrictions apply here, leaving out the corresponding OCL formulae for the matter of brevity.

1. The collection kind of `PathLiteralExp` is `CollectionKinds::Sequence`.
2. Each sequence element has a lower bound and an upper bound.
3. Lower bounds must evaluate to non-negative Integer values.
4. Upper bounds must evaluate to non-negative Integer values or to the String `'inf'` (for *infinity*). In the first case, the upper bound must be greater or equal to the corresponding lower bound.

3.2 Concrete Syntax Changes

Having defined new classes for temporal expressions on the abstract syntax level, modelers are not yet able to use these extensions, as they specify OCL expressions by means of a concrete syntax. In Chapter 4 of the OCL2.0 metamodel proposal, a concrete syntax is given that is compliant with the current OCL1.4 standard. The new concrete syntax is

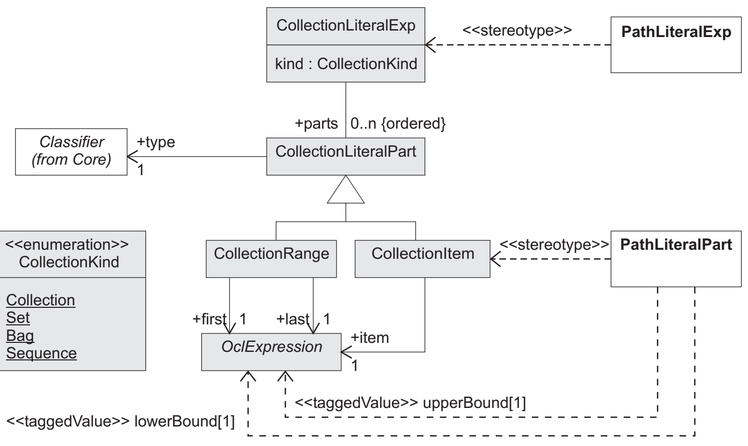


Fig. 3. Parts of the OCL Expression Metamodel with Stereotypes for Execution Paths

defined by an attributed grammar with production rules in EBNF that are annotated with synthesised and inherited 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 according production rule. For instance, each production rule has an inherited attribute `env` (environment) that represents the rule's namespace. *Synthesised attributes* are used to keep results from evaluating the right hand sides of production rules. For instance, each production rule has a synthesised attribute `ast` (abstract syntax tree) that constitutes the formal mapping from concrete to abstract syntax. *Disambiguating rules* allow to uniquely determine a production rule if there are syntactically ambiguous production rules to choose from.

In the following, we present some additional production rules for the concrete syntax of the OCL2.0 metamodel proposal. A mapping to the extended abstract OCL syntax is provided for each new production rule.

OperationCallExpCS[2]

Eight different forms of operation calls are already defined in the OCL2.0 concrete syntax. In particular, it is distinguished between infix and unary operations, operation calls on collections, and operation calls on objects (with or without '@pre' annotation) or whole classes. We additionally introduce rule [J] for temporal operation calls.

```
[A] OperationCallExpCS ::= OclExpressionCS[1]   simpleNameCS
                                                      OclExpressionCS[2]
[B] OperationCallExpCS ::= OclExpressionCS '->' simpleNameCS
                                                    '(' argumentsCS? ')'
[C] OperationCallExpCS ::= OclExpressionCS '.'  simpleNameCS
                                                    '(' argumentsCS? ')'
...
[J] OperationCallExpCS ::= TemporalExpCS
```

[2] All non-terminals are postfixed by 'CS' (short for *Concrete Syntax*) to better distinguish between concrete syntax elements and their abstract syntax counterparts.

We here only list the synthesised and inherited attributes for syntax [J]. Disambiguating rules are defined in the particular rules for temporal expressions.

```
Abstract Syntax Mapping:
    -- (Re)type the abstract syntax tree ast to the according metaclass.
    OperationCallExpCS.ast : OperationCallExp
Synthesised Attributes:
    -- Build the abstract syntax tree (in this case a simple assignment).
    [J] OperationCallExpCS.ast = TemporalExpCS.ast
Inherited Attributes:
    -- Derive the namespace stored in env from left hand side of the rule
    [J] TemporalExpCS.env = OperationCallExpCS.env
```

TemporalExpCS
A temporal expression is either a past- or future-oriented temporal expression.

```
[A] TemporalExpCS ::= PastTemporalExpCS
[B] TemporalExpCS ::= FutureTemporalExpCS
```

We here omit the rather simple attribute definitions. Basically, the abstract syntax mapping defines `TemporalExpCS.ast` to be of type `TemporalExp`, the synthesised attribute `ast` is built from the right hand sides, and the inherited attribute `env` is derived from `TemporalExpCS`.

FutureTemporalExpCS
A future-oriented temporal expression is a kind of operation call. We additionally introduce the symbol '@' to indicate a subsequent temporal operation. Note that an operation call in the abstract syntax has a source, a referred operation, and operation arguments (cf. Figure 2b), so the abstract syntax tree `ast` must be built with corresponding synthesised attributes.

```
FutureTemporalExpCS ::= OclExpressionCS '@' simpleNameCS
                                                    '(' argumentsCS? ')'
Abstract Syntax Mapping:
    FutureTemporalExpCS.ast : FutureTemporalExp
Synthesised 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:
    -- The 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()
```

If other temporal operations than `@post()` need to be introduced at a later point of time, only disambiguating rule [1] has to be modified accordingly. For instance, `@next()` might be introduced as a shortcut for `@post(1,1)`.

A corresponding extension to past temporal operations can be easily introduced, e.g., by means of the operation name `pre()`. In the remainder of this article, we only focus on `FutureTemporalExpCS`. Note that `pre` and `post` as operation names cannot be mixed up with pre- and postcondition labels or the `@pre` time marker, because operations require subsequent brackets.

PathLiteralExpCS

Path literal expressions are a special form of collection literal expressions, as they represent sequences of explicitly specified configurations. In order to allow interval definitions in sequences of configurations, some new production rules have to be formulated. As these are quite similar to the existing production rules for collection literals, definitions of `ast` and `env` are left out for brevity reasons.

```
[A] CollectionLiteralExpCS ::= CollectionTypeIdentifierCS
                                          '{' CollectionLiteralPartsCS? '}'
[B] CollectionLiteralExpCS ::= PathLiteralExpCS
```

In syntax [A], `CollectionTypeIdentifierCS` simply distinguishes between literals for collections (e.g., `'Set'`, `'Bag'`) and `CollectionLiteralPartsCS` is an enumeration of OCL expressions. Option [B] is added to provide a notation for sequences of configurations. For each configuration in a `PathLiteralExpCS`, a timing interval may be associated which specifies how long a configuration is active. If no interval is specified, the bounds are implicitly set to `[0,'inf']`.

```
PathLiteralExpCS      ::= 'Sequence' '{' PathLiteralPartsCS '}'
...
PathLiteralPartsCS[1] ::= PathLiteralPartCS (',' PathLiteralPartsCS[2] )?
...
PathLiteralPartCS     ::= OclExpressionCS IntervalCS?
Abstract Syntax Mapping:
    PathLiteralPartCS.ast : PathLiteralPart
Synthesised Attributes:
    PathLiteralPartCS.ast.item = OclExpressionCS.ast
    PathLiteralPartCS.ast.lowerBound = if IntervalCS->notEmpty()
                                       then IntervalCS.ast.lowerBound
                                       else '0' endif
    PathLiteralPartCS.ast.upperBound = if IntervalCS->notEmpty()
                                       then IntervalCS.ast.upperBound
                                       else 'inf' endif
Inherited Attributes:
    OclExpressionCS.env = PathLiteralPartCS.env
    IntervalCS.env      = PathLiteralPartCS.env
Disambiguating Rules:
    [1] OclExpressionCS.ast.type.oclIsKindOf(ConfigurationType)
```

Basically, intervals are of the syntactical form `[a,b]`, with `a` as a non-negative Integer, and `b` either an Integer with $b \geq a$ or the String literal `'inf'`.

3.3 Standard Library Operations

In our previous work [9], we introduced two new built-in types called `OclConfiguration` and `OclPath` on the M1 layer to handle temporal expressions. We present an alternative approach that avoids to introduce new types and instead operates on the already existing OCL collection types.

Configuration Operations. For configurations as a special form of sets of states, we have to elaborate on operations applicable to sets that return collections, since the resulting collection can be an invalid configuration with an arbitrary set of states. Nevertheless, general collection operations [22, Section 6.5.1] can be directly applied to configurations. These are =, <>, size(), count(), isEmpty(), notEmpty(), includes(), includesAll(), excludes(), and excludesAll(). In addition, iterator operations exists(), forAll(), any(), one() are applicable as well [22, Section 6.6.1]. Other OCL set operations on valid configurations, e.g., union() and intersection(), generally result in arbitrary sets of states rather than in valid configurations. We generally allow such operations, but explicitly mention that they have to be used with care.

Execution Path Operations. Similar to configurations, many of the existing OCL sequence operations can immediately be applied to execution paths. These operations are =, <>, size(), isEmpty(), notEmpty(), exists(), forAll(), includes(), includesAll(), excludes(), excludesAll(), at(), first(), last(), append(), prepend(), subSequence(), asSet(), asBag(), and asSequence(). Correspondingly, some sequence operations may evaluate to invalid execution paths, e.g., select() and collect().

Temporal Operations for OclAny. We introduce `post(a,b)` as a new temporal operation of `OclAny` and allow the @-operator to be used only for such temporal operations. @`post(a,b)` returns *a set of possible future execution paths* in the interval [a,b]. First, all possible execution paths that start with the current configuration are regarded, and then the timing interval [a,b] determines the subpaths that have to be returned by the operation. The result has to be a *set* of paths, as there are typically different orders of executions possible in the future steps of a Statechart. Note that in an actual execution of a Statechart there is of course only exactly one of the possible execution paths selected. An informal semantics of `post(a,b)` is given as follows. Let `obj` be an object in the context of OCL.

```
obj@post(a:Integer,b:OclAny) : Set(Sequence(Set(OclState)))
    pre: a >= 0 and
         ( (b.oclIsTypeOf(Integer) and b >= a) or
           (b.oclIsTypeOf(String) and b = 'inf') )
    The operation returns a set of possible future execution paths in the
    interval [a,b], including the configurations of time points a and b.
```

Further operations, such as @`next()` or @`post(a:Integer)` can be easily added [9]. These are operations basically derived from @`post(a,b)`.

3.4 Semantics of Temporal Expressions

To define the semantics for the future-oriented temporal extension, we provide a mapping from instances of `FutureTemporalExpCS` to Clocked Computational Tree Logic (CCTL) formulae [18]. CCTL is an extension of classical CTL and introduces timing intervals to temporal logic operators.

By definition, OCL invariants for a given type must be true for all its instances at any time [22, Section 2.3.3]. Consequently, corresponding CCTL formulae have to start with the `AG` operator ('On **A**ll paths **G**lobally'), i.e., the expression following `AG` must be true on all possible future execution paths at all times. Table 1 lists OCL operations that directly match to CCTL expressions. In that table, `expr` denotes a Boolean OCL expression. `cctlExpr` is the equivalent Boolean expression in CCTL syntax. `cfg` denotes a valid configuration and `cctlCfg` is the corresponding set of states in CCTL syntax. `p` and `c` are iterator variables for execution paths and configurations, respectively.

Table 1. Mapping Temporal OCL Expressions to CCTL Formulae

Temporal OCL Expression	CCTL Formula
inv: obj@post(a,b)→exists(p | p→forAll(c | expr))	AG EG$_{[a,b]}$(cctlExpr)
inv: obj@post(a,b)→exists(p | p→exists(c | expr))	AG EF$_{[a,b]}$(cctlExpr)
inv: obj@post(a,b)→exists(p | p→includes(cfg))	AG EF$_{[a,b]}$(cctlCfg)
inv: obj@post(a,b)→forAll(p | p→forAll(c | expr))	AG AG$_{[a,b]}$(cctlExpr)
inv: obj@post(a,b)→forAll(p | p→exists(c | expr))	AG AF$_{[a,b]}$(cctlExpr)
inv: obj@post(a,b)→forAll(p | p→includes(cfg))	AG AF$_{[a,b]}$(cctlCfg)

Consider as an example the last row of Table 1. When taking the particular interval `[1,100]` and a configuration from Figure 1 for `cfg`, the resulting OCL expression is:

```
inv: obj@post(1,100)->forAll(p | p->includes(Set{S::X::A::K,S::X::B::M}))
```

We read that formula as: At any time, given the current configuration of the Statechart associated to object `obj`, all future execution paths `p` starting from the current configuration reach – at a certain point of time within the timing interval [a,b] – the configuration represented by `Set`$\{$`S::X::A::K,S::X::B::M`$\}$.[3]

For mapping path literal expressions, let $e_1, e_2, ..., e_n$ be the path literal parts of `PathLiteralExpCS` with timing intervals $[a_i, b_i]$. The temporal OCL expression

`inv: obj@post(a,b)`$\rightarrow$`includes(Sequence`$\{e_1[a_1,b_1], e_2[a_2,b_2], ..., e_n\}$ `)`

maps to CCTL applying the `strong until` temporal operator (i.e., $expr_1$ $\underline{\mathtt{U}}_{[a,b]}$ $expr_2$ requires that $expr_1$ must be true between a and b time units until $expr_2$ becomes true) as follows.

$$\mathtt{AG}_{[a,b]}\ \mathtt{EF(}\ \mathtt{E(}e_1\ \underline{\mathtt{U}}_{[a_1,b_1]}\ \mathtt{E(}e_2\ \underline{\mathtt{U}}_{[a_2,b_2]}\ \mathtt{E(...}\ \mathtt{E(}e_{n-1}\ \underline{\mathtt{U}}_{[a_{n-1},b_{n-1}]}\ e_n\mathtt{)...))))}$$

Though we have given only simplified examples here, more complex should be easily derived from the above.

[3] This kind of formula is often categorized as *liveness property*.

4 MFERT

MFERT[4] provides means for specification and implementation of planning and control assignments in manufacturing processes [20]. In MFERT, nodes represent either storages for production elements or production processes. *Production Element Nodes* (PENs) are used to model logical storages of material and resources and are drawn as annotated shaded triangles. *Production Process Nodes* (PPNs) represent logical locations where material is transformed and are drawn as annotated rectangles. 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 a 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, edges represent the dynamic flow of production elements (i.e., material and resources) within the manufacturing system.

Two examples for MFERT graphs are shown in Figure 4. On the left hand side, a production of tables is illustrated, and on the right hand side, transportation of items between processing steps is modeled. The latter is a small outtake of a model that is composed of different manufacturing stations and transport vehicles (AGVs) that transport items between stations.

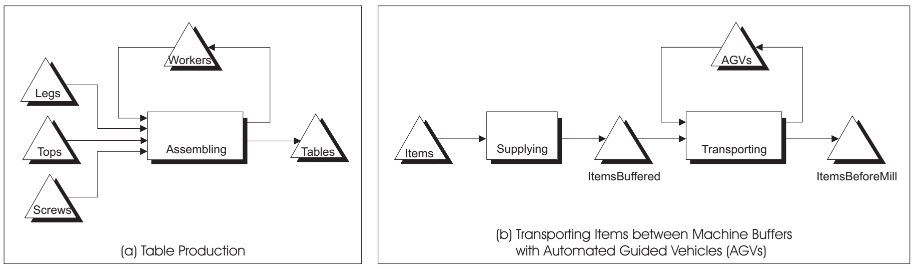


Fig. 4. Sample MFERT Graphs

In MFERT, the processing is specified by finite state machines (FSMs) that are associated with PPNs. Although several variations to formally define the behavior of MFERT PPNs are presented in literature[5], we focus in our work on FSMs and their hierarchical variants like Harel's Statecharts [11]. For instance, an FSM assigned to the node `Assembling` in Figure 4a may specify that the production of a table requires 4 legs, 1 table top, 16 screws, and 1 worker for mounting the table.

[4] MFERT stands for ”Modell der FERTigung” (German for: Model of Manufacturing).

[5] E.g., Quintanilla uses a graphical representation called interaction diagrams and formally defines them in [14].

4.1 UML Profile for MFERT

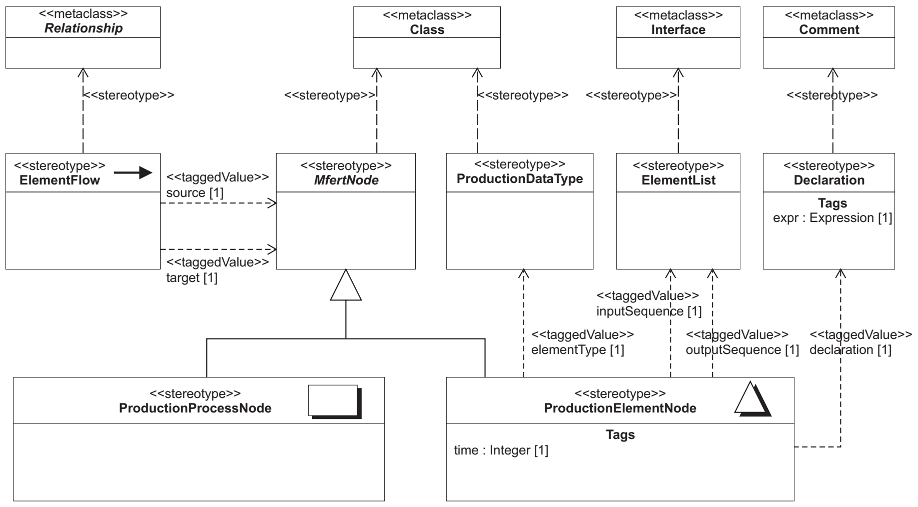
Modeling the static and dynamic aspects of manufacturing systems can be done by UML class diagrams. For that, we introduce stereotypes that represent production processes, storages, and element flow by the virtual metamodel shown in Figure 5, where

1. a `ProductionDataType` defines a tuple of data types. Only query and constructor operations are allowed for production data types, which may only aggregate or be composed of `DataTypes` or `ProductionDataTypes`.
2. The `ElementList` stereotype represents a parameterized interface that provides certain operations to manage lists with elements of a certain `ProductionDataType`. We assume that appropriate operations for lists are specified.
3. `MFERTNode` is the abstract superclass of `ProductionProcessNode` and `ProductionElementNode`. MFERT nodes may only inherit from other MFERT nodes of the same kind. Associations between two MFERT nodes have to be modeled using `ElementFlow` associations. There is at most one relationship between each pair of MFERT nodes, which is either a generalization or an `ElementFlow` association.
4. `ProductionProcessNodes` (PPNs) consume from and send production elements to `ProductionElementNodes`. Each PPN has its own thread of control, i.e., the instances are active objects.
5. `ProductionElementNodes` (PENs) store production elements for further processing by subsequent PPNs. Two lists with production elements (`ElementLists`) are managed by a PEN; one for incoming, one for outgoing production elements. The two lists are storing elements of a certain `ProductionDataType` that is specified by the tagged value `elementType`.
6. `ElementFlow` represents a restricted association between MFERT nodes. Tagged values `source` and `target` refer to the two classifiers that are determined by the association ends of `ElementFlow`. We restrict multiplicity of these association ends to 1, as an `ElementFlow` association shall indicate a relationship between two instances of MFERT nodes. Though it is allowed to navigate on `ElementFlow` associations in both ways, we graphically represent these associations as directed edges towards the target end to indicate the direction of element flow.

Due to space limitations, it is only a summary of the model-inherent restrictions defined by the profile. For a complete outline we refer to [8], where a complete definition of the MFERT profile is given including OCL constraints, the formal MFERT model, and a mapping to I/O-Interval Structures.

Validation/Verification Constraints. For validation and verification of MFERT designs, we have to restrict UML class diagrams as follows.

1. We require that each concrete MFERT node is complete in the sense that its behavior description is given in form of a single Statechart diagram.
2. An MFERT node may communicate with a non MFERT node by operation call, signal or attribute modification, which is not considered for verification.
3. Variables must have a finite value range to be applicable for verification by means of model checking, i.e., attributes can only be enumerations or finite subsets of type Integer.



<<metaclass>> Relationship
<<metaclass>> Class
<<metaclass>> Interface
<<metaclass>> Comment
<<stereotype>>
<<stereotype>> ElementFlow
<<stereotype>> MfertNode
<<stereotype>> ProductionDataType
<<stereotype>> ElementList
<<stereotype>> Declaration
Tags
expr : Expression [1]
<<taggedValue>> source [1]
<<taggedValue>> target [1]
<<taggedValue>> inputSequence [1]
<<taggedValue>> elementType [1]
<<taggedValue>> outputSequence [1]
<<taggedValue>> declaration [1]
<<stereotype>> ProductionProcessNode
<<stereotype>> ProductionElementNode
Tags
time : Integer [1]

Fig. 5. MFERT Stereotypes

4.2 Dynamic Semantics of MFERT

We have identified different execution semantics of MFERT for different purposes, e.g., for analysis, control, or planning [8]. Here, we take the so-called *synchronous semantics* that allows a direct mapping to I/O-Interval Structures. The term *synchronous* in this context refers to immediate receiving of signals without timing delay between different I/O-Interval Structures.

The dynamic semantics for *production process nodes* is defined by an abstract interpreter cycle that controls the execution of the according FSMs. We assume that each PPN has its own thread of control and runs independently of a global signal. Each interpreter cycle starts with selection of a transition with a true condition. After the selection phase, some checks on adjacent PENs are necessary; some actions associated with the selected transition can have an effect on adjacent PENs, e.g., by consuming production elements from preceding nodes. In the final phase, the selected transition fires, a new state is entered and the interpretation cycle starts from the beginning.

For *production element nodes*, cyclic shifting of elements from the input sequence to the output sequence is controlled by an additional interpretation cycle. Queries and manipulations on the sequences are handled independently of the interpretation and are performed with mutual exclusive access to avoid conflicts.

As we restrict all data types to be finite, we can directly map MFERT models as defined by the UML profile in Section 4.1 to the formal MFERT model. PPNs and PENs are directly mapped to I/O-Interval Structures. For I/O-Interval Structures derived from PENs, mainly data types have to be mapped. Transition rules are only necessary for cyclic shifting of elements from the input to the output sequences. For I/O-Interval Structures derived from PPNs, we get the transition rules from the required Statechart diagrams.

4.3 Example

As an example, we take the model of Figure 4b and require that each item that is loaded at buffer `ItemsBuffered` must be transported within 120 time units to the next buffer `ItemsBeforeMill`. In the context of PPN `Transporting`, with its internal states `Loading` and `Unloading`, the corresponding temporal OCL invariant may be

```
context Transporting inv:
self.oclInState(Loading) implies self@post(1,120)->includes(Unloading)
```

For further readings, more examples can be found in [8, 10].

5 Summary and Conclusion

We have presented a UML profile for specification of real-time constraints on the basis of the latest OCL2.0 metamodel proposal. 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. Nevertheless, some extensions have to be made on the 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.

As an example, we applied our temporal OCL extensions to MFERT. A semantics is given to both, MFERT profile and temporal OCL expressions, by a mapping to synchronous time-annotated finite state machines (I/O-Interval Structures) and temporal logics formulae (CCTL), respectively. This provides a sound basis for formal verification by real-time model checking with RAVEN [17].

We have implemented an editor for MFERT [5]. Code generation for I/O-Interval Structures is currently under implementation. The temporal OCL extensions as presented here are integrated into our OCL parser and type checker [9]. The type checker can translate constraints with temporal operations to CCTL formulae.

Our temporal OCL extension has also been used as a specification means for railway systems requirements [2]. In order to investigate the potential for domain-independent application of our approach, we currently map the general *property patterns* identified by Dwyer et al. [7] to according temporal OCL expressions.

Acknowledgements

We thank Henning Zabel for his work on the MFERT editor and the code generator. 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] T. Baar and R. Hähnle. An Integrated Metamodel for OCL Types. In *OOPSLA 2000, Workshop Refactoring the UML: In Search of the Core*, Minneapolis, MN, USA, 2000.

[2] F. Bitsch. Requirements on Methods and Techniques in Perspective to Approval Process for Railway Systems. In *2nd International Workshop on Integration of Specification Techniques for Applications in Engineering (INT'02), Grenoble, France*, April 2002.

[3] J. Bradfield, J. Kuester Filipe, and P. Stevens. Enriching OCL Using Observational mu-Calculus. In *FASE 2002, Grenoble, France*, volume 2306 of *LNCS*. Springer, April 2002.

[4] S. Conrad and K. Turowski. Temporal OCL: Meeting Specifications Demands for Business Components. In *Unified Modeling Language: Systems Analysis, Design, and Development Issues*. IDEA Group Publishing, 2001.

[5] W. Dangelmaier, C. Darnedde, S. Flake, W. Mueller, U. Pape, and H. Zabel. Graphische Spezifikation und Echtzeitverifikation von Produktionsautomatisierungssystemen. In *4. Paderborner Frühlingstagung 2002*, Paderborn, Germany, April 2002. (in German).

[6] D. Distefano, J.-P. Katoen, and A. Rensink. On a Temporal Logic for Object-Based Systems. In *FMOODS'2000*, Stanford, CA, USA, September 2000.

[7] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *ICSE'99, Los Angeles, CA, USA*, May 1999.

[8] S. Flake and W. Mueller. A UML Profile for MFERT. Technical report, C-LAB, Paderborn, Germany, March 2002. URL: http://www.c-lab.de/vis/flake/publications/index.html.

[9] S. Flake and W. Mueller. An OCL Extension for Real-Time Constraints. In *Object Modeling with the OCL*, volume 2263 of *LNCS*, pages 150–171. Springer, February 2002.

[10] S. Flake and W. Mueller. Specification of Real-Time Properties for UML Models. In *Proc. of the 35th Hawaii Internat. Conf. on System Sciences (HICSS-35)*, Hawaii, USA, 2002.

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

[12] A. Kleppe and J. Warmer. Extending OCL to include Actions. In *UML 2000 - The Unified Modeling Language. Advancing the Standard*, volume 1939 of *LNCS*, pages 440–450. Springer, 2000.

[13] OMG. Unified Modeling Language 1.4 Specification. OMG Document formal/2001-09-67, September 2001. URL: http://www.omg.org/technology/documents/formal/uml.htm.

[14] J. Quintanilla de Simsek. *Ein Verifikationsansatz für eine netzbasierte Modellierungsmethode für Fertigungssysteme*. PhD thesis, Heinz Nixdorf Institute, HNI-Verlagsschriftenreihe, Band 87, Paderborn, Germany, 2001. (in German).

[15] S. Ramakrishnan and J. McGregor. Extending OCL to Support Temporal Operators. In *ICSE'99, Workshop on Testing Distributed Component-Based Systems, Los Angeles, CA, USA*, May 1999.

[16] E. E. Roubtsova, J. van Katwijk, W. J. Toetenel, and R. C. M. de Rooij. Real-Time Systems: Specification of Properties in UML. In *ASCI 2001 Conference*, pages 188–195, Het Heijderbos, Heijen, The Netherlands, May 2001.

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

[18] J. Ruf and T. Kropf. Symbolic Model Checking for a Discrete Clocked Temporal Logic with Intervals. In *Conference on Correct Hardware Design and Verification Methods (CHARME'97)*, pages 146–166, Montreal, Canada, October 1997.

[19] J. Ruf and T. Kropf. Modeling and Checking Networks of Communicating Real-Time Systems. In *CHARME'99*, pages 265–279. Springer, September 1999.

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

[21] S. Sendall and A. Strohmeier. Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML. In *UML 2001 – The Unified Modeling Language: Modeling Languages, Concepts, and Tools*, volume 2185 of *LNCS*, pages 391–405. Springer, 2001.

[22] J. Warmer et al. Response to the UML2.0 OCL RfP, Version 1.5 (Submitters: Boldsoft, Rational, IONA, Adaptive Ltd., et al.). OMG Document ad/02-05-09, June 2002.