Semantics of State-Oriented Expressions in the Object Constraint Language

Stephan Flake and Wolfgang Mueller
C-LAB, Paderborn University, Fuerstenallee 11, 33102 Paderborn, Germany
E-mail: {flake,wolfgang}@c-lab.de

Abstract

The textual Object Constraint Language (OCL) is an official part of the Unified Modeling Language (UML). It is primarily used to formulate restrictions for UML class diagrams. Additionally, it is possible to refer to UML Statechart states in OCL expressions to reason about currently activated states.

However, neither the current OCL standard nor the proposal for the new OCL 2.0 version integrate Statecharts on the language definition level, i.e., the semantics of Statechart states in the context of OCL expressions is not sufficiently defined so far. To overcome this deficiency, this article provides a formal semantics for state-oriented OCL expressions for application with UML Statecharts.

1. Introduction

The Object Constraint Language (OCL) is part of the Unified Modeling Language (UML) since UML version 1.3 [5]. OCL is an expression language that enables modelers to formulate constraints in the context of a given UML model. It is mainly used to specify invariants attached to classes and pre- and postconditions of operations. OCL is a declarative language, i.e., evaluation of OCL expressions does not have side effects on the respective UML model.

In the official UML 1.5 specification, a concrete syntax of OCL is given. Due to a missing metamodel only an informal description of the semantics of OCL expressions is provided [5, Chapter 6]. To overcome this, Warmer et al. [8] have recently submitted a proposal with an enhanced OCL language definition for standardization to address a better integration of OCL with other parts of UML. In the remainder, we refer to that document as the *OCL 2.0 proposal*. Though that proposal has not yet officially been adopted, our work is based on that document since it comprises several significant works concerning the development of OCL through the recent years.

OCL enables modelers to formulate expressions intended to check for currently activated Statechart states with a Boolean operation called `oclInState(s:OclState)`. However, the semantics of that operation are still only informally described in the OCL 2.0 proposal, i.e., an integration of UML Statecharts into the language concepts of OCL on the meta level is still missing and there is no other work that provides a formal definition of OCL and takes states of UML Statecharts into account. In this article, we extend OCL by a concise notion of states and provide a formal semantics of OCL's state-related operation `oclInState`.

The remainder of this article is structured as follows. In Section 2, we extend *object models*, i.e., an existing formalization of UML class diagrams, to include UML Statecharts. Based upon that extension, we introduce a notion of Statechart configurations and define a formal semantics for the operation `oclInState` in Section 3. In Section 4, we provide the dynamic semantics of Statecharts by means of sequences of system states over extended object models and discuss when state-oriented OCL constraints have to be checked during execution of (the implementation of) the model. Section 5 concludes the article.

2. Extended Object Models

The OCL 2.0 proposal includes two semantic descriptions. The *normative* description recursively uses UML concepts to define the OCL semantics. It is structured into different packages that constitute the relationship between OCL types and their *value domains*. Evaluation of an OCL expression then returns an element from the domain of the type. The *informative* OCL semantics is provided in form of a set theory-based approach called *object model*, which is based on work of Richters [6]. It covers major parts of the current OCL standard as well as new OCL 2.0 concepts (e.g., nested collections). Unfortunately, the two semantic definitions are not completely consistent, e.g., the `OclMessage` concept is not yet supported in the informative semantics.

But even more significantly, both semantic descriptions do not integrate the notion of Statechart states, although the operation `oclInState` is already defined in the current OCL standard. For example, in the context of a manufactur-

ing scenario with a machine that has a limited input buffer to store items before processing them, the invariant

```
context InputBuffer inv:
    self.oclInState(WaitingForDelivery)
        implies self.storedItems < self.maxItems
```

specifies that there must be at least one vacant input buffer position as long as the buffer is in state `WaitingForDelivery`. The latter state represents the situation that the delivery of an item is expected, but the item has not arrived yet.

Evaluation of an OCL expression is performed over a single *snapshot* or *system state*, i.e., an overall description of the current status of a model. However, two snapshots have to be considered to evaluate operation postconditions when the operator `@pre` is attached to objects or attributes. I.e., in addition to the snapshot after operation execution also values of the snapshot just before the operation execution have to be regarded.

Concerning Statecharts, the OCL 2.0 proposal simply assumes that a dedicated enumeration type called `OclState` represents all state names in the OCL type system. Thus, state names can be used as an argument of the operation `oclInState(s:OclState)`. However, to be able to evaluate an OCL expression that makes use of that operation, the snapshot description must comprise the set of currently activated states of all objects. This aspect is still missing in both semantic descriptions of the OCL 2.0 proposal.

In the remainder of this section, we formally define the syntax of *extended object models* that take Statecharts as a behavioral description of classes. Compared to the original definition of object models, we also newly introduce signal specifications and provide an extension of the formal descriptor of a class.

For that, we start with the definiton of the syntax of an *extended object model* by

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

with sets $CLASS$ of classes, $ATT = \bigcup_{c \in CLASS} ATT_c$ of attributes, $OP = \bigcup_{c \in CLASS} OP_c$ of operations, $SIG = \bigcup_{c \in CLASS} SIG_c$ of signals, $SC = \bigcup_{c \in CLASS} SC_c$ of Statecharts, $ASSOC$ of associations, as well as a generalization hierarchy $\prec$ over classes. The functions $associates$, $roles$, and $multiplicities$ assign for each association $as \in ASSOC$ its classes, role names, and multiplicities, respectively.

In the following, we consider the tuple elements of $\mathcal{M}$ in more detail. For element names in $\mathcal{M}$, let $\mathcal{A}$ be an alphabet and $\mathcal{N} \subseteq \mathcal{A}^+$ a set of finite, non-empty names.

2.1. Types

We assume a set $\Sigma \stackrel{def}{=} (T, \Omega)$, where T is a set of type names and Ω as a set of operations over types in T. Set T comprises types for user-defined classes (T_C), types for enumerations (T_E), and basic standard library types (T_B), such as `Integer`, `Real`, `Boolean`, `String`, and `OclVoid`. The latter is a subtype of any other type that allows operations with unknown values. The only value of `OclVoid` is called `OclUndefined` and represented in the following by symbol $\perp$.

We further denote the value set $I_{Type}(t)$ represented by a type t as the *type domain*. E.g., we have $I_{Type}(OclVoid) = \{\perp\}$. For convenience, we presume that the undefined value is included in each type domain, i.e., $\forall t \in T : \perp \in I_{Type}(t)$. Operations in Ω comprise common arithmetic operations, e.g., +, -, *, / for Integer values. Furthermore, we define so-called *collection types* in Σ to manage collections of values, e.g., `Set(Integer)`.

2.2. Classes and their Characteristics

A class is a description for a set of objects sharing the same characteristics, i.e., attributes, operations, signals, and associations. Note that associations are separately defined in $ASSOC$.

Definition 1 *(Classes and Types)*
$CLASS$ is a finite set of class names, $CLASS \subseteq \mathcal{N}$. Each class $c \in CLASS$ induces a type $t_c \in T_C \subset T$, with the same name as the class. A value $val \in I_{Type}(t_c)$ of a type $t_c \in T_C$ refers to an object of the corresponding class $c \in CLASS$.

Each class c is associated with a set ATT_c of attributes that describe characteristics of their objects. An attribute has a name $a \in \mathcal{N}$ and a type $t \in T$ that specifies the domain of attribute values. Though attribute names of a class must be pairwise distinct, attributes with the same name may appear in several classes, which are not related by generalization. A class may also have a number of operations. Operations are used to describe behavioral characteristics of objects. The behavior can be specified by an associated Statechart; we here only consider *signatures* of operations that declare their interface.

Definition 2 *(Operations)*
The operations of class c are defined by a set OP_c of operation signatures,

$$OP_c \stackrel{def}{=} \{\, (\omega : t_c \times t_1 \times \ldots \times t_n \to t) \mid \omega \in \mathcal{N}, n \in \mathbb{N}_0, \text{ and } t, t_1, \ldots t_n \in T \}.$$

Symbol ω determines the operation name and parameter t_c denotes the type to which operation ω is applied to. In the case that an operation does not return any result, we set the return type to `OclVoid`.

Richters [6] does not consider asynchronous signals that are communicated between objects in the formal object model. However, when integrating Statecharts into the object model, signals have to be included as they are used in Statecharts to trigger state transitions. Though signals are generally defined independently of classes, we here assume that we can build the set SIG_c that comprises the signals, which are handled by objects of a given class c.

Definition 3 *(Signals)*
The signals that can be handled by instances of a class c are defined by the set SIG_c of signal signatures,

$$SIG_c \stackrel{def}{=} \{ (\omega : t_c \times t_1 \times \ldots \times t_n) \mid \omega \in \mathcal{N}, n \in \mathbb{N}_0, \text{ and } t_1, \ldots t_n \in T \}.$$

Symbol ω denotes the signal name and t_c refers to the type that signal ω is applied to. As signals are asynchronous, no return value is expected, such that all signal parameters are input parameters.

2.3. Abstract Syntax of Statecharts

UML 1.5 specifies concepts for modeling discrete behavior through finite state-transition systems [5, Section 2.12]. The provided state machine formalism is an object-based variant of Harel Statecharts [4]. Generally, state machines are applicable to various model elements within UML. We here focus on UML Statechart Diagrams, which are used to model the *reactive behavior of objects*, so that only this Statechart interpretation is relevant for OCL's state-related operation `oclInState`. The UML 1.5 specification only informally defines the dynamic semantics of Statecharts by means of natural language phrases and leaves the semantics open for interpretation, e.g., the dispatching policy for selecting events from the implicit event queue. Reaction on that selected event is performed in a so-called *run-to-completion step* (RTC-step). In that context, numerous approaches have been published to formally define the execution semantics of UML Statecharts [1].

The abstract Statechart syntax defined in the following comprises all relevant information, which is required to fully capture the definition of a *configuration* of a Statechart, i.e., a complete description of what is informally known in UML by an *active state configuration* that is reached after completion of an RTC-step [5, Section 2.12.4.3]. We here do not make the assumption *how* an RTC-step is executed to reach the next configuration. This must be formally defined in the dynamic semantics of each individual approach.

Definition 4 *(Abstract Syntax of Statecharts)*
Each $c \in CLASS$ can have an associated Statechart SC_c representing the reactive behavior of instances of c. If c does not have an associated Statechart, we set $SC_c := \varnothing$, otherwise SC_c is a tuple

$$SC_c \stackrel{def}{=} \langle\, S_c, EVTS_c, GUARDS_c, ACTS_c, TR_c, internalTrans_c, substates_c, entry_c, exit_c, doActivity_c, deferrableEvents_c \,\rangle$$

For the matter of brevity, we omit the class annotator c for all Statechart components in the remainder of this article. We define the components of a Statechart SC as follows.

1. $S \subseteq \mathcal{N}$ is a set of states. S is the union of the following disjoint sets
   - pseudo states $Pseudo$, consisting of seven disjoint sets (a) initial states $Init$; (b) shallow history states $History$; (c) deep history states $DeepHistory$; (d) merging states $Join$; (e) splitting states $Fork$; (f) static conditional branch states $Junction$; (g) dynamic conditional branch states $Choice$;
   - synchronization states $Synch$;
   - simple states $Simple$;
   - composite states $Composite$, composed of the two disjoint sets of sequential composite states Xor and orthogonal composite states And;
   - final states $Final$;

   Details are given in [5, Section 2.12.2]. For convenience, we define
   $$Proper \stackrel{def}{=} And \cup Xor \cup Simple.$$
2. $EVTS \subseteq EXPR_{Evts}$ is a set of events. We presume an expression language $EXPR_{Evts}$ for the definition of events such as operation calls, signals, timers, etc. Correspondingly, we presume expression languages for the set of $GUARDS$ of transition conditions and the set $ACTS$ of transition actions (e.g., assignments, operation calls, signals).
3. $TR \subseteq (S \setminus Final) \times EVTS \times GUARDS \times ACTS \times (S \setminus Init)$ is a set of transitions. A transition has a source state $s \in S \setminus Final$ and a destination state $s' \in S \setminus Init$; it may have a trigger event $e \in EVTS$, a guard condition $g \in GUARDS$, and an action expression $a \in ACTS$.
4. $internalTrans : Proper \rightarrow \mathcal{P}(EVTS \times GUARDS \times ACTS)$ gives the set of *internal transitions* for a given state $S \in Proper$. When triggering an internal transition in a state s, the entry- and exit-actions of s are not executed.

5. $substates : Composite \rightarrow \mathcal{P}(S)$ gives all substates of a state, such that
   (a) there is a unique state $top \in Composite$ such that $\forall s \in Composite : top \notin substates(s)$,
   (b) $\forall s \in And : substates(s) \subseteq Composite$,[1]
   (c) $\forall s \in Composite \setminus \{top\}$ there is exactly one path $\langle s_1, \ldots, s_n \rangle \in Composite^n$, such that $s_1 = top \wedge s_n = s \wedge s_{i+1} \in substates(s_i)$ for $1 \leq i \leq n-1$.
6. Functions $entry, doActivity, exit : Proper \rightarrow ACTS$ give the actions which are executed when a state is entered, active, or left, respectively.
7. $deferrableEvents : Proper \rightarrow \mathcal{P}(EVTS)$ gives the set of events to be retained for later consumption.

Definition 4 covers most of the abstract Statechart syntax as defined in the official UML 1.5 specification [5, Section 2.12.2]. We only leave out minor details, which are of no importance in the remainder of this article, e.g., local variables and the bounds of synch states.

2.4. Associations

Associations are used to model structural relationships between classes. We give no formal definitions for associations, role names (i.e., association-end names), association multiplicities, and their syntactical restrictions here, since they are kept unchanged due to the definition in [6]. In the following, we will make use of function $navEnds : CLASS \rightarrow \mathcal{P}(\mathcal{N})$ to denote the set of role names that can be directly accessed from a given class by navigating the associations this class participates in.

2.5. Generalization

By generalization, we refer to a taxonomic relationship between two classes, in which a general class is specialized into a more specific class.

Definition 5 *(Generalization, Child and Parent Classes)*
A generalization hierarchy $\prec$ is an irreflexive partial order on $CLASS$, i.e., $\prec$ is an irreflexive, anti-symmetric, and transitive relation. Pairs in $\prec$ describe generalization relationships between two classes.

For $c_1, c_2 \in CLASS$ with $c_1 \prec c_2$, c_1 is called a child class of c_2, and c_2 is called a parent class of c_1. A child class transitively inherits characteristics (attributes, operations, signals, and associations) of its parent classes.

[1] This is a well-formedness rule of the UML standard (see [5, Section 2.12.3.1]). In many alternative formal syntax definitions, even $s' \in Xor$ is required in this case, yielding a *normal form* of alternating Xor- and And-states in the state hierarchy.

We define function $parents : CLASS \rightarrow \mathcal{P}(CLASS)$ for collecting all transitive parents of a given class by $parents(c) \stackrel{def}{=} \{c' \mid c' \in CLASS \wedge c \prec c'\}$. The complete set of attributes of c is then defined by

$$ATT_c^* \stackrel{def}{=} ATT_c \cup \bigcup_{c' \in parents(c)} ATT_{c'}.$$

The complete sets OP_c^*, SIG_c^*, and $navEnds^*(c)$ of operations, signals, and navigable role names are defined accordingly.

While the problem of consistency among generalization of classes and inheritance of characteristics for attributes and operations has been widely studied for object-oriented languages, *consistency among inheritance of behavior* in object-oriented design notations like UML has received less attention. UML only provides an informal description of three different inheritance policies for state machines [5, Section 2.12.5.3]. In contrast, different formal notions for behavioral consistency have already been identified in [2, 7].

As we cannot restrict our general definition to a specific form of behavioral inheritance consistency, we assume that there is some suitable policy provided that guarantees behavioral consistency among the Statecharts that participate in a generalization hierarchy.

2.6. Full Descriptor of a Class

The UML standard restricts the abstract syntax of models by well-formedness rules, e.g., a class may not define an operation, attribute, or role name that is already defined in one of its parent classes. Such constraints are already formally captured in [6] by the *full descriptor of a class*. We here extend that notion by signals and Statecharts.

Definition 6 *(Full Descriptor of a Class)*
The full descriptor of a class $c \in CLASS$ is a tuple

$$FD_c \stackrel{def}{=} \langle ATT_c^*, OP_c^*, SIG_c^*, SC_c, navEnds^*(c) \rangle$$

containing all attributes, operations, signals, navigable role names, and an (optional) associated Statechart.

The following well-formedness rules apply for signals and Statecharts in the full descriptor:

1. A signal may only be defined once in a full class descriptor. Formally, we require that two signals with same names and parameter types are automatically defined for the same type.

$$\forall\, (\omega : t_c \times t_1 \times \ldots \times t_n) \in SIG_c^*,$$
$$\forall\, (\omega' : t_{c'} \times t'_1 \times \ldots \times t'_n) \in SIG_c^* :$$
$$(\omega = \omega' \wedge t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n) \implies t_c = t_{c'}.$$

2. Operation and signal names (in combination with the corresponding parameters) must be pairwise distinct.

$$\forall\,(\omega : t_c \times t_1 \times \ldots t_n \to t) \in OP_c^*,$$
$$\forall\,(\omega' : t_{c'} \times t_1 \times \ldots t_n) \in SIG_c^* : \;\; \omega \neq \omega'.$$

Note that types $t_1, \ldots, t_n$ are fixed for ω' by the parameter types of ω.

3. To guarantee syntactical consistency among Statecharts and class definitions, each operation call expression in Statechart SC_c must have an associated operation signature specified in $OP_{c'}^*$, where c' is the class of the called object. Correspondingly, each signal expression must have an according signature in $SIG_{c'}^*$.

Since we abstract from a particular expression syntax of $EVTS_c$ and ACT_c, we do not give a formalization in more details here.

Note that it is allowed for operations and signals to have the same name as attributes or role names, because the concrete syntax of OCL allows to distinguish between both.

3. Static Semantics

Based on the general syntax of extended object models, we now define *system states*, i.e., a formal description of objects and their characteristics in an instantiation of an extended object model $\mathcal{M}$ at a particular point of time. We first have to define the *domain* of classes by means of object identifiers and state configurations.

The domain of a class $c \in CLASS$ is the set of objects of this class and all child classes. We refer to objects by identifiers that are unique in the context of the whole system. In the remainder, no distinction will be made between objects and their identifiers.

Definition 7 *(Object Identifiers and Domain of a Class)* *The set of object identifiers of a class $c \in CLASS$ is defined by an infinite set $oid(c) \stackrel{def}{=} \{\underline{oid}_1, \underline{oid}_2, \ldots\}$. The domain of a class $c \in CLASS$ is defined as*

$$I_{CLASS}(c) \stackrel{def}{=} \bigcup_{c' \in CLASS|_{(c' \prec c \;\vee\; c' = c)}} oid(c').$$

3.1. State Configurations

For UML Statecharts, the term 'current state' cannot be applied without disambiguities, as Statecharts can have composite (i.e., nested and orthogonal) states and thus may reside in more than one state at the same time. UML therefore provides the notion of *active state configurations* as follows.

If the Statechart is in a simple state that is contained in a composite state, then all the composite states that (transitively) contain the simple state are active as well. Furthermore, since composite states in the state hierarchy may be concurrent, the currently active states are represented by a tree of states starting with the single state top_c at the root down to individual simple leaf states $s_i \in Simple_c$. Definition 8 defines state configurations where we make use of function $superstate_c$ that gives the direct superstate of a state $s \in S_c$:

$$superstate_c : \begin{cases} S_c \to Composite_c \\ s \mapsto \begin{cases} s', \text{ if } \exists s' \in Composite_c \\ \quad \text{with } s \in substates_c(s'), \\ \varnothing, \text{ else} \end{cases} \end{cases}$$

In contrast to UML, we also consider final states in state configurations, since final states might be active after an RTC-step. Note here that a final state, which is a direct child of the outermost state top_c is not part of any state configuration, since entering that state is equivalent to the destruction of the corresponding object. Additionally, we explicitly exclude *immediate states*. Immediate states are proper states that have an outgoing triggerless transition and no associated activity; they are entered and directly left within the same RTC-step. Consequently, immediate states can never be part of a state configuration. Thus, we simply refer to the set $Immediate_c$ that includes all immediate proper states of a Statechart SC_c and omit a formal definition. Furthermore, we make use of the following help sets for classes $c \in CLASS$ with $SC_c \neq \varnothing$:

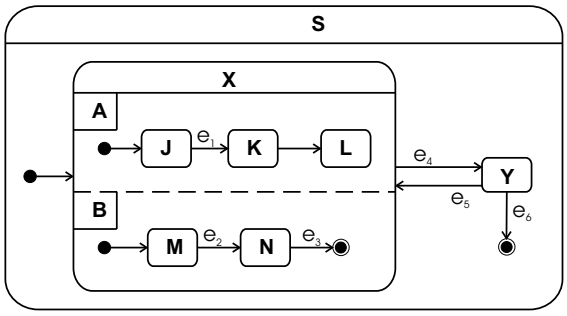
$$\begin{aligned} ProperStay_c &\stackrel{def}{=} Proper_c \setminus Immediate_c, \\ Stay_c &\stackrel{def}{=} ProperStay_c \cup \\ &\qquad \{f \in Final_c \mid f \notin substates_c(top_c)\}, \end{aligned}$$

Definition 8 *(State Configurations for a State)* *Let $c \in CLASS$ with $SC_c \neq \varnothing$. A* state configuration $\mathcal{C}$ with respect to a state s *is a maximal set of states that the Statechart can be simultaneously in with state s as a root state. Function cfg_c that maps a state $s \in ProperStay_c$ to the set of configurations $\mathcal{C}$ with respect to s is defined by*

$$cfg_c : \begin{cases} ProperStay_c \to \mathcal{P}(\mathcal{P}(Stay_c)) \\ s \mapsto \{\, \mathcal{C} \subseteq \mathcal{P}(Stay_c) \mid s \in \mathcal{C} \wedge \\ \qquad \forall s' \in \mathcal{C} \cap And_c : substates_c(s') \subseteq \mathcal{C} \wedge \\ \qquad \forall s' \in \mathcal{C} \cap Xor_c : |substates_c(s') \cap \mathcal{C}| = 1 \\ \qquad \wedge \forall s' \in \mathcal{C} \setminus \{s\} : superstate_c(s') \in \mathcal{C} \,\} \end{cases}$$

Definition 9 *(State Configurations)* *The set $I_{SC}(c)$ of overall* state configurations *for a class $c \in CLASS$ with $SC_c \neq \varnothing$ is determined by $cfg_c(top_c)$. For convenience, we define $I_{SC}(c)$ for all $c \in CLASS$ by*

$$I_{SC}(c) \stackrel{def}{=} \begin{cases} cfg_c(top_c) & \text{if } SC_c \neq \varnothing \\ \varnothing & \text{if } SC_c = \varnothing \end{cases}.$$




```
Proper States:          { S,X,Y,A,B,J,K,L,M,N }
Final States:           { S::FinalState, B::FinalState }
Immediate States:       { K }
State Configurations: { {S,X,A,B,J,M}, {S,X,A,B,J,N},
                        {S,X,A,B,J,B::FinalState},
                        {S,X,A,B,L,M}, {S,X,A,B,L,N},
                        {S,X,A,B,L,B::FinalState},
                        {S,Y} }
```

Figure 1. Statechart Example

Example. Figure 1 gives a Statechart example with state configurations. All proper states except immediate state `K` have an outgoing transition with a specified triggering event e_i, $1 \leq i \leq 6$. To textually refer to final states in UML Statecharts, we have introduced `FinalState` as a new keyword. Note here that `S::FinalState` is not part of the configuration set.

3.2. System State

In the following, we call an instantiation of an extended object model a *system*. A system can be in different states as it varies over time, i.e., the number of objects, their attribute values, Statechart configurations, and other characteristics change when executing the system. To define system details, we have to define what a single system state exactly consists of. It is important to point out here that different notions of a system state are generally possible. This is due to the scope of model analysis one wants to perform.

In Richters' original work on object models, a system state is a tuple consisting of (a) the current set of objects, (b) their attribute values, and (c) the current connections between objects (so-called *links*) based on the set of associations [6, Section 5.2]. However, as Richters did not consider Statecharts, he cannot manage state-related operations and thus does not provide the semantics for the OCL standard operation `oclInState`. We therefore have to extend system states by a notion of state configurations.

Additionally, for a concise definition of *system state sequences* (see Section 4) we need to define the control flow between operations (for OCL preconditions) and which operations are terminated lately (for OCL postconditions). For that, we adopt ideas of [9] to formalize this aspect by introducing the tuple components Σ_{OP} and Σ_{PARAM}.

Definition 10 *(System State)*
A system state for an extended object model $\mathcal{M}$ is a tuple

$$\sigma(\mathcal{M}) \stackrel{def}{=} \langle\, \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{CONF}, \Sigma_{ASSOC}, \Sigma_{OP}, \Sigma_{PARAM} \,\rangle, \;\; where$$

1. $\Sigma_{CLASS} \stackrel{def}{=} \bigcup_{c \in CLASS} \Sigma_{CLASS,c}$ comprises the finite sets $\Sigma_{CLASS,c} \subset oid(c)$ that contain all currently existing objects of a class $c \in CLASS$.

2. The current attribute values are kept in Σ_{ATT}. This set consists of functions $\sigma_{ATT,a}$, where index 'a' stands for an attribute of a given class $c \in CLASS$. Formally, we have $\Sigma_{ATT} \stackrel{def}{=}$

$$\bigcup_{c \in CLASS} \{\sigma_{ATT,a} : \Sigma_{CLASS,c} \rightarrow I_{Type}(t) \mid \langle a, t_c, t \rangle \in ATT_c^* \}.$$

3. The current Statechart configurations are captured by

$$\Sigma_{CONF} \stackrel{def}{=} \bigcup_{c \in CLASS} \{\sigma_{CONF,c} : \Sigma_{CLASS,c} \rightarrow I_{SC}(c)\}.$$

   Each function $\sigma_{CONF,c}$ assigns a state configuration to each object $\underline{oid}$ of a given class $c \in CLASS$. If $SC_c = \varnothing$. We set $\sigma_{CONF,c}(\underline{oid}) := \varnothing$.

4. $\Sigma_{ASSOC} \stackrel{def}{=} \bigcup_{as \in ASSOC} \Sigma_{ASSOC,as}$ comprises the finite sets $\Sigma_{ASSOC,as}$ that contain links that connect objects. A set of links $\Sigma_{ASSOC,as}$ must satisfy all multiplicity specifications defined for an association as. A formalization of this requirement can be found in [6, page 47].

5. $\Sigma_{OP} \stackrel{def}{=} \bigcup_{c \in CLASS} \{\sigma_{OP,c} : \Sigma_{CLASS,c} \times OP_c \rightarrow \mathcal{P}(\mathbb{N})\}$ is a set of functions $\sigma_{OP,c}$, where each of those functions determines the identities of currently executed operations for a given object $\underline{oid}$ and operation name op. The identity of a currently executed operation is implicitly given by an associated natural number that must not change during execution of that operation.

6. $\Sigma_{PARAM} \stackrel{def}{=}$

$$\bigcup_{c \in CLASS} \{\, \sigma_{PARAM,c} : \Sigma_{CLASS,c} \times OP_c \times \mathbb{N} \rightarrow I_{Type}(t_1) \times ... \times I_{Type}(t_n) \times I_{Type}(t) \}$$

   is a set of functions that give the parameter values of each of the currently executed operations. For each $c \in CLASS$, we define function $\sigma_{PARAM,c}$ as follows, where $op = (\omega : t_c \times t_1 \times \ldots \times t_n \rightarrow t) \in OP_c$:

$$\sigma_{PARAM,c}(\underline{oid}, op, i) \mapsto \begin{cases} \langle val(t_1), ..., val(t_n), val(t) \rangle, & \text{if } i \in \sigma_{OP,c}(\underline{oid}, op) \\ \varnothing, & \text{otherwise} \end{cases}$$

In the definition above, $val(t_j) \in I_{Type}(t_j)$ denotes an arbitrary value defined for type $t_j \in T$, $1 \leq j \leq n$. The same holds for $val(t) \in I_{Type}(t)$. If an operation does not return a result, the result type t of operation op is `OclVoid`. In that case, we set $val(t) := \perp$.

There are additional Statechart characteristics that can also be taken into account to be part of a system state, e.g., event queues and changes occurring to them, additional information required for re-entering composite states via history states, etc. However, the definition presented here is sufficient to reason about currently activated states and executed operations and thus completely fulfills our present needs.

3.3. Semantics of oclInState

The signature of the operation `oclInState` is given by $oclInState : OclAny \times OclState \rightarrow Boolean$, where $OclAny$ is the supertype of all basic OCL types T_B, enumeration types T_E, and user-defined types T_C. The domain of $OclAny$ is formally defined by

$$I_{Type}(OclAny) = (\bigcup_{t \in T_B \cup T_E \cup T_C} I_{Type}(t)) \cup \{\perp\}.$$

The semantics of `oclInState` over a given system state $\sigma(\mathcal{M})$, an existing object $\underline{oid} \in \Sigma_{CLASS,c}$, and a state literal $s \in I_{Type}(OclState)$ is then defined by function

$$I_{(oclInState)}(\underline{oid}, s) \stackrel{def}{=} \begin{cases} true, & \text{if } SC_c \neq \varnothing \\ & \quad \wedge\, s \in Stay_c \\ & \quad \wedge\, s \in \sigma_{CONF,c}(\underline{oid}), \\ false, & \text{if } SC_c \neq \varnothing \\ & \quad \wedge\, s \in Stay_c \\ & \quad \wedge\, s \notin \sigma_{CONF,c}(\underline{oid}), \\ \perp, & \text{if } SC_c \neq \varnothing \\ & \quad \wedge\, s \notin Stay_c \cup \{\perp\}, \\ \perp, & \text{if } SC_c = \varnothing, \\ \perp, & \text{if } s = \perp\ . \end{cases}$$

Note here that `oclInState` returns $\perp$ when there is no associated Statechart SC_c or when state s is not a suitable proper or final state of SC_c. It would also be possible to return $false$ instead, but neither UML 1.5 nor the OCL 2.0 proposal give any information about this issue.

4. Dynamic Semantics

We can now consider sequences of system states as *traces*. At this point, we have to decide and formally define a *valid* trace, i.e., when a new system state has to be appended to the trace at execution time. When checking OCL constraints, we are not interested in every single attribute value change that occurs during execution of an operation. Instead, we are interested in system states in which an operation has been completed or a signal has been consumed.

In the simplest case, i.e., when (an implementation of) the system is executed on a single CPU, there is a clear temporal relationship among subsequent trace elements. But when (the implementation of) the system is distributed, we have a partial order between configurations of different objects. This problem can be solved, for instance, by introducing a *global clock*.

Definition 11 *(Trace)*
A well-defined system state sequence called trace *for an instantiation of an extended object model $\mathcal{M}$ is an (infinite) sequence of system states as defined in Definition 10,*

$$trace(\mathcal{M}) \stackrel{def}{=} \langle\, \sigma(\mathcal{M})_{[0]}, \sigma(\mathcal{M})_{[1]}, \ldots, \sigma(\mathcal{M})_{[i]}, \ldots \,\rangle$$

We assume that $\sigma(\mathcal{M})_{[0]}$ denotes the initial system state. Given a system state $\sigma(\mathcal{M})_{[i]}$, $i \in \mathbb{N}_0$, the next system state $\sigma(\mathcal{M})_{[i+1]}$ is added to the trace when one of the following holds during execution:

- *an operation is called,*
- *an operation has terminated,*
- *a new Statechart state configuration is reached.*

In the remainder, we apply the $[i]$-annotation also for the components and functions of the tuple $\sigma(\mathcal{M})_{[i]}$, $i \in \mathbb{N}_0$. We presume that the following restrictions apply to traces:

1. Two adjacent sequence elements may differ in at most one begin or end of operation execution *per object*. We denote the overall number of current operation executions for an object $\underline{oid}$ of class c in system state $\sigma(\mathcal{M})_{[i]}$ by

   $$\psi(\underline{oid})_{[i]} \stackrel{def}{=} \sum_{op \in OP_c} |\sigma_{OP,c}(\underline{oid}, op)_{[i]}|.$$

   For each pair of adjacent system states $\sigma(\mathcal{M})_{[i]}$ and $\sigma(\mathcal{M})_{[i+1]}$, $i \in \mathbb{N}_0$, in $trace(\mathcal{M})$, it must hold that

   $$\forall c \in CLASS, \forall \underline{oid} \in \Sigma_{CLASS,c[i]} : \\ \underline{oid} \in \Sigma_{CLASS,c[i+1]} \implies \\ abs(\psi(\underline{oid})_{[i]} - \psi(\underline{oid})_{[i+1]}) \leq 1$$

2. Each operation call occurring in the trace must eventually be terminated, i.e., for all operations $op \in OP_c$ of an object $\underline{oid}$ in a system state $\sigma(\mathcal{M})_{[i]}$, $i \in \mathbb{N}_0$, it must hold that

   $$\forall execOp \in \sigma_{OP,c}(\underline{oid}, op)_{[i]}\ \exists j \in \mathbb{N}, j > i : \\ execOp \notin \sigma_{OP,c}(\underline{oid}, op)_{[j]} \wedge \\ \forall i \leq k \leq j : \underline{oid} \in \Sigma_{CLASS,c[k]}$$

The formula above requires additionally that an object must not be destroyed when one of its operations is still executed.

3. Values of input parameters of operations must not change, as OCL only accepts input parameters for operations. For operation return types $t \neq OclVoid$, the value of the (implicitly defined) result variable changes from $\perp$ to a well-defined value $val(t) \in I_{Type}(t)$ when the operation call terminates.

Traces as defined above should be seen as a rather general approach to capture those parts of the system runtime information that is necessary to reason about (sequences of) system states.

Evaluating OCL Constraints. It is quite obvious when to check pre- and postconditions for operations, i.e., just before and just after execution of the according operation. Invariants of an object, in contrast, have to be checked each time the system state changes w.r.t. that object. It is often assumed that the status of an object changes only through operation calls. While this might be suitable in some application domains, the situation becomes different when objects are modeled in combination with Statecharts. In UML Statecharts, *elapsed time events* and *change events* can be specified to trigger transitions, e.g., `after(1 sec)` or `when(x > 100)`. These are basically monitors that permanently check for a condition to become true and then raise an internal event to trigger the according transition in the next RTC-step. Thus, a new Statechart configuration is entered without any operation call. Similarly, signals consumed by an RTC-step also cause a new Statechart configuration to be entered. Invariants must therefore be checked in such cases as well. More formally, we evaluate invariants for an object $\underline{oid} \in \Sigma_{CLASS,c[i]}$ at location i of $trace(\mathcal{M})$, $i \geq 1$, when the following holds

$$\begin{array}{rl} \underline{oid} \notin \Sigma_{CLASS,c[i-1]} \vee & \text{// object is new} \\ \sigma_{CONF,c}(\underline{oid})_{[i]} \neq \sigma_{CONF,c}(\underline{oid})_{[i-1]} \vee & \\ \psi(\underline{oid})_{[i]} \neq \psi(\underline{oid})_{[i-1]}. & \end{array}$$

Additionally, for $i = 0$, we check invariants for all objects that exist in the initial state $\sigma(\mathcal{M})_{[0]}$.[2]

5. Conclusion

Statechart states are already used in OCL on the syntactical level, but their semantics in the context of OCL expressions have not been sufficiently investigated so far. We therefore have formalized UML Statechart configurations and integrated them into a formalization of UML class diagrams. Based on that, we have defined a semantics for OCL expressions that make use of the OCL standard operation `oclInState`. This is an essential step towards completing the formal semantics of OCL to make it applicable for state-based reasoning. However, a formalization of the OCL message concept is still missing.

The presented work is a generalization of our previous work that introduced a state-oriented *temporal* OCL extension [3]. Our ongoing work focuses on formal verification by model checking over (parts of) UML models with respect to temporal OCL constraints for property specification.

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

[2] Note here that the check point is not defined by OCL.

References

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

[2] J. Ebert and G. Engels. Observable or Invocable Behaviour: You have to Choose. Technical report, Universität Koblenz, Koblenz, Germany, 1994.

[3] S. Flake and W. Mueller. A UML Profile for Real-Time Constraints with the OCL. In *UML 2002 - The Unified Modeling Language. 5th International Conference, Dresden, Germany*, volume 2460 of *LNCS*, pages 179–195. Springer, 2002.

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

[5] OMG. Unified Modeling Language 1.5 Specification. OMG Document formal/03-03-01, March 2003. URL: http://www.omg. org/technology/documents/formal/uml.htm.

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

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

[8] J. Warmer et al. 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.

[9] P. Ziemann and M. Gogolla. An Extension of OCL with Temporal Logic. In *Critical Systems Development with UML*, pages 53–62. Technische Universität München, Institut für Informatik, 2002.