Using Description Logic in Object-Oriented Software Development

Ragnhild Van Der Straeten
System and Software Engineering Lab, Vrije Universiteit Brussel
Brussels, Belgium
Email: rvdstrae@vub.ac.be

Abstract

The blueprint of an object-oriented software application consists mainly of models drawn in a modeling language. Our goal is to develop a formal framework to support the semantic linking of these models within the software development life cycle (SDLC) using Description Logic (DL). In this paper, the translation of the state diagrams of the Unified Modeling Language (UML) and of constraints written in the Object Constraint Language (OCL) to the DL $\mathcal{DLR}$ is described. $\mathcal{DLR}$ is choosen because in [4], UML class diagrams are translated in this DL.

1 Introduction

The major activities in software development are the determination and specification of requirements the system should meet, the architectural design which is a description of the system in terms of its modules, the design of the system and at last the implementation of the system.

Nowadays, the de-facto modeling language used in object-oriented development is the Unified Modeling Language (UML) [1]. The visual representation of this language consists of several diagram types. All major UML case-tools support different UML diagram types. Several levels of abstraction are used in the SDLC. Within one abstraction level several kinds of UML diagrams are used, e.g. class diagrams, state diagrams, interaction diagrams. In this paper, one such layer will be called a model of the application. What is not supported in the case-tools is the semantic linking between the different diagrams of a model and between the different models. As a result, the different models are less maintainable, reusable and understandable. Nowadays models are only used as a kind of documentation, in the extreme case, changes to the software application are not reflected in the different models and these models become obsolete.

Our goal is to develop a formal framework to support the semantic linking of the different diagram types within a model and of the different models. The advantages of a formal framework are: the provided reasoning capabilities about diagrams and models in the SDLC, through the semantic links the co-evolution of the models is more guaranteed, reuse and adaptability of software is improved, the understandability of the software designs increases. We believe that Description Logic can be used as a formal framework.

There are several reasons for choosing the Description Logic family as formalism. Object-oriented programming languages originate from frame-based systems. Description logics are based on the same ideas as semantic networks and frames but provide these with exact semantics. DLs have already proven their use in knowledge representation and reasoning. A DL consists of a description language, a knowledge specification language and automatic reasoning procedures. The most important aspect is these automatic reasoning procedures. They allow to reason about the consistency of knowledge bases. The question remains if state-of-the-art DLs are expressive and powerful enough to express and reason about different diagram types, constraints and the coupling of the different diagram types. In this paper, we show the translation of UML state diagrams and constraints written in the Object Constraint Language (OCL) [2] into the DL $\mathcal{DLR}$. In [4], class diagrams are translated into this DL. For the syntax and semantics of $\mathcal{DLR}$ we refer to [4]. The different translations will enable us to couple the class diagrams, corresponding state diagrams and OCL constraints in a model. We call this the horizontal view, it consists of the coupling of different UML diagram types in one model.

In section 2 we show how DL can be used to reason about UML state diagrams. The translation of OCL constraints into $\mathcal{DLR}$ is the topic of section 3. Section 4 concludes this paper.

2 UML State Diagram

An important diagram to represent the behaviour of an object-oriented application is the state diagram. State diagrams are finite state machines describing all the possible states that a particular object can get into and how the object's state changes as a result of events that reach the object.

As an example consider the class diagram on the left side of figure 1 which consists of six classes `Node`, `Access Control Node`, `RoadSegment`, `Trajectory`, `Customer` and `Card` and six associations. These associations are binary as most associations on UML class diagrams and are composed of two association-ends or roles. Some of these roles have a name, if they do not have a name, by default the names of the target classes are used. The multiplicities on the association-ends indicate the number of associated objects, e.g. each roadsegment has exactly one begin node, but this node can belong to several (zero or more) road segments. On the right side of figure 1, the state diagram of the class `Access Control`

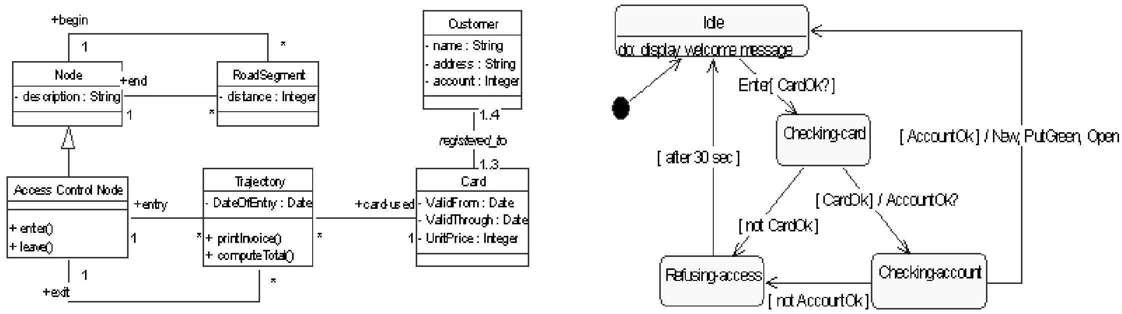

Figure 1: Class Diagram and State Diagram

`Node` is shown.

2.1 Mapping of the Basic Notations of State Diagrams

In this section the mapping of the basic notations of which a state diagram consists, into $\mathcal{DLR}$ is specified.

- **start state** This is a special state indicated by a black blob. This state has no name and it has an outgoing transition to the first state in which a newly created object starts off. This state is translated into $\mathcal{DLR}$ as a primitive concept. The name of this concept consists of the string "start" together with the name of the class to which the diagram belongs. State diagrams are drawn for a single class to show the lifetime behaviour of a single object. To specify this the following assertions are added:

$$\begin{aligned}
\texttt{startControlAccessNode} &\sqsubseteq \forall[1](\texttt{classrelation} \Rightarrow (2:\texttt{ControlAccessNode}))\\
\texttt{startControlAccessNode} &\sqsubseteq \exists[1]\texttt{classrelation} \sqcap (\leq 1[1]\texttt{classrelation})
\end{aligned}$$

- **states** Nodes in the state diagram show states labeled with a name. These states are mapped onto primitive concepts in $\mathcal{DLR}$.

- **transitions** The transitions between the different states have the following syntax in UML: *Event[Guard]/Action*. A guard is a logical condition and is translated as a logical expression of concepts. An event is also mapped into a concept. The actions are modeled as part of a transition, which is a relation between two states and the arity of this relation equals the number of actions plus one. Internal transitions are modeled in the same way. An example of an internal transition is `do: display welcome message` in the `Idle` state, this specifies that as long as we are in the `Idle` state a welcome message is displayed. Consider the transitions between the `Checking` and the `Refusing-access` state and the `Idle` state and the internal transition of the `Checking` state, these can be expressed as follows:

$$
\begin{aligned}
\mathtt{Checking} \sqcap \neg\mathtt{Ok} &\sqsubseteq \mathtt{Refusing\text{-}access} \\
\mathtt{Checking} \sqcap \mathtt{Ok} &\sqsubseteq \exists[1](\mathtt{transtoIdle} \Rightarrow (2\colon \mathtt{Idle}) \sqcap (3\colon \mathtt{New}) \sqcap \\
&\quad (4\colon \mathtt{PutGreen}) \sqcap (5\colon \mathtt{Open})) \\
\mathtt{Checking} \sqcap \mathtt{do} &\sqsubseteq \exists[1](\mathtt{transtoChecking} \Rightarrow (2\colon \mathtt{Checking}) \sqcap \\
&\quad (3\colon \mathtt{checks}) \sqcap (4\colon \mathtt{Display\_wait\_message}))
\end{aligned}
$$

- **Superstates** These can be introduced in state diagrams to group a number of states. Transitions can be defined starting from this state. A superstate can be modeled in $\mathcal{DLR}$ by a concept definition. For example, the states `Checking-card`, `Checking-account`, `Refusing-access` can be grouped into one state called `Active` and this can be defined as follows: $\mathtt{Active} \doteq \mathtt{Checking\text{-}card} \sqcap \mathtt{Checking\text{-}account} \sqcap \mathtt{Refusing\text{-}access}$ where $A \doteq B$ stands for $A \sqsubseteq B$ and $B \sqsubseteq A$.

2.2 Reasoning with State Diagrams

This translation of UML state diagrams provides the ability to reason about these diagrams. Some examples of reasoning tasks that become possible are:

1. A possible reasoning task is to check if only one transition can be taken out of a given state. This can be checked by verifying the disjointness of the combination of a state, guard and event for a given state. For example, starting from the `Checking` state two transitions can be taken, namely to the `Refusing-access` state and to the `Idle` state. In this case, it is possible to check the mutual exclusiveness of the guards because one guard is the primitive concept `Ok` and the other one the negation of this concept. As shown in the following section, mutual exclusiveness can also be checked if the guards consist of well-formed OCL constraints. However, if arbitrary expressions are allowed it is in general not possible to check this.

2. Another reasoning task is to verify if a particular state is reachable. The simplest form of reachability of a state is to check if starting from the start state it can be reached by following the transitions. To do this, we start from the start state, look at the direct superconcept which will give the next state. From this state we go on along possibly different paths untill the particular state is reached through one of these paths. In a more complex form reachability consists also of

verifying if the several events and actions specified on the different transitions exist. To do this, information from the class diagram is necessary. Remark that events and actions have global scope in UML. The most complex form of reachability is to verify if the events can be thrown by the objects in a certain configuration. To do this, the information found in state and class diagrams is not enough. We need to know how the application is initialized and which instances are alive. UML provides instance diagrams, but these diagrams are not yet translated into a DL.

3. A third possible reasoning task is to verify if the state diagram is deadlock free. This means that the guards on the outgoing transitions of one state must be complete.

3 OCL Expressions

The visual representation elements composing a UML diagram, such as a class diagram, are typically not enough to express all the constraints and specifications the system has to fulfill. UML modelers can use the OCL to specify application-specific constraints in their models. OCL is part of the UML specification [1]. For the grammar of OCL expressions we refer to [2]. The translation from OCL to $\mathcal{DLR}$ is currently subject of study in our lab. In this paper, we give an overview of the decisions we took and the difficulties we encountered even with quite simple OCL constraints.

In OCL, each object has a certain type. Types are divided in two groups: predefined types which include basic types such as `Integer` and `String` and collection types, i.e. `Set`, `Bag` and `Sequence`, and user-defined model types defined by the UML model.

3.1 Translation of Basic OCL Types

A first problem is the mapping of the basic types. These types are `Integer`, `String`, `Boolean` and `Real`. These primitive datatypes can be represented by concrete datatypes and a mechanism can be provided for deriving new datatypes as done in [6]. Consider the OCL invariant:

```
context Customer inv:
age >= 18
```

which states that a customer must be 18 years or older. We could for example define the new type (min 18). In this case, the constraint could be asserted as follows in the formalization of the attribute `age`: $\texttt{Customer} \sqsubseteq \forall[1](\texttt{age} \Rightarrow (2 : (\text{min } 18))$. $\mathcal{DLR}$ extended with concrete datatypes in the sense of [6] does not give any problem. However, these concrete domains are unary.

Another way to extend $\mathcal{DLR}$ is with concrete domains as is done in [3]. Recall that in that context a concrete domain is a pair (Δ_D, Φ_D) with Δ_D is a set and Φ_D is a set of predicate names. The OCL basic types can be viewed as concrete domains. The `Integer` and `Real` types could be taken together. This allows us to compare for example an integer number with a real number. The predicates for this new domain are: $\{\leq, \geq, \neq, =, <, >\}$. However, nothing is known about decidability of $\mathcal{DLR}$ extended with concrete domains.

3.2 Translation of Navigation through the Class Diagrams

A second difficulty concerns the OCL dot-operator (.). With this operator it is possible to navigate through the class diagrams. Consider for example the constraint that if the distance of a roadsegment is greater than 120, this implies that the description of the begin node of the roadsegment must be "Orleans". In OCL, this could be written down as follows:

```
context RoadSegment inv:
self.distance > 120 implies self.begin.description = "Orleans"
```

Consider this OCL expression, the `RoadSegment` class is the starting point of the navigation, because the multiplicity of the role `begin` is one, the result of `self.begin` is an object of type `Node`. The second use of the dot in `self.begin .description` is to access the `description` property of `Node`. If we write the constraint down starting from the `Node` class, the `self.roadsegment` navigation results in a set of objects of type `RoadSegment`. In the first case, it would be natural to write down the constraint on the `RoadSegment` concept and we could use feature-chain agreement [3], however $\mathcal{DLR}$ extended with feature-chain agreement is undecidable. The second case is even worse because we encounter the problem of multi-valued roles and this causes undecidability of subsumption [5]. However, the first constraint can be written down as follows:

$$\texttt{RoadSegment} \sqcap \exists[1](\texttt{distance} \sqcap (2{:}(>_{120}\texttt{distance}))) \sqsubseteq \forall[2](\texttt{roadsegment} \Rightarrow (1{:}(\texttt{begins\_in} \Rightarrow \forall[1](\texttt{begin} \Rightarrow (2{:}(\texttt{Node} \Rightarrow \forall[1](\texttt{description} \Rightarrow (2{:}\ (=_{\text{"}Orleans\text{"}}\ \texttt{description})))))))))$$

This translation is an example of semantic linking. We can verify the existence of the navigation path used in the OCL constraint w.r.t. the class diagram.

3.3 Collection types

In this section, we only consider the `Set` type. This is the most common type in OCL constraints and in many OCL constraints, expressions of type `Bag` and `Sequence` are actually used as sets. For example, consider the constraint:

```
context Card inv:
self.customer->forAll(c |c.age >= 21)
```

The navigation from `Card` to `Customer` results into a set. However even if it would be a bag or a sequence it would not matter, because the order of the elements and the number of their occurrences is irrelevant. The translation of this constraint into $\mathcal{DLR}$ becomes:

$$\begin{aligned}\texttt{Card} \sqsubseteq\; & \forall[2](\texttt{card} \Rightarrow (1{:}(\texttt{registered\_to} \Rightarrow \forall[1](\texttt{customer} \Rightarrow (2{:}(\texttt{Customer} \\ & \Rightarrow \forall[1](\texttt{age} \Rightarrow (2{:}(\geq_{21}\ \texttt{age})))))))))\end{aligned}$$

The features `includes, includeAll, isEmpty, exists` and `select` can be translated in an analogue way.

3.4 Reasoning with OCL Expressions

The translations in the previous sections do not cover all the OCL constraints, but the most important ones are captured. The advantages of a translation of OCL expressions into a DL are that the software modeler does not have to know anything about DL's, but only the OCL which is more and more supported by case-tools. Using the reasoning tasks of the DL, some automated reasoning support on OCL expressions can be provided to the designer. Examples of reasoning tasks that become possible are:

1. The *constraints* can be checked on *consistency with respect to the class diagram.* A constraint is consistent w.r.t. the class diagram if it can be satisfied without contradicting the conditions imposed by the classes. E.g. the navigation paths could be checked as explained in section 3.2.
2. The *set of constraints* defined on one model can be checked on internal *consistency.*
3. Furthermore, it would be possible to check *constraint equivalence*, this boils down to equivalence of logical formula's.
4. Finally, the OCL constraints used in guards on state diagrams can be checked on mutual exclusiveness and completeness.

4 Discussions and Conclusions

The previous translations of UML state diagrams and OCL constraints to $\mathcal{DLR}$ are part of the horizontal view. The use of the same underlying formalism enables the coupling of the different types of UML diagrams within a model. However, we also want to use DL's for the vertical view on object-oriented design. This view consists of the linking of the several models in the SDLC. As the development of the software application progresses, these different types of UML diagrams get refined and information is added or deleted resulting in new more detailed diagrams which are part of a model on a less abstract level. This is especially the case in the design phase, because the conceptual models are refined and adapted to take the programming language and platform into account. The support for the linking of these models is currently lacking in state-of-the-art case-tools. It is our goal also to use DL for this purpose. To support the linking of these models, the changes between the different diagrams in the different models should be kept. We think in the direction of a library of change operations. Change operations on UML class diagrams include: add/delete class, association, aggregation, inheritance, attribute and method. These operations correspond to the following change operations in $\mathcal{DLR}$: add/delete concept, binary relation, and subsumption relationships. The ideas specified here are preliminary and should be further explored.

5 Acknowledgements

We would like to thank Diego Calvanese for his explanation of $\mathcal{DLR}$ and his ideas on the translation of OCL expressions to $\mathcal{DLR}$ and Maja D'Hondt, Viviane Jonckers and Miro Casanova for editing the manuscript.

References

[1] The OMG Unified Modeling Language Specification. The Object Management Group http://www.omg.org.

[2] Kleppe A. and Warmer J. *The Object Constraint Language: Precise Modeling with UML.* Addison Wesley, 1999.

[3] F. Baader and P. Hanschke. A Scheme for Integrating Concrete Domains into Concept Languages. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence, IJCAI-91*, pages 452–457, Sydney (Australia), 1991.

[4] Andrea Calí, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Reasoning on UML Class Diagrams in Description Logics. In *Proc. of IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development (PMD 2001)*, 2001.

[5] B. Hollunder and W. Nutt. Subsumption Algorithms for Concept Languages. Technical Report RR-90-04, DFKI, Kaiserslautern, 1990.

[6] I. Horrocks and U. Sattler. Ontology Reasoning in the SHOQ(D) Description Logic. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2001.