Towards the Harmonisation of UML and SDL - Syntactic and Semantic Alignment -

R. Grammes, R. Gotzhein

Technical Report 327/03
Technical University of Kaiserslautern

Towards the Harmonisation of UML and SDL - Syntactic and Semantic Alignment -

Rüdiger Grammes, Reinhard Gotzhein
Technical University of Kaiserslautern
`{grammes,gotzhein}informatik.uni-kl.de`

Technical Report 327/03

Computer Science Department
Technical University of Kaiserslautern
Postfach 3049
67653 Kaiserslautern
Germany

Towards the Harmonisation of UML and SDL - Syntactic and Semantic Alignment -

Rüdiger Grammes, Reinhard Gotzhein

Department of Computer Science
University of Kaiserslautern
67653 Kaiserslautern, Germany
{grammes,gotzhein}@informatik.uni-kl.de

Abstract. UML and SDL are languages for the development of software systems that have different origins, and have evolved separately for many years. Recently, it can be observed that OMG and ITU, the standardisation bodies responsible for UML and SDL, respectively, are making efforts to harmonise these languages. So far, harmonisation takes place mainly on a conceptual level, by extending and aligning the set of language concepts.

In this paper, we argue that harmonisation of languages can be approached both from a syntactic and semantic perspective. We show how a common syntactical basis can be derived from the analysis of the UML meta-model and the SDL abstract grammar. For this purpose, conceptually sound and well-founded mappings from meta-models to abstract grammars and vice versa are defined and applied. On the semantic level, a comparison between corresponding language constructs is performed.

1 Introduction

UML (Unified Modeling Language [1], [2]) is a graphical language for specifying, modelling and documenting software systems with widespread use in industry, standardised by the Object Management Group (OMG). It is a family of notations (e.g., use case diagrams, class diagrams, sequence diagrams, statechart diagrams, deployment diagrams) supporting different views of a system throughout the software life cycle. Recently, the UML 2.0 standard was finalised. The new standard is a major revision of UML 1.x, and introduces, amongst other things, better support for system structure and components.

SDL (System Design Languages [3]) is a graphical specification language for distributed systems and, in particular, communication systems, standardised by the International Telecommunications Union (ITU). It is widely used in telecommunications industry. SDL is a sophisticated set of notations (e.g., MSC-2000, SDL-2000, ASN.1, TTCN), supporting different system views on different levels of abstraction.

With SDL-2000, several important steps towards its future harmonisation with UML were made. For instance, classes and associations including aggregation, composition, and specialisation were added to the language. Furthermore, composite states that are similar to submachines in UML statecharts were incorporated. In turn, UML 2.0 introduced structured classes, which extend classes by an internal structure consisting of nested structured classes, ports and connectors. This makes it possible to model architectural aspects of systems in a fashion similar to SDL.

First attempts to harmonise UML and SDL have already been made. The Z.109 standard [4] defines a subset of UML 1.3 [2] that has a mapping to SDL-2000. The UML subset is used in combination with SDL, with the semantics based on SDL-2000. In [5], Selic and Rumbaugh define a transformation from SDL-92 to UML 1.3 extended with the Rational Rose real-time profile.

Ultimately, these efforts are directed towards an integration of both languages and the corresponding notations. However, at the time being, UML and SDL still deviate in many ways, making it hard to see whether and when integration might actually be achieved. Differences range from pure syntactic aspects to semantic concepts, resulting from the origin of the languages. Also, it is not clear whether different views of a system even if expressed in notations belonging to the same family are consistent.

A true integration of both languages and the corresponding notations will require a common syntactic and semantic basis. This basis may then be extended in diverging ways, yielding a variety of language profiles. This way, the system developer will be enabled to model different parts of a system using different notations, and to combine them into a single view.

In order to derive a common syntactic and semantic basis, the existing language definitions of UML and SDL should be taken as a starting point. In this paper, we present the results of analysing several corresponding excerpts of UML and SDL, compare them, and derive a common subset. On the syntactical level, this is done by defining conceptually sound and well-founded mappings from meta-models (used to define the abstract syntax of UML, see Section 2) to abstract grammars (used by SDL, see Section 2) and vice versa (Section 3), and by extracting common production rules (Section 4). Results of a case study are shown in the Appendix. On the semantic level, an informal comparison between corresponding language constructs is performed (Section 5). Results are discussed in Section 6.

2 UML Meta-Model and SDL Abstract Grammar

The definition of a language consists of its syntax and semantics. The concrete syntax of a language includes separators and other constructs needed for parsing the language. The abstract syntax omits these details and contains only the elements relevant for the definition of the semantics. Both the concrete and the abstract syntax of a language can be defined in terms of a grammar, consisting of a set of production rules that define the syntactically correct sentences.

For SDL, a concrete (textual and graphical) syntax and two abstract syntaxes, AS0 and AS1, are defined. The AS0 is obtained from the concrete syntax by omitting details such as separators and lexical rules. Otherwise, is very similar to the concrete syntax of SDL. The abstract syntax AS1 is obtained from the abstract syntax AS0 through a step of transformations followed by a mapping. During the transformation, additional concepts are translated into core concepts of SDL as described in the standard.

The abstract syntax of SDL is described in terms of an abstract grammar, similar to BNF. It consists of two kinds of production rules, namely concatenations and synonyms. A *concatenation* 'lhs ::=(::) rhs' describes the non-terminal lhs (left hand side) as a composite object consisting of the elements denoted by rhs (right hand side). Optional elements are enclosed in square brackets, and alternatives are separated by vertical bars. The suffix '*' describes a possibly empty list of elements, '+' a non-empty list and '**-set**' a set of distinguishable elements. A *synonym* 'lhs ::=(=) rhs' describes that the non-terminal lhs is an element of the kind rhs and can not be syntactically distinguished from other elements of this kind.

For the mapping described in Section 3, we assume a normal form of the abstract grammar, where concatenations have no alternatives on the right hand side. The SDL abstract grammar can be easily transformed into this normal form by introducing new synonyms for these alternatives.

Below an excerpt of the AS1, the production rule State-node, is shown. State nodes are composite objects consisting of a state name, a save signalset, and sets of input nodes, spontaneous transitions, continuous signals and connect nodes. Optionally, a state node can have a composite state type identifier (in that case, the state represents a composite state of the respective type).

```
State-node ::=(::)    State-name
                      [On-exception]
                      Save-signalset
                      Input-node-set
                      Spontaneous-transition-set
                      Continuous-signal-set
                      Connect-node-set
                      [Composite-state-type-identifier]
```

A meta-model is a model used to define a language for the specification of models. In UML, this meta-model approach is used to define the language syntax [...]. In particular, the abstract syntax of the language is defined using UML class diagrams. This approach is reflective, since class diagrams are UML models, and therefore described in terms of themselves. On top of the model and the meta-model, more layers can exist (meta-meta-models, etc.). UML uses a four layer meta-model structure: user objects (M0), model (M1), meta-model (M2) and meta-meta-model (M3). Every element in a layer is an instance of an element of the direct superordinate layer.

The UML class diagrams used for the description of the abstract syntax comprise packages, classes, attributes, associations and specialisation. Classes in the UML meta-model describe language elements. An occurrence of the language element in the model (M1) is an instance of the meta-model class. Classes in the meta-model can be parameterised by attributes. Attributes describe properties of the language element described by a class. Composition between meta-model classes describes that a language element contains another. General associations relate language elements with each other, e.g., a transition with a trigger. The meta-model uses packages, abstract classes and specialisation to structure the abstract syntax.

Fig. 1 shows an excerpt from the abstract syntax of statemachine states. States are described by a class with four attributes, describing the type of the state. These attributes are derived ('/'), meaning that their value is derived from other information in the meta-model. A State contains up to one Activity in the role of a 'doActivity'. It is also associated with an arbitrary number of triggers in the role of deferrable triggers. The composition between State and Activity is a subset of the association 'ownedElement'.

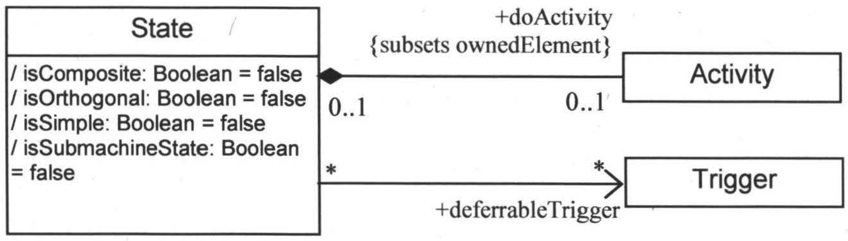

State
/ isComposite: Boolean = false
/ isOrthogonal: Boolean = false
/ isSimple: Boolean = false
/ isSubmachineState: Boolean = false
+doActivity
{subsets ownedElement}
0..1
0..1
Activity
*
*
Trigger
+deferrableTrigger

Fig. 1: Excerpt from the abstract syntax of states

3 Defining Mappings between UML Meta-Model and SDL Abstract Grammar

In this section, we define precise, conceptually sound mappings from meta-models to abstract grammars, and vice versa. As it turns out, not every element of the UML meta-model can be mapped. Also, several meta-model elements may have the same representation in the abstract grammar. Therefore, the mapping is not completely reversible. However, it is possible to map every element of an abstract grammar to a meta-model representation. In Section 4, these mappings will be applied to UML and SDL to extract a common syntactical basis.

3.1 Classes and Enumerations

map(MM): A *concrete class* of the meta-model represents a language element of the model. E.g., the meta-model class State represents all state descriptions in a UML statemachine. In an abstract grammar, a language element is represented by a specific production rule, namely a concatenation. Therefore, a concrete class in the meta-model is mapped to a concatenation of the abstract grammar. The name of the non-terminal is derived from the class name and the package structure of the meta-model (see below). The right hand side of the concatenation is derived from the class definition (attributes) and context (associations) as defined below (see 3.2, 3.3).

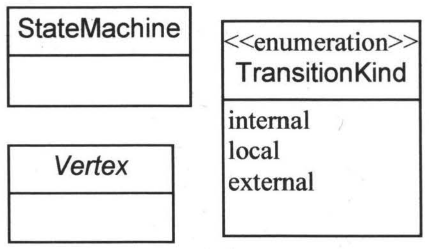
An *abstract class* of the meta-model describes properties that are common to its subclasses. E.g., the meta-model class *Vertex* describes properties that are common to states and pseudo-states (initial states, ...). Since an abstract class can not be instantiated, it does not represent a language element of the model. Therefore, no concatenation is used in the mapping. Instead, we have decided to map an abstract class of the meta-model to another kind of production rule, namely a synonym, of the abstract grammar. In an abstract grammar, a synonym replaces the element on its left hand side with an element of the right hand side. This is a similar to abstract classes in the meta-model, which must be replaced by one of their concrete subclasses in a model. The name of the non-terminal is selected as in the case of a concrete class. The right hand side is derived from the context (specialisation) as described below (see 3.5).

An *enumeration* in the meta-model is a set of values used to parameterise meta-model classes. E.g., the meta-model class Pseudostate describes different language elements (entry point, exit point, ...) of the model depending on the value of the attribute 'kind' of the enumeration type PseudostateKind. Enumerations do not directly describe language elements of the model. Therefore, as in the case of abstract classes, no concatenation is used in the mapping. Instead, enumerations are also mapped to synonyms of the abstract grammar. This production rule replaces the enumeration by one of its values.

The name of the non-terminals introduced by the mappings described above is the qualified name of the class or enumeration. The qualified name is a sequence of the packages the class or enumeration is contained in (from outermost to innermost) and the name of the class or enumeration, each separated by underscores. E.g., Kernel_Element is the name of the non-terminal introduced by the class Element in the package Kernel. The qualified name is used in order to avoid name clashes between equally named classes in different packages.

Example: The following example comes from the meta-model of UML state machines. It describes two classes, an abstract class *Vertex* and a concrete class StateMachine. Furthermore, there is an enumeration TransitionKind. All of these elements are contained in the package BehaviorStatemachines (not shown), that we will shortly refer to as BehSM.

StateMachine is a concrete class, and is therefore mapped to a concatenation. The name BehSM_StateMachine comes from the package structure and the name of the class. The abstract class *Vertex* and the enumeration TransitionKind are mapped to synonyms.

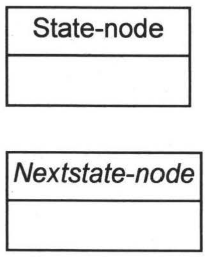
MM	map(MM)
 StateMachine <<enumeration>> TransitionKind internal local external Vertex	BehSM_StateMachine ::=(::) BehSM_Vertex ::=(=) BehSM_TransitionKind ::=(=)

map(AG): As mentioned in the mapping from meta-models to abstract grammars, concrete classes and *concatenations* both represent language elements of the model. Therefore, concatenations of the abstract grammar are mapped to concrete classes in the meta-model. The name of the concrete class is derived from the production rule (see below).

A *synonym* of the abstract grammar represents a language element that does not appear in the model, but stands for other language elements. E.g., a Data-type-definition in the SDL abstract grammar is a synonym for a Value-data-type-definition, an Object-data-type-definition or an Interface-definition. This is a similar concept to abstract classes in the meta-model, which we have mapped to synonyms in the abstract grammar. However, it is also similar to an enumeration, where the enumeration stands for one of its values. Therefore, we map a synonym in the abstract grammar either to an abstract class or an enumeration. The exact mapping depends on the right hand side of the synonym (see 3.2, 3.3).

The name of the class or enumeration is the name of the non-terminal on the left hand side of the production rule.

Example: In the abstract grammar of SDL, the production rule for the non-terminal State-node is a concatenation. It is therefore mapped to a concrete class. The production rule for Nextstate-node is a synonym. It would therefore either map to an abstract class or an enumeration. Because of the right hand side of the production rule, which we do not treat at this point, it is mapped to an abstract class.

AS1	map(AS1)
State-node ::=(::) … Nextstate-node ::=(=) …	

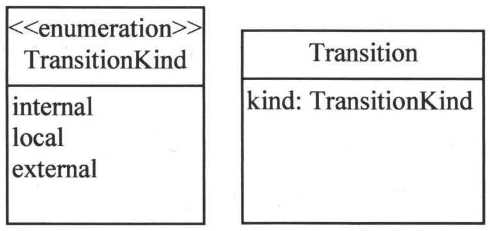
3.2 Attributes

map(MM): In the meta-model, *attributes* of a class represent properties of a language element of the model. E.g., the attribute 'kind' of the meta-model class Transition describes if the transition is internal, local or external. In an abstract syntax tree, an attribute is represented as a sub-node of the non-terminal and corresponds to a class. We have mapped concrete classes to concatenations of the abstract grammar. Therefore, an attribute of a concrete class is mapped to a terminal on the right hand side of the concatenation. We map attributes to terminals since they do not need to be refined any further. The only exception is an enumeration type; in that case we map the attribute to a non-terminal, since we have mapped enumerations to synonyms and non-terminals. The name of the terminal is the name of the type (e.g. Boolean). The name of the non-terminal is derived from the name of the enumeration and the package structure, as defined in 3.1.

Attributes that are marked as *derived* carry no additional information and can be omitted. E.g., the attribute 'isComposite' of State can be derived from the number of associated regions. If they are not omitted, additional static conditions are needed to define the dependencies between the original and the derived attributes. Default values of attributes can not be mapped to the abstract grammar. They can be described by static conditions.

Elements of an enumeration represent values of the enumeration type. A value in an abstract grammar is represented by a terminal. Therefore, enumeration elements are mapped to terminals of the abstract grammar. The name of the terminal is the name of the enumeration element. An enumeration is mapped to a synonym of the abstract grammar. Therefore, we map the terminals to the right hand side of the synonym corresponding to the enumeration.

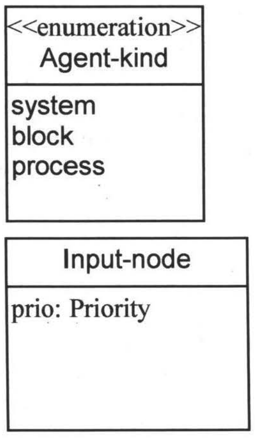
Example: The following example (again from BehaviorStatemachines) contains a concrete class Transition and the enumeration TransitionKind. The classes are mapped as described in the previous section. The attribute 'kind' of Transition is an element on the right hand side of BehSM_Transition. In this special case ('kind' is an enumeration), it is a non-terminal that refers to the mapping of the enumeration TransitionKind. The name of the attribute is appended as a comment. The enumeration literals of TransitionKind appear as an alternative of terminals on the right hand side of the production rule, written in all caps for better distinction.

MM	map(MM)
	BehSM_TransitionKind ::=(=) INTERNAL | LOCAL | EXTERNAL BehSM_Transition ::=(::) BehSM_TransitionKind /* kind */

map(AG): A terminal on the right hand side of a concatenation represents a property of the language element. In the meta-model, an attribute represents a property of a language element. The terminal is therefore mapped to an attribute of the concrete class corresponding to the concatenation. The type of the attribute is the name of the terminal. The name of the terminal can be chosen arbitrarily as long as it does not conflict with other attribute names of the class.

A synonym with only terminals on the right hand side represents an enumeration of values. E.g., the synonym Agent-kind of the SDL abstract grammar is an enumeration of the values SYSTEM, BLOCK and PROCESS. Therefore, the terminals are mapped to enumeration values of the enumeration corresponding to the synonym.

Example: The terminal PRIORITY on the right hand side of Input-node is mapped back to an attribute of the corresponding concrete class. Agent-kind is a synonym and has only terminals on the right hand side. Therefore, it is mapped to an enumeration. The alternatives on the right hand side (SYSTEM, BLOCK and PROCESS) are mapped to enumeration literals of the corresponding enumeration.

AS1	map(AS1)
Agent-kind ::=(=) SYSTEM | BLOCK | PROCESS Input-node ::=(::) PRIORITY ...	

3.3 Associations

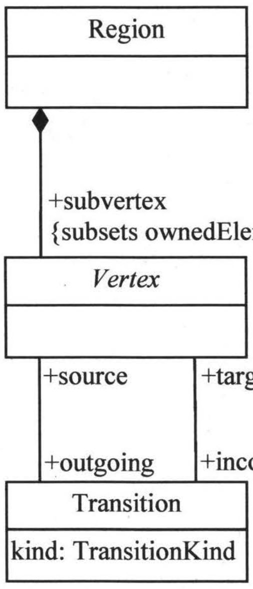
map(MM): An *aggregation* or *composition* between two classes means that one language element contains or is made up of other language elements. E.g., a Region in a statechart contains vertices and transitions. In the same way, a node in an abstract syntax tree can have sub-nodes. E.g., a State-transition-graph of the SDL abstract grammar has a set of State-nodes as sub-nodes. Therefore, we map aggregation and composition to the abstract grammar so that the definition of the aggregated class is a sub-node of the aggregating class. This is achieved by adding the non-terminal corresponding to the aggregated class on the right hand side of the concatenation corresponding to the aggregating concrete class.

A *general association* between two classes is an association between language elements, in which the elements play a certain role. E.g., a State is associated with a number of triggers, the triggers playing the role of deferrable triggers. In the SDL abstract grammar, two language elements are associated by identifiers. E.g., an Input-node is associated with a Signal by a Signal-identifier on the right hand side of the concatenation corresponding to the Input-node. Therefore, a directed general association is mapped to an identifier on the right hand side of the concatenation corresponding to the concrete class the association originates from. An undirected general association is split into two directed general associations.

An associations with the *union* property is the union of the associations that subset it. This is expressed by the property *subsets*. As in the case of derived attributes, associations with the union property are not mapped to the abstract grammar.

Example: The following example shows the abstract class *Vertex* and the concrete classes Transition and Region. Region is composed of a *Vertex* called subvertex. This composition is a subset of the association 'ownedElement' between two Elements. In the AST, BehSM_Region thus has BehSM_Vertex on the right hand side, with the name appended as a comment. Between *Vertex* and Transition there are two bidirectional associations, which are split into two unidirectional associations respectively. Attributes and associations of an abstract class are not mapped to the corresponding synonym in the AST, since an abstract class is not a synonym for one of its attributes or associations. Instead, they are copied into the respective subclasses, as described in Section 3.5. In this example, Vertex has no subclasses. Therefore, we only have to map the two general associations 'source' and 'target'. To distinguish between general association and composition, an association is mapped to an identifier (in this case, BehSM_Vertex-Identifier) on the right hand side of the corresponding production rule. How the identifier looks like is not further specified. It could be a qualified name like in the case of SDL.

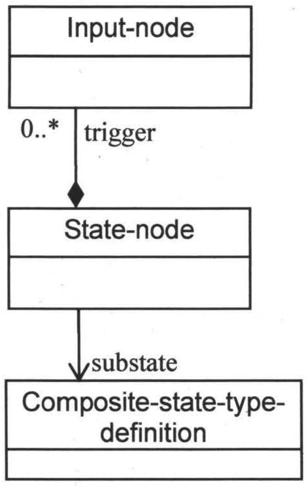
MM	map(MM)

 Region
+subvertex
{subsets ownedElement}
Vertex
+source
+target
+outgoing
+incoming
Transition
kind: TransitionKind
	BehSM_Region ::=(::)
	    BehSM_Vertex /* subvertex */
	
	BehSM_Vertex ::=(=)
	
	BehSM_Transition ::=(::)
	    BehSM_TransitionKind /* kind */
	    BehSM_Vertex-Identifier
	        /* source */
	    BehSM_Vertex-Identifier
	        /* target */
	
	BehSM_Vertex-Identifier ::=(=)
	    Identifier

map(AG): *Non-terminals* on the right hand side of a *concatenation* can stand for an enumeration or a class in the meta-model. In case they represent an enumeration they represent an attribute of the class (see 3.2). In case they represent a class, this class is a sub-node of the class corresponding to the concatenation. This is similar to a class in the meta-model that is composed of other classes. Therefore, in this case we map a non-terminal on the right hand side of a concatenation to a composition in the meta-model. The composing class is the class corresponding to the concatenation; the composed class is the class corresponding to the non-terminal on the right hand side. The role of the classes can be chosen arbitrarily.

An *identifier* on the right hand side of a *concatenation* identifies a language element that is associated with the language element described by the concatenation. E.g., in the SDL abstract grammar, an Input-node is associated with a Signal by a Signal-identifier. Therefore, we map an identifier on the right hand side of a concatenation to a directed general association in the meta-model. The source of the association is the concrete class corresponding to the concatenation, according to the mapping in 3.1. The target is the concrete class corresponding to the language element referenced by the identifier. The role of the classes can be chosen arbitrarily.

Example: The non-terminal Input-node on the right hand side of State-node is mapped to a composition of Input-node in State-node in the meta-model. The set-suffix is mapped to the multiplicity '0..*', as defined in 3.4. Composite-state-type-identifier is an identifier referring to a Composite-state-type-definition (in the abstract grammar of SDL, identifier/definition pairs usually have the same name with a -identifier/-definition suffix, e.g. Signal-identifier and Signal-definition). This is mapped to a general association between the two corresponding classes in the meta-model.

AS1	map(AS1)
State-node ::=(::) Input-node-**set** Composite-state-type-identifier ... Input-node ::=(::) Composite-state-type-definition ::=(::) Composite-state-type-identifier ::=(=) Identifier	 Input-node 0..* trigger State-node substate Composite-state-type-definition

3.4 Multiplicity

The following table defines a mapping between multiplicities in the meta-model and the abstract grammar. In UML, multiplicities consist of a lower bound and an optional upper bound, which can be infinite. The property *ordered* expresses that there is a linear order for the elements. The property *unique* expresses that no element appears more than once. In the abstract grammar, an optional element is enclosed by square brackets. A possibly empty list of elements is marked by a '*' behind the element, a non empty list by a '+'. A set of distinct elements is marked by the suffix '-set'.

MM	**AG**
0..1	*[Name]*
0..n, 1 < n	(as 0..*)
*0..**	*Name **
0.. {unique}*	*Name –**set***
1	*Name*
1..n, 1 < n	(as 1..*)
*1..**	*Name +*
1..* {unique}	(as 0..* {unique})
n	(as 0..*)
n..m, 1 < n < m	(as 1..*)
n..*, 1 < n	(as 1..*)

Table 1: Mapping of Multiplicities

Table 1 shows the mapping of multiplicities between meta-model and abstract grammar. If we use lists in the abstract grammar, the elements are ordered and not necessarily unique. If we use sets, they are not ordered and unique. Therefore, we can only map one of the properties to the abstract grammar. In this case, the property *ordered* is omitted from the mapping.

3.5 Specialisation

In the UML meta-model, abstract classes and specialisation are used frequently to capture common aspects of different classes, and as part of a meta-language core reused in several standards (see UML: Infrastructure [1]). For the abstract syntax, abstract classes are not directly interesting, since they can not be instantiated and therefore do not appear in a model, except through their subclasses. Nonetheless we map them to the abstract grammar, to preserve as much of the structure of the meta-model as possible.

map(MM): We have to map specialisation to the abstract grammar, and the fact that a specializing class inherits properties of the specialised class. The easiest way to do this is to copy these properties into the specializing classes before the mapping. This has the advantage that redefinition of properties is easy to realise. They are not copied to subclasses that overwrite them.

This is done as follows:

Step 1. For every class that has subclasses, copy all attributes of the class and all associations that originate from this class to each of its direct subclasses.

Step 1.1. An attribute is only copied to a subclass if no attribute of the same name already exists, i.e., if the attributed is not redefined.

Step 1.2. An association is only copied to a subclass if it is not redefined in the subclass.

Step 2. Repeat step *1* for all subclasses that have new attributes and associations after the last execution of step 1.

In the meta-model, an abstract class can take part in an association. In the model, an instance of a concrete class that specialises the abstract class takes part in the association instead. E.g., a Vertex is associated with Transitions as the source of these transitions. In the model, the source of these transitions is either a State or a Pseudostate. In the abstract grammar, we can express this using a synonym. We have already mapped an abstract class to a non-terminal and a synonym (see 3.1). To map the specialisation to the abstract grammar, we add the non-terminals corresponding to the direct sub-classes of the abstract class to the right hand side of the synonym. This means that every occurrence of the non-terminal (the abstract class) is replaced by a non-terminal (one of the subclasses) in the abstract syntax tree.

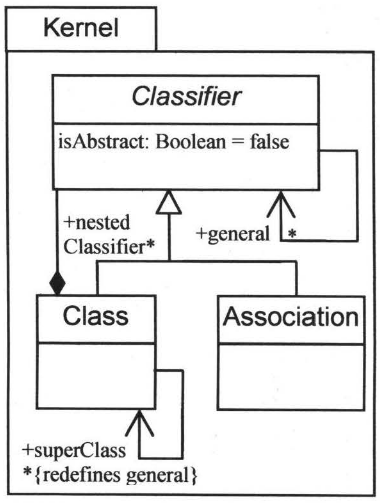
To map specialisation to the abstract grammar, we need synonyms. On the other hand, a concrete class can have subclasses, but is mapped to a concatenation (see 3.1). In this case, we transform the meta-model before we perform the mapping. The concrete class with subclasses is replaced by an abstract class of the same name. The concrete class is renamed, e.g. by adding a special prefix, and added as a subclass of the new abstract class. The subclasses of the concrete class are now subclasses of the new abstract class and the mapping can be performed. However, we still have to copy the attributes of the concrete class to its former subclasses, as described above.

Example: The following example is taken from the package Kernel and covers classifiers, classes and associations. *Classifier* is an abstract class with the attribute 'isAbstract'. A classifier can be generalised by another classifier, described by the association named 'general'. Concrete subclasses of *Classifier* are Class and Association. The association 'superClass' between two classes redefines the association 'general'.

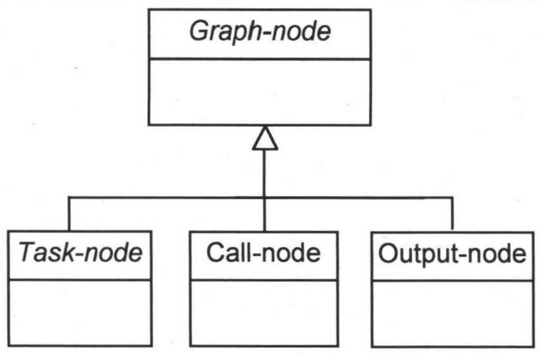
Before mapping to the abstract grammar, we have to copy the attributes and associations of the abstract class *Classifier* to its subclasses. The attribute 'isAbstract' is copied to the classes Class and Association, since no attribute of the same name exists. A new association 'general' from Association to *Classifier* is added. The association 'superClass' redefines 'general', therefore no new association is added to Class.

The abstract class *Classifier* is mapped to a synonym. Class and Association are direct subclasses of *Classifier*; therefore, we add the non-terminals corresponding to these classes on the right hand side of the synonym.

MM	map(MM)

 Kernel
Classifier
isAbstract: Boolean = false
+nested Classifier*
+general
*
Class
Association
+superClass
*{redefines general}
	Kernel_Classifier ::=(=)
	    Kernel_Class
	    | Kernel_Association
	
	Kernel_Class ::=(::)
	    Boolean /*isAbstract*/
	    Kernel_Class-Identifier-**set** /*superClass*/
	    Kernel_Classifier-**set** /* nestedClassifier */
	
	Kernel_Association ::=(::)
	    Boolean /*isAbstract*/
	    Kernel_Classifier-Identifier-**set** /*general*/
	
	Kernel_Class-Identifier ::=(=) Identifier
	
	Kernel_Classifier-Identifier ::=(=) Identifier

map(AG): *Non-terminals* of the abstract grammar represent language elements. A synonym of the abstract grammar with non-terminals on the right hand side replaces a language element by another. E.g., a Return-node in the abstract grammar of SDL is replaced by an Action-return-node, a Value-return-node or a Named-return-node. We map synonyms to abstract classes in the meta-model. Abstract classes can not be instantiated, but can have instances through their subclasses. Therefore, we map a synonym with non-terminals on the right hand side in the abstract grammar to a specialisation relationship. The specialised class is the class corresponding to the non-terminal on the left hand side. The specialising classes are the classes corresponding to the non-terminals on the right hand side.

AS1	map(AS1)
Graph-node ::=(=) Task-node | Output-node | Call-node | ... Task-node ::=(=) Output-node ::=(::) Call-node ::=(::)	

Example: The synonyms for Graph-node and Task-node are mapped to abstract classes. Output-node and Call-node are concatenations and therefore concrete classes. For the non-terminals Task-node, Output-node and Call-node on the right hand side of the synonym, a specialisation relationship is added between *Graph-node* and *Task-node*, Call-node and Output-node.

3.6 Meta-Model Approach vs. Abstract Grammar Approach

From the discussion so far, it is quite obvious that the meta-model approach to defining an abstract syntax is more expressive than the (context free) grammar approach. As a consequence, the mapping from the SDL abstract grammar to a meta-model is completely reversible. However, this is not the case for the mapping from the UML meta-model to an abstract. Several elements of the UML meta-model can not be expressed in the abstract grammar, including the following:

- Visibility information, default values and derived attributes are not expressible in a context free grammar. It is, however, possible to describe them with static conditions.
- Associations with the property *union*, as well as the property *subsets* are not expressible.
- When an attribute or an association of an abstract class is redefined in a subclass, it cannot be properly reproduced. Instead, it is copied into the subclasses. The property *redefined* is lost in the mapping.
- Several multiplicities have identical mappings, and cannot be properly reproduced. The property *ordered* (alternatively, the property *unique*, see Section 3.4) is lost in the mapping.

In consequence, the meta-model approach seems to be preferable as a basis for the harmonisation of UML and SDL. It covers and extends the expressiveness of abstract grammars, and thus seems to be the right choice. However, when it comes to implementing a language by providing tool support, an abstract grammar is still needed. With the mapping defined above, such an abstract grammar can be systematically derived.

4 Extracting a Common Abstract Syntax from SDL and UML

Translating the meta-model of UML 2.0 into an abstract grammar supports the comparison of the abstract syntax of UML 2.0 and SDL-2000. In particular, enables us to examine how the common constructs of SDL and UML are reflected in common parts of the abstract syntax of both languages, and to extract a common abstract grammar.

As it has turned out, some information of the meta-model is lost when it is mapped to an abstract grammar (see Section 3.6). However, the information lost is not important for the extraction, because it is not present in the abstract syntax of SDL.

Instead of mapping the UML meta-model to an abstract grammar, we could apply the mapping from the SDL abstract grammar to a meta-model. This way, no information would be lost, as the meta-model is more expressive. However, the extraction process would not benefit from this choice. Even worse, the extraction would be harder, since the UML meta-model defines a large number of abstract classes with attributes and associations, which would not show up in the SDL meta-model. It would be necessary to either copy the attributes of abstract classes to their subclasses in the UML meta-model (as described in Section 3.5), or to identify common attributes and associations, and shift them to super-classes in the SDL meta-model.

To relate language elements of SDL-2000 and UML 2.0 on a syntactical level, substantial knowledge of both languages is required. In particular, it is necessary to take the semantics of language elements into account. E.g., we need knowledge of the semantics of the language elements to relate the Package-name of a Package-definition in the abstract syntax of SDL with the String of a structured class in the abstract syntax of UML. Also, it can be expected that for some of the common constructs the abstract syntax will be different, although the semantic is the same. In some cases, there might even be a common abstract syntax, although the semantics is different.

To extract the common abstract syntax of the two languages, we take the production rules for language elements that are similar in UML and SDL, e.g. packages, as a starting point, and compare their right hand sides. For corresponding elements in both sets of production rules that represent similar concepts, the production rules for these elements are compared. If they overlap, we can relate the two elements with each other and include them in the common abstract syntax. We start the extraction with very high level language elements, namely packages and agents/classes, before moving to language elements with a finer granularity.

4.1 Packages

Both SDL and UML have a concept of *packages* for grouping and reuse of elements of the specification. Both support the nesting of packages (2). The abstract syntax of UML describes the contents of a package as a set of *PackageableElement*s, a synonym for all elements that can be contained in a package. SDL describes sets of the elements that can be contained in a package, e.g. Signal-definition-**set**. Common packageable elements in SDL and UML are agents/classes (3), signals (4) and composite states/statemachines (5).

SDL-2000 (AS1)	Common AS	UML 2.0 (derived AS)
Package-definition ::=(::)	Package-definition ::=(::)	Kernel_Package ::=(::)
1 Package-name	1 Package-name	1 [String]
2 Package-definition-**set**	2 Package-definition-**set**	2 Kernel_Package-**set**
Data-type-definition-**set**	3 Signal-definition-**set**	Kernel_PackageableElement-**set**
Syntype-definition-**set**	4 Agent-type-definition-**set**	Kernel_PackageMerge-**set**
4 Signal-definition-**set**	5 Statemachine-**set**	Kernel_VisibilityKind
Exception-definition-**set**		Kernel_ElementImport-**set**
3 Agent-type-definition-**set**		Kernel_PackageImport-**set**
5 Composite-state-type-definition-**set**		Kernel_Constraint-**set**
		Kernel_PackageableElement ::=(=)
Procedure-definition-**set**		3 StructuredClasses_Class
		5 BehStateMachines_StateMachine
		4 Communications_Signal
		...

4.2 Agent-type/Class

UML 2.0 introduces *structured classes*, which are classes extended with internal structure and ports. Structured classes are semantically and syntactically similar to Agent-types in SDL. Both have an internal structure of properties (respectively agents, 9), connectors (channels, 7) and gates (ports, 6). Both agent-types and structured classes can specialise other agent-types and structured classes (2), however SDL only supports single inheritance while UML supports multiple inheritance.

Behaviour is associated with an Agent-type as a State-machine-definition, which consists of a name and a Composite-state-type-identifier (8). Behaviour is associated with structured classes by a Behavior-Identifier (8). Behaviour in the abstract syntax of UML is a synonym for statemachines and other behaviour models. Statemachines are syntactically similar to composite-state-types in SDL. The abstract syntax of the two languages differs slightly, since UML does not have a State-machine-definition. In the common abstract grammar, we include the State-machine-definition but discard the name associated with it, since it does not exist in UML.

SDL-2000 (AS1)	Common AS	UML 2.0 (derived AS)
Agent-type-definition ::=(::)	Agent-type-definition ::=(::)	StructuredClasses_Class ::=(::)
1 Agent-type-name	1 Agent-type-name	1 [String]
Agent-kind	2 [Agent-type-identifier]	…
2 [Agent-type-identifier]	3 Signal-definition-**set**	Kernel_VisibilityKind
Agent-formal-parameter *	4 Agent-type-definition-**set**	Kernel_Classifier-Identifier-**set**
Data-type-definition-**set**	5 Statemachine-**set**	2 StructClasses_Class-Identifier-**set**
Syntype-definition-**set**	9 Agent-definition-**set**	[Kernel_Type]
3 Signal-definition-**set**	6 Port-definition-**set**	Kernel_ElementImport-**set**
Timer-definition-**set**	7 Channel-definition-**set**	Kernel_PackageImport-**set**
Exception-definition-**set**	8 [Agent-behaviour]	Kernel_Constraint-**set**
Variable-definition-**set**		Kernel_Behavior-**set**
4 Agent-type-definition-**set**	Agent-behaviour ::=(::)	8 [Kernel_Behavior-Identifier]
5 Composite-state-type-	8 Statemachine-identifier	Boolean /*isActive*/
definition-**set**		Communications_Reception-**set**
Procedure-definition-**set**		6 Ports_Port-**set**
9 Agent-definition-**set**		7 CompStruct_Connector-**set**
6 Gate-definition-**set**		9 IntStruct_Property-**set**
7 Channel-definition-**set**		Kernel_Property *
8 [State-machine-definition]		Kernel_Classifier-**set**
		Kernel_Operation *
State-machine-definition		
::=(::) State-name		Kernel_Classifier ::=(=)
8 Composite-state-type-		4 StructuredClasses_Class
identifier		5 BehStateMachines_StateMachine
		3 Communications_Signal
		…

4.3 Signals

Signal types exist in SDL and UML to describe communication between agents/objects. Signals have a name (1) and parameters, which are represented by sorts in SDL and properties in UML. While representing similar concepts, the abstract syntax of sorts and properties are different, therefore signals in the common abstract grammar have no parameters.

SDL-2000 (AS1)	Common AS	UML 2.0 (derived AS)
Signal-definition ::=(::)	Signal-definition ::=(::)	Communications_Signal ::=(::)
1 Signal-name	1 Signal-name	Kernel_Property *
Sort-reference-identifier*		Boolean
		Boolean
		Kernel_VisibilityKind
		1 [String]
		Kernel_Classifier-Identifier-**set**
		Kernel_Generalization-**set**
		Kernel_ElementImport-**set**
		Kernel_PackageImport-**set**
		Kernel_Constraint-**set**

4.4 Channel/Connector

Channels/connectors connect gates/ports. In SDL, a channel has one or two channel-paths. In case of two channel-paths, the channel is bi-directional and the originating gate of the first path is the destination gate of the second path and vice versa. In UML, the connector connects two or more ports. In the common AS, a channel is a set of channel-ends (2), which is a pair of ports (3). No direction is specified.

SDL-2000 (AS1)	Common AS	UML 2.0 (derived AS)
Channel-definition ::=(::)	Channel-definition ::=(::)	IntStruct_Connector ::=(::)
1 Channel-name	1 Channel-name	IntStruct_Connector-Identifier
[NODELAY]	2 Channel-end-**set**	2 Ports_ConnectorEnd *
2 Channel-path-**set**	Channel-end ::=(::)	/* end, 2..* */
Channel-path ::=(::)	3 Port-identifier	[Kernel_Association-Identifier]
3 Originating-gate	3 Port-identifier	Boolean
3 Destination-gate		Boolean
Signal-identifier-**set**		[Kernel_VisibilityKind]
		1 [String]
		[Kernel_Type]
		Ports_ConnectorEnd ::=(::)
		3 [IntStruct_Connectable Element-Identifier]
		[IntStruct_Property-Identifier]
		...

4.5 Gate/Port

Gates/ports are endpoints for channels/connectors. Gates specify valid signals for both directions, while ports have required and provided interfaces (2, 3).

SDL-2000 (AS1)	Common AS	UML 2.0 (derived AS)
Gate-definition ::=(::)	Port-definition ::=(::)	Ports_Port ::=(::)
1 Gate-name	1 Port-name	Boolean
2 In-signal-identifier-**set**	2 Signal-identifier-**set**	Boolean
3 Out-signal-identifier-**set**	3 Signal-identifier-**set**	3 Interfaces_Interface-Identifier **-set** /*required*/
		2 Interfaces_Interface-Identifier **-set** /*provided*/
		Ports_Port-set-Identifier
		IntStruct_ConnectorEnd-**set**
		...
		[Kernel_VisibilityKind]
		1 [String]
		[Kernel_Type-Identifier]
		[Kernel_ValueSpecification]
		[Kernel_ValueSpecification]

4.6 Agent/Property

Agents and properties are both instances of a type (2) (agent-type in SDL, structured class in UML). Both specify upper and lower bounds for the number of instances (3). While the lower bound in UML is optional, it is required in SDL.

SDL-2000 (AS1)	Common AS	UML 2.0 (derived AS)
Agent-definition ::=(::)	Agent-definition ::=(::)	IntStruct_Property ::=(::)
1 Agent-name	1 Agent-name	...
Number-of-instances	2 [Agent-type-identifier]	Kernel_AggregationKind
2 Agent-type-identifier	Number-of-instances	Kernel_Property* /*subset*/
		Kernel_Property* /*refined*/
Number-of-instances::=(::)	Number-of-instances ::=(::)	[Kernel_ValueSpecification]
3 Initial-number	3 [Initial-number]	[Kernel_Association-Identifier]
3 [Maximum-number]	3 [Maximum-number]	Ports_ConnectorEnd-Identifier-**set**
		...
Initial-number ::=(=)Nat	Initial-number ::=(=) Nat	[Kernel_VisibilityKind]
Maximum-number ::=(=)	Maximum-number ::=(=) Nat	1 [String]
Nat		2 [Kernel_Type-Identifier]
		3 [Kernel_ValueSpecification]
		/*upper*/
		3 [Kernel_ValueSpecification]
		/*lower*/

4.7 Composite-state-type/Statemachine

Composite-state-types as well as statemachines have a name (1), a sequence of parameters (3) and an identifier of the composite-state-type/statemachine that they specialise (2), if any. In UML, a statemachine has one or more regions that contain states and transitions. The equivalent in SDL is a Composite-state-graph (one region) or a State-aggregation-graph (two or more regions). A Composite-state-graph contains a State-transition-graph which contains the states of the Composite-state-type. A Region in UML maps to a State-transition-graph in SDL. Both contain the states (5) and transitions of the composite-state-type/statemachine. Multiple regions are not included in the common AS, because of the different syntax and semantics in SDL and UML.

SDL-2000 (AS1)	Common AS	UML 2.0 (derived AS)
Composite-state-type-definition	Statemachine ::=(::)	BehSM_StateMachine ::=(::)
::=(::)	1 Statemachine-name	BehSM_Pseudostate-**set**
1 State-type-name	2 [Statemachine-Identifier]	4 BehSM_Region +
2 [Composite-state-type-	3 Parameter *	2 [BehSM_StateMachine-Identifier]
identifier]	Statemachine-graph	3 Kernel_Parameter *
3 Composite-state-formal-		BasBeh_Behavior-Identifier-**set**
parameter *	Statemachine-graph ::=(::)	[BasBeh_BehavioralFeature
State-entry-point-definition-**set**	4 State-transition-graph	-Identifier]
State-exit-point-definition-**set**		Kernel_Constraint-**set**
Gate-definition-**set**	State-transition-graph ::=(::)	Kernel_Constraint-**set**
...	5 State-**set**	Kernel_Property-**set**
Composite-state-type-		Kernel_Class-Identifier-**set**
definition-**set**		Kernel_Classifier-**set**
Variable-definition-**set**		Kernel_Operation-**set**
Procedure-definition-**set**		...
[Composite-state-graph |		Kernel_VisibilityKind
State-aggregation-node]		1 [String]
		Kernel_Classifier-Identifier-**set**
Composite-state-graph ::=(::)		Kernel_Generalization-**set**
4 State-transition-graph		Kernel_PackageElement-**set**
[Entry-procedure-definition]		Kernel_ElementImport-**set**
[Exit-procedure-definition]		Kernel_PackageImport-**set**
Named-start-node-**set**		Kernel_Constraint-**set**
State-transition-graph ::=(::)		BehSM_Region ::=(::)
[On-exception]		5 BehSM_Vertex-**set**
[State-start-node]		BehSM_Transition-**set**
5 State-node-**set**		[BehSM_Region]
Free-action-**set**		[Kernel_VisibilityKind]
Exception-handler-node-**set**		[String]

4.8 State-node/State

State-nodes in SDL are similar to states in UML, however the syntax is different. Both have a name (1) and an identifier of the composite-state-type/statemachine that is the submachine of this state (2), if any. States are the source of transitions, but in SDL these transitions are associated with the trigger of the transition (Input-node) and in UML with the state itself.

SDL-2000 (AS1)	Common AS	UML 2.0 (derived AS)
State-node ::=(::) 1 State-name [On-exception] Save-signalset Input-node-set Spontaneous-transition-set Continuous-signal-set Connect-node-set 2 [Composite-state-type-identifier]	State ::=(::) 1 State-name 2 [StateMachine-Identifier]	BehSM_State_Concrete ::=(::) [Beh_ConnectionPointReference] 2 [BehSM_StateMachine-Identifier] ... Com_Trigger-Identifier-**set** BehSM_Region-**set** [BehSM_State-Identifier] BehSM_Transition-Identifier-**set** BehSM_Transition-Identifier-**set** [Kernel_VisibilityKind] 1 [String]

4.9 Parameter

Like signals, parameters in the common AS have a name (1) but no sort or type, since the abstract syntax of sorts and types is different.

SDL-2000 (AS1)	Common AS	UML 2.0 (derived AS)
Parameter ::=(::) 1 Variable-name Sort-reference-identifier	Parameter ::=(::) 1 Variable-name	Kernel_Parameter ::=(::) 1 [String] Kernel_ParamerDirectionKind ... [Kernel_Type-Identifier] [Kernel_ValueSpecification] [Kernel_ValueSpecification]

4.10 Case Study

Instead of mapping the meta-model of UML to an abstract grammar, we can map the abstract grammar of SDL to a meta-model. In this way, we can avoid the mapping loss from meta-models to abstract grammars. The result of mapping a part of the abstract grammar of SDL to a meta-model can be seen in Annex C. The mapping covers parts of the object behaviour.

Based on the SDL and the UML meta-model, we can compare the abstract syntax as in the case of abstract grammars. We can identify common classes in both meta-models, e.g. State and StateNode or StateMachine and CompositeStateTypeDefinition. As in the abstract grammar, there is an association between these classes in both meta-models with the same multiplicity. However, where the abstract grammars of the languages are different, so are the meta-models. E.g., states in the UML meta-model are associated with transitions, while states in SDL are associated with input-nodes. We assume that the representation of the abstract grammar has no influence on the coverage of the extracted common abstract syntax.

4.11 Discussion

In this section, we have compared corresponding excerpts of the SDL abstract syntax AS1, and the abstract syntax AS derived from the UML meta-model by applying the mapping defined in Section 3. As a result, we have extracted a common abstract syntax for UML and SDL. As a pure syntactical mapping turned out to be infeasible, additional knowledge about the language definitions of UML and SDL was taken into account for this extraction. As a result, a substantial coverage of syntax elements of UML and SDL has been achieved for structural language elements, i.e., Packages, Agent-types, Agents, Channels and Gates. However, there is little coverage for behavioural language elements only, despite the fact that both SDL and UML have a concept of hierarchical statecharts. We therefore draw the conclusion that in order to harmonise UML and SDL, an extraction on a purely syntactical basis is not sufficient. In order to obtain the best possible coverage, the comparison should be made on a semantic basis in the first place. Here, the syntactical comparison can provide valuable information and feedback.

5 Semantic Mapping Between UML Statecharts and SDL Process Graphs

Following the conclusions reached so far, we now compare corresponding language elements of UML and SDL on a semantic level. For this comparison, we choose UML statecharts and SDL process graphs, respectively. UML statecharts have a complete semantics with few variation points. Several attempts to formally define the behaviour of statecharts exist, e.g. [6].

The syntactic comparison of UML and SDL revealed that the abstract syntax of statecharts and process graphs is very different. However, there are several language elements in both languages that have a similar notation and represent similar concepts, despite major syntactic differences. E.g., both languages have the concept of a guarded transition. In the following, we compare the semantics of corresponding constructs of UML statecharts and SDL process graphs. We omit those cases where corresponding constructs are semantically different. Work that provides mappings for these cases, too, can be found in [5] for SDL-92 to UML 1.3.

5.1 States

Both UML [1] and SDL [2] have the concept of states as a condition in which the state machine waits for an event to occur. The state (or set of states) the state machine waits in is called the active state. In UML, it can also model a condition in which the process performs a certain activity (do-activity). There are several kinds of states (see [1], 9.3.11):

- **Simple States:** A simple state is a state without sub-states (Fig. 2).

Fig. 2: Simple states in SDL and UML

- **Composite States:** A composite state is a state that contains sub-states, which can again be composite states. In UML, the sub-states of a composite state are partitioned into one or more regions. If the state machine is in the composite state, it is also in *exactly one* sub-state of each region (this applies recursively). SDL distinguishes between composite states, which have a set of sub-states of which exactly one is active when the state is entered (composite state with one region in UML), and state aggregations, which consist of several composite states that are interpreted in an interleaving manner (composite state with multiple regions in UML).
- **Submachine States:** UML introduces submachine states, which are semantically equivalent to UML composite states. Syntactically, they are closer to SDL composite states than the composite states in UML (Fig. 3). Submachine states make it possible to build the specification in a modular way.

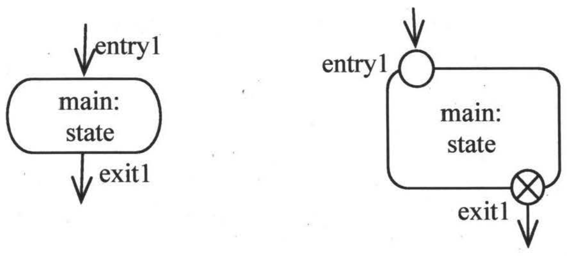


Fig. 3: Submachine (composite) states and entry/exit-points in SDL and UML

Unlike SDL, simple states in UML can have entry-activities (exit-activities) that are executed when the state is entered (left).

Composite states in SDL and submachine states in UML (both referred to as composite states in the future) have entry- and exit-procedures/activities. A transition targeting the composite state leads to an entry of the state at the default entry point (*initial* pseudostate in UML, unnamed state start node in SDL). A transition to an entry connection-point reference of the composite state leads to an entry of the state at the entry point that is referenced by the connection-point reference. If a region of an UML composite state reaches an exit point, or if all regions reach the final state, the composite state is left at the corresponding exit connection point reference (completion transition in case of leaving via final states). In SDL, the composite state is left when all state partitions have reached a return node. The state is left via the corresponding exit point of the composite state. If more than one exit point is valid, one is chosen in an indeterministic way.

5.2 Signals and Events

State transitions are triggered by *signals* in SDL and *events* in UML. An SDL agent has an input port associated with its state machine, in which signals are queued in the order of their arrival time (see [3], 9). If a signal is *saved* in a state, it is not enabled. The signal that is dispatched is the first enabled signal in the queue; the saved signals in the queue are retained in the order of their arrival (see [3], 11.7).

UML objects have an event pool, with an unspecified ordering of events (see [1], 7.3.5), with the exception of *completion events*, which are dispatched before any other event in the pool (see [1], 9.3.14). If an event is *deferred* in a state, it is retained in the event pool as long as the state machine is in a state where the event is deferred, or a transition for this event is enabled (see [1], 9.3.11). No priority between events that become enabled again and non-deferred events is specified. If an event was deferred in the previous state and is not deferred in the current state, it has no priority over an event that was not deferred at all.

SDL signals and UML events are both referred to as 'events' in the remainder of the document.

5.3 Timers

In SDL, a *timer* can be set to expire at a specified time or after a duration. When the timer expires, an event is generated and put into the input queue. This can trigger a transition when the event is dispatched (see [3], 11.15). UML has TimeEvents that occur at a certain point in time or after a specified duration (though the starting time is not defined in the standard). TimeEvents can trigger a TimeTrigger (see [1], 7.3.27).

5.4 Transitions

Transitions in UML and SDL have a *run-to-completion* semantics, meaning that an event is completely processed before the next event can be handled.

Transitions with a trigger and no conditions have the same semantics in SDL and UML for simple and composite states. The transition in the active state is enabled when the event the transition is labelled with is dispatched. When the transition is fired, the source (composite) state of the transition is left, a sequence of actions is executed and the target (composite) state is entered. More than one transition can be enabled in a state for an event. When the state machine is in a composite state, and both the composite state and an active sub-state have a transition for the dispatched event, both transitions are enabled. In SDL and UML, the transition of the nested state has a higher priority than the transition of the containing state, and the transition of the sub-state is fired (see [3], 11.11 and [1], 9.3.12). This also applies to conflicts between deferred (saved) and consumed events (Fig. 4). In case of conflicts between consumed and deferred events in orthogonal states, the event is consumed. This is only possible in UML.

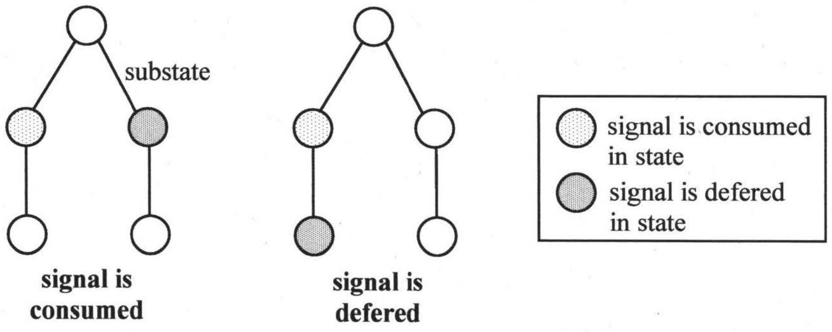


Fig. 4: Priority of consume and defer

States in SDL can not have more than one transition for the same event (see [3], 11.2), and orthogonal states must have a disjoint set of input signals (see [3], 11.11.2). However, in UML this is not the case. If there is more than one transition for the same event in the active state, one of them is processed. For every orthogonal region, if there is a transition for the dispatched event in the active state, it is enabled and can be processed. More than one transition, up to the number of orthogonal regions in the state, can be fired in the same run-to-completion step in arbitrary order (see [1], 9.3.12). In Fig. 5, if the orthogonal state is in the states s11 and s21 and the event sig1 is dispatched, both transitions are enabled and fired, since they do not conflict with each other.

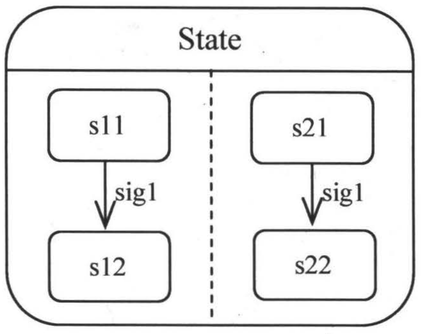


Fig. 5: State with two enabled transitions (UML)

Both SDL and UML support transitions that have *guards* (enabling conditions in SDL) (Fig. 6). Transitions are only enabled when their guard evaluates to true. If the guard evaluates to false, in SDL the event is not enabled and the next event is selected from the queue (see [3], 11.6). UML discards the event, unless it is explicitly deferred or there is another transition for this event that is enabled (see [1], 9.3.12 and 9.3.14). Therefore the semantics of guards differ between SDL and UML.

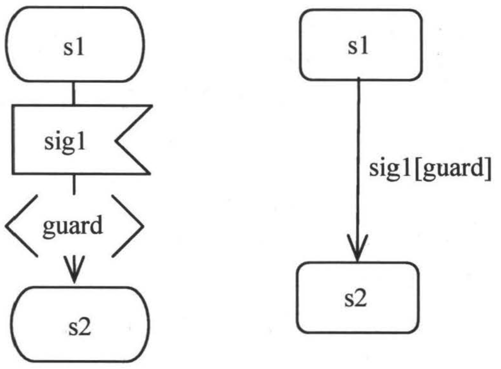


Fig. 6: Transitions in SDL and UML

Transitions (with guard) that are not explicitly labelled with an event are called *continuous signals* in SDL and *completion transitions* in UML (Fig. 7). Continuous signals are fired when their guard is true and the event queue is empty (see [1], 11.5). Completion transitions are fired when a completion event occurs and their guard is true. Completion events are dispatched before all other events. They occur when a do-activity is finished or when a composite state is left because all regions have reached a final state (see [3], 9.3.14). A completion transition in UML has a similar semantics as a *connect-node* in SDL. A connect-node originates from a composite state and is taken when the composite state is left via the default exit point. A connect-node cannot have a guard.

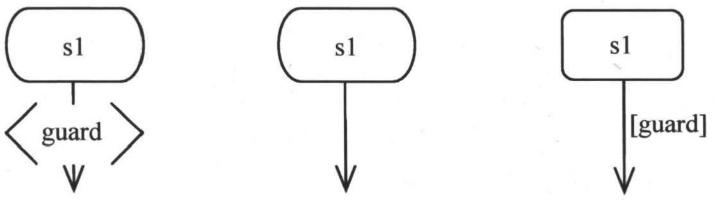


Fig. 7: Continuous signal, connect node and completion transition

5.5 History

UML has two *history pseudostates*, deep history and shallow history (see [1], 9.3.8). Entering a composite state over a history state leads to the restoration of the active states as it was when the composite state was left. For a deep history state this applies recursively for all substates. All entry activities of states that are entered are executed. In SDL a transition can end in a *history nextstate*. In this case, the next state is the one in which the transition originated. In case of a composite state, the state is re-entered and the entry-procedure is invoked (see [3], 11.12.2.1).

5.6 Actions

In UML, actions are used to describe behaviour. Actions take inputs and transform them into outputs, possibly modifying the state of the system. Outputs of actions can be connected to inputs of other actions by an activity flow, as in a statechart transition. UML supports a number of primitive actions that more complex actions can be mapped on. In SDL, transitions perform a sequence of actions. Actions in SDL manipulate data, output signals and call procedures. SDL supports a small number of specific actions important for communication systems.

- **Task/StructuralFeatureAction,VariableAction:** Tasks in SDL can contain assignments that are interpreted when the task is interpreted. In UML, assignments can be realised by a combination of actions. With ReadStructuralFeatureAction/ReadVariableAction and WriteStructuralFeatureAction/ WriteVariableAction, attributes and variables can be read and written. With ApplyFunctionAction, primitive functions can be applied.
- **Create/CreateObjectAction:** CreateObjectAction creates a new object for a given classifier, without further initialisation. The Create action in SDL creates a new agent in the scope of the creating agent. Its variables are created and its formal parameters initialised.
- **Procedure Call/CallOperationAction:** When a procedure is called in SDL, the interpretation continues at the start node of the procedure graph that was invoked by the call. It resumes after the call node when the interpretation of the procedure is finished. A CallOperationAction in UML leads to the execution of an operation in a local or remote object. If the call is synchronous, the interpretation of the transition resumes after the CallOperationAction when the operation is finished. How parameters and results are transmitted is not specified.
- **Output/SendSignalAction:** An output action in SDL leads to the creation of a signal instance of the specified type and parameters. The signal instance can have an agent set or an agent as target, or it can be transmitted to any agent reachable via a sequence of valid channels, possibly restricted by the via argument. In UML, a SendSignalAction leads to the creation of a signal instance of the specified type and parameters. The signal instance is transmitted to the target object. The path the signal takes, its transmission time and the order in which signals arrive is undefined.
- **Decision/Choice:** A decision node in SDL evaluates a question and selects an outgoing transition that has the answer to the question in its range. If no range of an outgoing transition is matched and an else-branch exists, the else-branch is selected; otherwise a NoMatchingResult exception is thrown. SDL forbids having the same answer in more than one range of an outgoing transition. When a Choice pseudostate is reached in UML, the guards of the outgoing transitions are evaluated, and one of the transitions whose guard evaluates to true is selected in an indeterministic way. If there is an else-branch and none of the guards evaluates to true, the else-branch is selected, otherwise the model is ill-formed.

6 Conclusions and Outlook

With regard to recent language developments, harmonisation and finally integration of languages are becoming urgent topics. With more notations being used during the development of a given system, the question whether these views are consistent is gaining importance. Also, in the context of large systems, the use of a mix of notations is getting more likely. Standardisation work to harmonise UML and SDL are important efforts towards the objective of having a set of languages that can be used together.

In this paper, we have argued that the harmonisation of languages requires a common syntactic and semantic basis. Following this line, we have first defined conceptually sound and well-founded mappings from meta-models – used to define the abstract syntax of UML – to abstract grammars – used by SDL –, and vice versa. By applying these mappings, we have then extracted common production rules, arriving at a common abstract grammar for several language constructs. While the results were encouraging for structural language elements, it turned out that the coverage was below expectations for behavioural constructs. From this experience, we have drawn the conclusion that an extraction on a purely syntactical basis is not sufficient. Therefore, we have compared language elements on a semantic basis, too. Here, the results of the syntactical study provided valuable information and feedback.

While the semantic comparison showed that there is indeed potential for the harmonisation of UML and SDL, it also revealed that without a common mathematical basis, the results that can be obtained will be of limited value. Without such a basis, any kind of reasoning has to remain informal, leaving too much room for concerns. Therefore, we conclude that future work should be directed towards a common semantic framework for UML and SDL, with the intention of having extensions of this framework to define them formally. Both languages are complex and sophisticated, so this will definitely not be an easy task. However, our experience with the definition of the SDL formal semantics has shown that this kind of work provides valuable feedback to the language designers, finally leading to an even better language.

References

[1] OMG Unified Modelling Language Specification, Version 2.0, 2003

[2] OMG Unified Modelling Language Specification, Version 1.3, 1999

[3] ITU Recommendation Z.100, Specification and Description Language. Geneva, 1999.

[4] ITU-T Recommendation Z.109, SDL combined with UML. Geneva, 2000.

[5] Bran Selic and Jim Rumbaugh: Mapping SDL to UML, Rational Software Whitepaper, 1999

[6] E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML State Machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, Abstract State Machines. Theory and Applications, pages 223-241. Springer-Verlag, 2000.

Annex A: Meta-model of UML Statemachines and corresponding Abstract Grammar

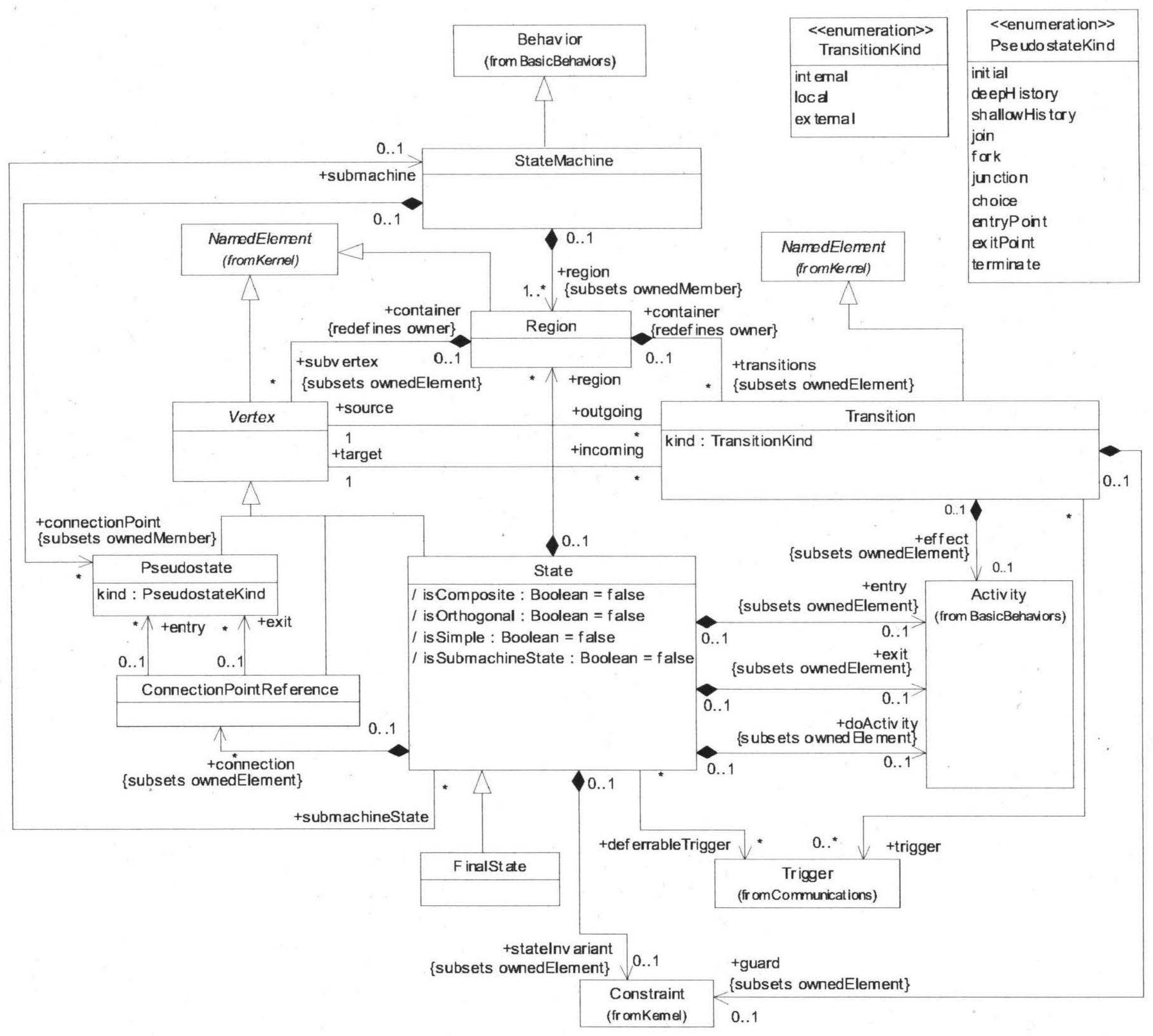


Fig. 8: Abstract syntax of UML statecharts (from the OMG UML language specification, [1])

```
BehSM_PseudostateKind ::=(=) INITIAL
            | DEEPHISTORY
            | SHALLOWHISTORY
            | JOIN
            | FORK
            | JUNCTION
            | CHOICE
            | ENTRYPOINT
            | EXITPOINT
            | TERMINATE

BehSM_TransitionKind ::=(=) INTERNAL
            | LOCAL
            | EXTERNAL
```

```
BehSM_Vertex ::=(=) BehSM_Pseudostate
            | BehSM_ConnectionPointReference
            | BehSM_State

BehSM_State ::=(=) BehSM_State_Concrete
            | BehSM_FinalState

BehSM_Pseudostate ::=(::)
            BehSM_PseudostateKind                     /* kind */
 /* from BehSM_Vertex */
            BehSM_Transition-Identifier-set           /* outgoing */
            BehSM_Transition-Identifier-set           /* incoming */
 /* from Kernel_NamedElement */
            [Kernel_VisibilityKind]                   /* visibility */
            [String]                                  /* name */

BehSM_ConnectionPointReference ::=(::)
            BehSM_Pseudostate-Identifier-set          /* entry */
            BehSM_Pseudostate-Identifier-set          /* exit */
 /* from BehSM_Vertex */
            BehSM_Transition-Identifier-set           /* outgoing */
            BehSM_Transition-Identifier-set           /* incoming */
 /* from Kernel_NamedElement */
            [Kernel_VisibilityKind]                   /* visibility */
            [String]                                  /* name */

BehSM_State_Concrete ::=(::)
            [Beh_ConnectionPointReference]            /* connection */
            [BehSM_StateMachine-Identifier]           /* submachine */
            [BasBeh_Activity]                         /* entry */
            [BasBeh_Activity]                         /* exit */
            [BasBeh_Activity]                         /* doActivity */
            [Kernel_Constraint]                       /* stateInvariant */
            Com_Trigger-Identifier-set                /* deferrableTrigger */
            BehSM_Region-set                          /* region */
            [BehSM_State-Identifier]                  /* redefinedState */
 /* from BehSM_Vertex */
            BehSM_Transition-Identifier-set           /* outgoing */
            BehSM_Transition-Identifier-set           /* incoming */
 /* from Kernel_NamedElement */
            [Kernel_VisibilityKind]                   /* visibility */
            [String]                                  /* name */

BehSM_FinalState ::=(::)
            [Beh_ConnectionPointReference]            /* connection */
            [Kernel_Constraint]                       /* stateInvariant */
            Com_Trigger-Identifier-set                /* deferrableTrigger */
            [BehSM_State-Identifier]                  /* redefinedState */
 /* from BehSM_Vertex */
            BehSM_Transition-Identifier-set           /* incoming */
 /* from Kernel_NamedElement */
            [Kernel_VisibilityKind]                   /* visibility */
            [String]                                  /* name */

BehSM_Transition ::=(::)
            BehSM_TransitionKind                      /* kind */
            [Kernel_Constraint]                       /* guard */
            Com_Trigger-Identifier-set                /* trigger */
            [BasBeh_Activity]                         /* effect */
            [BehSM_Transition-Identifier]             /* redefinedTransition */
            BehSM_Vertex-Identifier                   /* source */
            BehSM_Vertex-Identifier                   /* target */
 /* from Kernel_NamedElement */
            [Kernel_VisibilityKind]                    /* visibility */
            [String]                                   /* name */

BehSM_Region ::=(::)
            BehSM_Vertex-set                           /* subvertex */
            BehSM_Transition-set                       /* transitions */
            [BehSM_Region-Identifier]                  /* extendedRegion */
 /* from Kernel_NamedElement */
            [Kernel_VisibilityKind]                    /* visibility */
            [String]                                   /* name */

BehSM_StateMachine ::=(::)
            BehSM_Pseudostate-set                      /* connectionPoint */
            BehSM_Region-set                           /* region */
            [BehSM_StateMachine-Identifier]            /* extendedStateMachine */
 /* from BasBeh_Behavior */
            Kernel_Parameter-set                       /* parameter */
            BasBeh_Behavior-Identifier-set             /* redefinedBehavior */
            [BasBeh_BehavioralFeature-Identifier]      /* specification */
            Kernel_Constraint-set                      /* precondition */
            Kernel_Constraint-set                      /* postcondition */
 /* from Kernel_Class */
            Kernel_Property-set                        /* ownedAttribute */
            Kernel_Classifier-set                      /* nestedClassifier */
            Kernel_Operation-set                       /* ownedOperation */
 /* from Kernel_RedefinableElement */
            Boolean                                    /* isLeaf */
 /* from Kernel_PackageableElement */
            Kernel_VisibilityKind                      /* visibility */
 /* from Kernel_NamedElement */
            [String]                                   /* name */
 /* from Kernel_Classifier */
            Boolean                                    /* isAbstract */
            Kernel_Classifier-Identifier-set           /* redefinedClassifier */
            Kernel_Generalization-set                  /* generalization */
 /* from KernelNamespace */
            Kernel_PackageElement-set                  /* importedMember */
            Kernel_ElementImport-set                   /* elementImport */
            Kernel_PackageImport-set                   /* packageImport */
            Kernel_Constraint-set                      /* ownedRule */

BehSM_StateMachine-Identifier ::=(=) Identifier

BehSM_Pseudostate-Identifier ::=(=) Identifier

BehSM_Transition-Identifier ::=(=) Identifier

BehSM_Trigger-Identifier ::=(=) Identifier
```

Annex B: Common Abstract Grammar

```
Name ::=(=) Token
Identifier ::=(::) Qualifier Name
Qualifier ::=(=) Path-item +

Path-item ::=(=) Package-qualifier

Package-qualifier ::=(::) Package-name
```

```
Package-name ::=(=) Name
Agent-type-name ::=(=) Name
Signal-name ::=(=) Name
Channel-name ::=(=) Name
Port-name ::=(=) Name
Agent-name ::=(=) Name
Statemachine-name ::=(=) Name
State-name ::=(=) Name
Variable-name ::=(=) Name

Agent-type-identifier ::=(=) Identifier
Statemachine-identifier ::=(=) Identifier
Port-identifier ::=(=) Identifier

Package-definition ::=(::)
        Package-name
        Package-definition-set
        Agent-type-definition-set
        Signal-definition-set
        Statemachine-set

Agent-type-definition ::=(::)
        Agent-type-name
        [Agent-type-identifier]
        Signal-definition-set
        Agent-type-definition-set
        Statemachine-set
        Agent-definition-set
        Port-definition-set
        Channel-definition-set
        [Agent-behaviour]

Agent-behaviour ::=(::)
        Statemachine-identifier

Signal-definition ::=(::)
        Signal-name

Channel-definition ::=(::)
        Channel-name
        Channel-end-set

Channel-end ::=(::)
        Port-identifier
        Port-identifier

Port-definition ::=(::)
        Port-name
        Signal-identifier-set
        Signal-identifier-set

Agent-definition ::=(::)
        Agent-name
        [Agent-type-identifier]
        Number-of-instances

Number-of-instances ::=(::)
        [Initial-number]
        [Maximum-number]

Initial-number ::=(=) Nat
Maximum-number ::=(=) Nat
```

Statemachine ::=(::)
    Statemachine-name
    [Statemachine-identifier]
    Parameter *
    Statemachine-graph

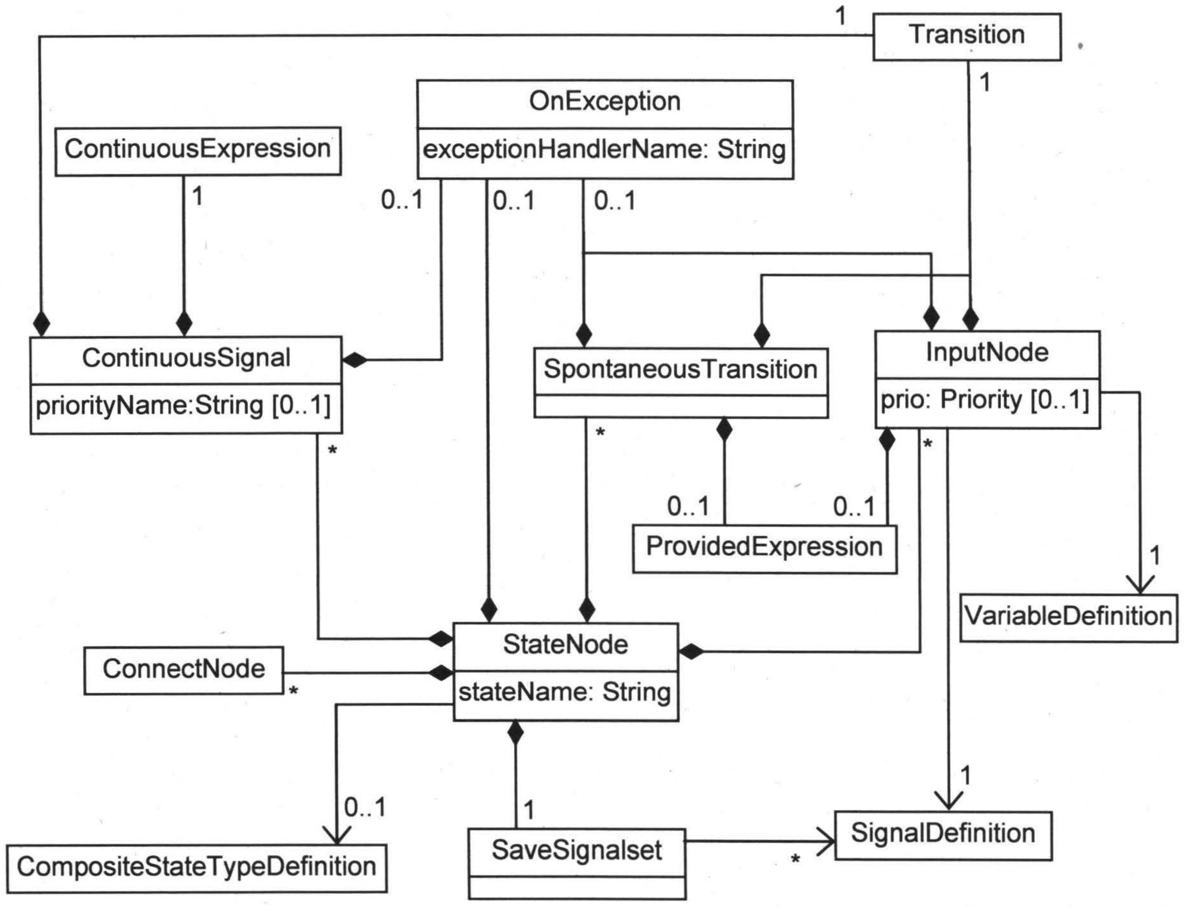
Statemachine-graph ::=(::)
    State-transition-graph

State-transition-graph ::=(::)
    State-set

State ::=(::)
    State-name
    [Statemachine-identifier]

Parameter ::=(::)
    Variable-name

Annex C: SDL Meta-model


Transition
1
1
OnException
exceptionHandlerName: String
ContinuousExpression
1
0..1
0..1
0..1
ContinuousSignal
priorityName:String [0..1]
SpontaneousTransition
InputNode
prio: Priority [0..1]
*
*
*
0..1
0..1
ProvidedExpression
1
VariableDefinition
StateNode
stateName: String
ConnectNode
*
0..1
1
1
CompositeStateTypeDefinition
SaveSignalset
*
SignalDefinition