Formalisation of the UML Profile for SDL - A Case Study

Rüdiger Grammes

October 27, 2006

Abstract. With the UML 2.0 standard, the Unified Modeling Language took a big step towards SDL, incorporating many features of the language. SDL is a mature and complete language with formal semantics. The Z.109 standard defines a UML Profile for SDL, mapping UML constructs to corresponding counterparts in SDL, giving them a precise semantics. In this report, we present a case study for the formalisation of the Z.109 standard. The formal definition makes the mapping precise and can be used to derive tool support.

Contents

1 Introduction **2**

2 UML Profiles **2**

2.1 UML Profile Definition 3

3 The UML Profile for SDL **4**

4 Mapping Abstract Syntax Representations **5**

5 Formalisation of the UML Profile for SDL **6**

6 Survey of an SDL-style Formalisation Approach **7**

6.1 Formalising the Constraints. 7

6.2 Mapping the Metamodel to an Abstract Grammar. 8

6.3 Transformations on the UML Abstract Grammar. 10

6.4 Formalisation of UML to SDL Mapping. 11

7 Conclusions **16**

1 Introduction

Since the Unified Modeling Language [11, 12] was introduced, the development of SDL [5, 7] has been influenced by UML, and vice versa. The mutual influence became especially apparent with the most recent language standards, SDL-2000 [7] and UML 2.0 [12]. With SDL-2000, the first version of the Z.109 standard [6] was introduced, which described the combined use of SDL and UML 1.3 [11] by providing a mapping from UML to SDL, using UML profiles.

With the UML 2.0 standard, the Unified Modeling Language took a big step towards SDL, incorporating many features of the language. For example, structured classes model architecture in a fashion similar to SDL. UML 2.0 also comes with a mature UML profile mechanism, defining it as a specific meta-modelling technique. Profiles have become a part of the UML meta-model, defined in the Profile package, giving UML profiles a formal abstract syntax. The new version of the Z.109 standard [10] takes these changes into account and defines a UML Profile for SDL based on the UML 2.0 and SDL-2000 standards.

SDL is a more mature and complete language than UML, with few semantic variation points and a formal semantics. The UML Profile for SDL utilises this by taking SDL as the semantic basis for UML. On the other hand, using UML as front end language utilises its advantages in the early phases of software development, and its integration of different modelling techniques.

The aim of this report is to survey an approach to formalise the UML Profile for SDL. This approach is based on the formalisation of the static semantics of SDL [8], and the formal mapping of meta-models to abstract grammars defined in [3]. The report is structured as follows: We describe UML profiles in Section 2, and the UML Profile for SDL in Section 3. In Section 4 we give a short overview over the mapping defined in [3]. Section 5 describes the concept of the formalisation, and Section 6 contains our survey of the formalisation. In Section 7, we draw conclusions from our work.

2 UML Profiles

The Unified Modeling Language aims at being a universal language for modelling software systems in the early phases of software development, particularly the requirements and design phases. To achieve this goal, UML provides only a complete formal abstract syntax definition of the language, while the semantics definition is imprecise and only partially defined. *Semantic variation points* in the language definition identify parts of the semantics that are explicitly left open for interpretation, or where alternative interpretations are provided. For example, the event pool of a classifier instance is a collection of events that occurred at this instance. Events in the event pool can trigger classifier behaviour. The order in which the events are processed is intentionally left open. This enables a tool provider to implement a strategy that fits the target domain of the tool within the framework given by UML, for example first-in-first-out or priority based strategies.

Semantic variation points make UML a flexible modelling language that can be adapted for a large variety of target domains. Tool providers resolve semantic variation points when implementing a subset of UML, providing a domain-specific solution. However, these solutions are tool-specific, not standardised and often proprietary. This is a disadvantage for the interoperability of UML tools.

UML provides the UML profile mechanism to extend and adapt existing meta-models. Using UML profiles, it is possible to give precise semantics to parts of the language, and to tailor it for different platforms and domains. UML profiles are tool-independent and can be defined as separate standards, augmenting the UML language definition. Standardised UML profiles include profiles for CORBA, quality of service and real-time, testing, and many more.

UML profiles are not a first-class extension mechanism, that is, it is not possible to modify existing semantics of UML. Profiles can add constraints to a meta-model, provide semantics that does not conflict with the semantics of the meta-model, and add different notation for already existing symbols. This ensures that the model with applied stereotypes is still a valid UML model, which can be processed by a UML tool with sufficient compliance to the UML standard. For extensions that modify the semantics of UML, the meta-model itself must be modified. However, this is not recommended.

2.1 UML Profile Definition

UML 2.0 defines the UML profile mechanism as a part of the UML meta-model, giving it a formal abstract syntax. Profiles and stereotypes are integrated as specialisations of packages and classes, respectively. The notation to be used for defining UML profiles is generally left unspecified. The Z.119 standard [9] gives a guideline for defining UML profiles for ITU languages in a similar fashion to the UML superstructure document, describing semantics and constraints of a stereotype using informal language and OCL [14].

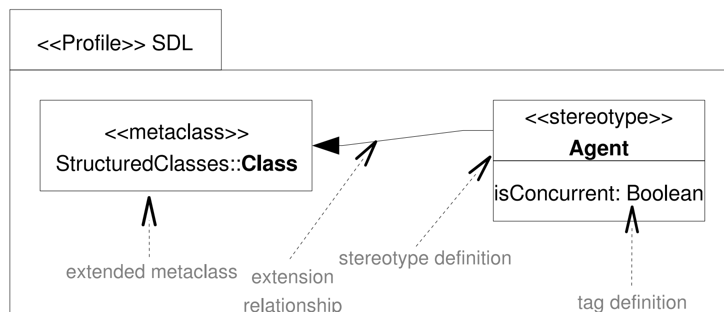

Figure 1: Profile package with stereotype

A profile is a specialised package that is applied to other packages (including profiles)

via a *profile application*. A profile uses the same notation as a package, with the keyword «profile» attached to the name of the package.

The profile consists of a set of owned *stereotypes*. A stereotype is a kind of meta-class that is linked to a meta-class of the referenced meta-model. Like classes, stereotypes can have properties, called *tag definitions*. Applying a stereotype to a meta-class adds the constraints, semantics and notations of the stereotype to the meta-class. Figure 1 shows the graphical notation of a profile SDL, which contains a stereotype Agent that extends the meta-class Class from package StructuredClasses. Stereotype Agent defines a tag definition isConcurrent of type Boolean. Semantics and constraints can be added to Agent as long as they don't conflict with existing semantics and constraints.

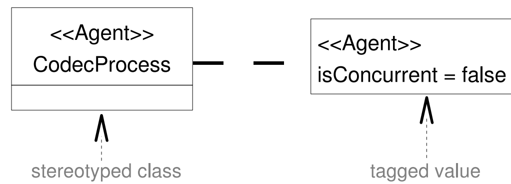

Figure 2: Stereotyped class

Figure 2 shows a UML model with stereotype Agent applied, using the standard notation for stereotyped UML classes. If the stereotype Agent defines a notation, for example the graphical syntax of process agents in SDL, it can be used instead. Tag value isConcurrent, defined in the stereotype, is set to false.

3 The UML Profile for SDL

The UML Profile for SDL gives a precise meaning to a subset of UML by mapping UML meta-model elements to elements of the SDL abstract syntax. Several meta-model classes are stereotyped, defining constraints and semantics to tailor the language to SDL. The semantics is defined as a mapping of the stereotyped meta-model classes to the abstract syntax of SDL. The meta-model and abstract syntax elements related by this mapping bear a strong resemblance to the common abstract syntax derived in [3]: for example, packages are mapped to *Package-definitions*, active classes to *Agent-definitions* and Signals to *Signal-definitions*. Mapping syntax elements provides a greater flexibility than the common syntax and semantics approach. For example, while UML guards do not have a direct representation in SDL, they can be mapped to a decision symbol at the start of the outgoing transition.

For each stereotype included in the profile, several aspects are defined:

- **Attributes** (tag definitions): Additional attributes defined by the stereotype that can be set in the model. Attributes give additional information that can otherwise not be expressed in the meta-model, and that is important for the mapping of model-elements to the abstract syntax. For example, the stereotype

«ActiveClass», which extends classes, defines the attribute isConcurrent of type Boolean. A class with stereotype «ActiveClass» is mapped to an *Agent-type-definition* in the abstract syntax, and attribute isConcurrent defines the *Agent-kind* of the *Agent-type-definition* - BLOCK if true and PROCESS if false.

- **Constraints**: Constraints define additional checks and conditions that the meta-model must satisfy. The meta-model is constrained to a subset for which a mapping to SDL is provided.
- **Semantics**: Gives a precise semantics to meta-model elements by describing a mapping to the abstract syntax of SDL. Meta-model elements are mapped directly to the AS1 of SDL, bypassing transformations in SDL from AS0 to AS1. The following describes a mapping of transitions with a ChangeEvent as trigger.

  > If the trigger event of a «Transition» Transition is a ChangeEvent, the transition is mapped to a *Continuous-signal*. The changeExpression maps to the *Continuous-expression* of the *Continuous-signal*. The effect property maps to the *Graph-node* list of the *Transition* of the *Continuous-signal*. The priority maps to the *Priority-name*.

- **Notation**: Describes the notation to be used for the stereotyped model element. Z.109 almost exclusively uses UML standard syntax. For some elements, additional textual syntax is introduced.

4 Mapping Abstract Syntax Representations

Mapping UML specifications to SDL, a mapping between the different abstract syntax representations of the languages is needed. In [3], we have provided such a mapping, from UML meta-models to SDL abstract grammars, and vice versa. For the UML Profile for SDL, only the mapping from meta-models to abstract grammars is of interest.

meta-model element	**abstract grammar element**
meta-model class	production rule
abstract meta-model class	synonym
enumeration	synonym
attribute	right hand side of production rule
association	right hand side of production rule
specialisation	right hand side of synonym

Table 1: Mapping of meta-model elements

Table 1 gives an overview over the mapping from meta-models to abstract grammars. In order to perform the mapping, the meta-model must first be flattened, recursively copying attributes and associations to subclasses of a class. Meta-model classes are mapped to production rules of the abstract grammar. Attributes and associations of

the meta-model class are mapped to the right hand side of the corresponding production rule. An abstract meta-model class is mapped as a synonym for its subclasses.

5 Formalisation of the UML Profile for SDL

The UML Profile for SDL is defined in an informal fashion, similar to the UML language definition. The SDL language definition, on the other hand, includes a complete formalisation of static and dynamic aspects of the language. In order to carry over the mathematical precision of SDL to the subset of UML covered by the profile, it was proposed to create a formal definition of Z.109. A formal definition of Z.109 has several advantages. For example, constraints formulated using the Object Constraint Language (OCL) can be automatically checked by many UML tools. From an operational formal definition of the mapping to the SDL abstract grammar, tool support can be automatically generated, as it is done in the case of the formal semantics of SDL [15].

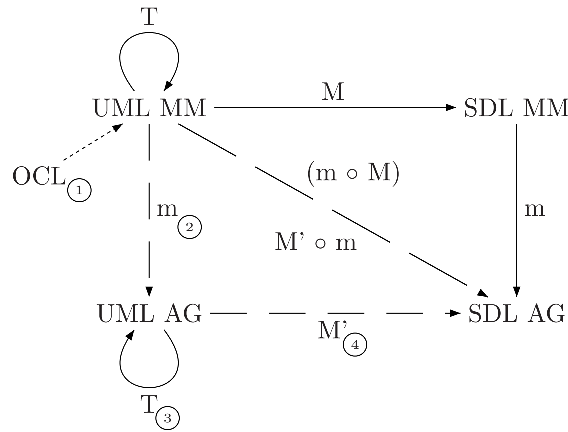

Figure 3: Mapping from UML meta-model to SDL abstract syntax

Figure 3 provides an overview over the steps taken in the profile definition. The intent is to provide a mapping from model elements of UML, described by a meta-model (UML MM), to abstract syntax elements of SDL, described by an abstract grammar AS1 (SDL AG). This mapping consists of two orthogonal steps: a mapping from UML to SDL (M), and a mapping between abstract syntax representations (m), from meta-models to abstract grammars (as described in Section 4). In addition to the mapping, constraints are defined on the meta-model, for example using OCL, and transformations (T) are performed on the UML side, since the mapping targets the already reduced abstract grammar AS1.

In [16], the semantics of the UML Profile for Communicating Systems is defined by a mapping of the UML meta-model to the abstract grammar AS1 of SDL. The mapping is defined by pre- and post-conditions on the meta-model and the abstract grammar,

using OCL [14]. The correctness of a concrete mapping can be verified using these constraints.

A way to perform the mapping is to map the UML meta-model to a UML abstract grammar first (m), then to map the UML abstract grammar to SDL (M'). The mapping between the abstract syntax representations can be derived from the mapping described in Section 4. The mapping M' is a mapping between two different abstract grammars. Such a mapping can be found in the formalisation of the static semantics of SDL (Z100 Annex F Part 2 [8]), where the abstract grammar AS0 is mapped to the abstract grammar AS1. Generally, the mapping M' is more complicated than the mapping in Z100.F2, since the mapping is performed between two different languages. We survey this approach in Sections 6.1 to 6.4, corresponding to the numbers given in Figure 3.

The mapping of the UML Profile for SDL covers a subset of SDL. Core features of SDL that are not covered include *timers*, *exceptions*, *enabling conditions*, *entry-* and *exit-procedures*. This subset defines an *SDL Profile*, for which a tailor-made formal semantics can be extracted [1, 4].

6 Survey of an SDL-style Formalisation Approach

In this section, we introduce a partial formalisation of the Z.109 standard, with the focus on transitions. The approach is to apply the techniques used for the formalisation of the static semantics of SDL as much as possible. This gives us the advantage of using an approach that has been applied successfully before, and for which tool support is available [15], allowing us to concentrate on the formalisation itself.

6.1 Formalising the Constraints.

The stereotypes in the UML Profile for SDL introduce additional constraints on the meta-model classes they extend. For the formalisation of these constraints, the Object Constraint Language (OCL) [14], which is used thoughout the UML superstructure specification, is a self-evident choice. The OCL is tailor-made for specifying constraints for MOF-compatible [13] meta-models. It provides a logic that allows navigation over properties and association ends of classifiers. Following are the constraints specified for the «Transition» stereotype formulated in OCL. Unless specified otherwise, all OCL expressions are formulated in the context of meta-class Transition.

- The Transition shall have kind == external or local. The UML concepts of internal transitions are not allowed.

  $self.kind = \#external$ **or** $self.kind = \#local$

- The port of the Trigger[1] shall be empty.

  $self.trigger{-}{>}forAll(t \mid t.port{-}{>}isEmpty())$

[1] The UML meta-model actually defines a set of triggers for a transition.

- In the Transition set defined by the outgoing properties of a State, the signal property of each event property that is a SignalEvent of each trigger shall be distinct.

  context *State*
  self.outgoing−>*forAll*(*t1*,*t2*: TRANSITION | *t1.trigger*−>*select*(*event.oclIsKindOf*(SIGNALEVENT)).*event.signal*−>*intersection*(*t2.trigger*−>*select*(*event.oclIsKindOf*(SIGNALEVENT)).*event.signal*)−>*isEmpty*())

- The event property of the trigger property shall be a MessageEvent or ChangeEvent.

  self.trigger−>*forAll*(*t* | *t.event.oclIsKindOf*(MESSAGEEVENT) **or** *t.event.oclIsKindOf*(CHANGEEVENT))

- The effect property shall reference an Activity.

  self.effect−>*notEmpty*() **implies** *self.effect.oclIsKindOf*(ACTIVITY)

While the informally specified constraints of the «Transition» stereotype are intuitive and easy to understand, three issues were discovered when specifying the constraints in OCL, two of them concerning multiplicities.

- A transition in UML can have an arbitrary number of triggers, while the stereotype constraints only assume a single trigger at most. The OCL constraints were formulated in a way that allows an arbitrary number of triggers, however, there should be an additional constraint that a transition should have at most one trigger.
- The informally specified constraints leave it unclear if the effect property is allowed to be empty. This has been clarified in the formalisation.
- The third constraint is formulated more naturally in the context of «State», since it deals with sets of transitions of states. Therefore, the constraint should be part of the stereotype «State».

6.2 Mapping the Metamodel to an Abstract Grammar.

In order to apply the approach from the formalisation of the static semantics of SDL, which provides a mapping between two abstract grammars, we first have to provide a mapping from the meta-model of the UML profile to a UML abstract grammar (mapping m in Figure 3). We apply the mapping defined in Section 4 to extract a UML abstract grammar, and use OCL expressions to define how model elements are mapped to an abstract syntax tree.

Extracting the UML Abstract Grammar. The UML Profile for SDL constrains the meta-model defined in the UML superstructure specification to classes and associations that can be expressed in SDL. In some cases, elements are not constrained but no mapping to SDL is defined, since no semantics is associated with them. These elements can be omitted in the extracted abstract grammar, keeping it concise. In the mapping of meta-model class Transition to the non-terminal *Transition*, name and visibility of the Transition are omitted.

TRANSITION(TRANSITIONKIND, [TRIGGER], [CONSTRAINT], [ACTIVITY], VERTEX−IDENTIFIER, VERTEX−IDENTIFIER, INTEGER)

The mapping m creates an abstract syntax node of the non-terminal corresponding to the mapped meta-class. Role names of the associations are used to navigate in the meta-model, and to set the elements on the right-hand side of the syntax node. The auxiliary function *toId* maps meta-classes to identifiers as required by the non-terminal.

context TRANSITION::*m*
mk−TRANSITION(*kind*, *trigger*−>*any*(), *guard*, *effect*, *source.toId*, *target.toId*, *priority*) `--tag definition priority--`

toId: METACLASS → IDENTIFIER

The mapping from meta-models to abstract grammars naturally maps general associations to identifiers and aggregation or composition to subtrees in an abstract syntax tree (see [3]). In few cases, due to the different structure of the abstract syntax of SDL and UML, it is of advantage to map general associations in the same way as compositions. For example, in UML, *states* and *transitions* are related by general associations, while in SDL the transition is a part of the state. Because a UML transition has a unique source state, it can be mapped as a subtree of a state instead of an identifier, in the SDL fashion.

STATE(STRING, TRIGGER−**set**, TRANSITION−**set**, CONNECTIONPOINTREFERENCE−**set**, PSEUDOSTATE−**set**, STATEMACHINE−IDENTIFIER)
PSEUDOSTATE(STRING, PSEUDOSTATEKIND, TRANSITION−**set**) `--Transition-Identifier replaced by Transition--`

context STATE::*m*
mk−STATE(*name*, *deferrableTrigger*, *outgoing*, *connection*, *connectionPoint*, *submachine*)
context PSEUDOSTATE::*m*
mk−PSEUDOSTATE(*name*, *kind*, *outgoing*)

Events do not have a counterpart in the abstract syntax of SDL. Therefore, as with transitions, we place events directly inside a trigger instead of an event-identifier. The abstract meta-class *MessageEvent* is merged with the abstract meta-class *Event*, since it doesn't introduce new attributes and associations.

TRIGGER(EVENT) `--Event-Identifier replaced by Event--`

context TRIGGER::*m*
  mk−TRIGGER(*event*)

EVENT = SIGNALEVENT | CALLEVENT | CHANGEEVENT | ANYRECEIVEEVENT

SIGNALEVENT(SIGNAL−IDENTIFIER)
CALLEVENT(OPERATION−IDENTIFIER)
CHANGEEVENT(VALUESPECIFICATION)
ANYRECEIVEEVENT()

context SIGNALEVENT::*m*
  mk−SIGNALEVENT(*signal.toId*)
context CALLEVENT::*m*
  mk−CALLEVENT(*operation.toId*)
context CHANGEEVENT::*m*
  mk−CHANGEEVENT(*changeExpression*)

6.3 Transformations on the UML Abstract Grammar.

To keep the language semantics concise, SDL distinguishes between core constructs of the language, for which the semantics are given directly, and additional constructs, which are expressed through the core constructs. In the abstract grammar AS1 of SDL, which is the target of the UML Profile for SDL, these additional constructs are already eliminated. UML constructs that correspond to additional SDL constructs are therefore transformed before the mapping to SDL is performed.

The transformations can be defined on the UML meta-model, using meta-model transformations, or on the UML abstract grammar. In order to apply the techniques from Z100 Annex F, here transformations are defined on the abstract grammar, using rewrite rules on abstract syntax trees.

If the trigger event of a «Transition» Transition is an AnyReceiveEvent, the transition is expanded according to the Model in SDL 11.3 for transforming <asterisk input list> before applying the mapping that follows in this section.

The function collectTriggers computes the set of triggers for a state from the complete valid set of triggers of the enclosing class, minus transitions and deferred signals defined for the state, and minus remote procedures and remote variables. If at least one trigger exists in the set of triggers returned by collectTriggers, the set of transitions is expanded with a copy of the transition triggered by AnyReceiveEvent, but triggered by a trigger from collectTriggers. If the set of triggers is empty, the transition triggered by AnyReceiveEvent is removed.

{ pre, tany=Transition(kind,Trigger(AnyReceiveEvent()),grd,eff,src,trg,prio), rest}

=1=>
if $\exists trig \in$ collectTriggers(tany.parent) **then**
  {pre, tany, Transition(kind,collectTriggers(tany.parent).take(),grd,eff,src,trg,prio),
    rest}
else
  {pre, rest}
endif

collectTriggers(s: State): Trigger-**set**$=_{def}$
  let ag: Class = enclosingAgent(s) **in**
    validTriggers(ag) $\setminus$ (remoteProcedures(ag) $\cup$ remoteVariables(ag) $\cup$
      inputTriggers(s) $\cup$ deferredTriggers(s))
  endlet

6.4 Formalisation of UML to SDL Mapping.

In order to define the mapping from UML to SDL, we introduce a function *Mapping* from nodes of the abstract syntax tree of UML to nodes of the abstract syntax tree of SDL.

Mapping: DEFINITIONUML $\rightarrow$ DEFINITIONAS1
idToNode: IDENTIFIER $\rightarrow$ DEFINITIONUML

In the same way as in Z100 Annex F Part 2, the mapping function is a concatenation of cases. A case consists of a pattern on the left hand side, and a resulting syntax tree on the right hand side. The pattern can contain nodes of the UML abstract grammar, as well as variables, wildcards ('*'), and a **provided**-clause to constrain the matches. Additionally, we introduce the notation *var*! to express that *var* does not match *undef*. It is a shortcut for specifying $var \neq undef$ in the **provided**-clause.

The function *idToNode* provides the same functionality as the function *idToNodeAS1* in the formal semantics of SDL. It maps an identifier to the node in the abstract syntax tree that corresponds to the definition the identifier refers to. The operators **s-** and **s2-** have the same semantics as in Z100 Annex F Part 2, selecting a subnode of a specified kind from a node. For example, s.**s-**Transition-**set** selects the set of outgoing transitions from STATE s.

Case Study: Mapping UML Transitions to SDL. We provide a formalisation for the semantics of the stereotyped class «Transition» Transition by defining the mapping function to SDL AS1. Depending on the properties of the transition, it is mapped to a *Spontaneous-transition*, *Input-node*, *Continuous-signal* or *Connect-node*.

If the trigger event of a «Transition» Transition is a SignalEvent and the name of the Signal is "none" or "NONE" (case sensitive therefore excludes "None"), the Transition is mapped to a *Spontaneous-transition-node*. The effect property maps to the *Graph-node* list of the *Transition* of the *Spontaneous-transition-node*.

Transitions triggered by the signal "none" or "NONE" are *Spontaneous-transitions* with undefined *On-exception* and *Provided-expression*. Here, we define the case where the guard of the transition is undefined. Transitions with guard have a more complicated mapping and are defined below.

Mapping(
  Transition(*,Trigger(SignalEvent(signal)),undef,effect,*,target,*)
    provided signal.idToNode.**s**-String ∈ {"NONE", "none"}
  ⇒ Spontaneous-transition(undef,undef,Transition(*Mapping*(effect),
      $Mapping_{trg}$(target.idToNode)))
)

If the trigger event of a «Transition» Transition is a SignalEvent and the name of the Signal is neither "none" nor "NONE" (so it does not map to *Spontaneous-transition-node*), the Transition is mapped to an *Input-node*. The qualifiedName of the Signal maps to the *Signal-identifier* of the *Input-node* and for each ⟨attr-name⟩ in the ⟨assignment-specification⟩ (see the Notation given in UML SS 13.3.24) the qualifiedName of the attribute (with this name) of the context object owning the triggered behavior is mapped to the corresponding (by order) *Variable-identifier* of the *Input-node*. The effect property maps to the *Graph-node* list of the *Transition* of the *Input-node*.

Transitions triggered by all other kind of signals are *Input-nodes* without *Priority*, *Provided-expression* and *On-exception*. To get the *Signal-identifier* of the Signal, the second subnode of kind string is selected with **s2**-String (**s**-String selects the signal name).

Mapping(
  Transition(*,Trigger(SignalEvent(signal)),undef,effect,*,target,*)
    provided signal.idToNode.**s**-String ∉ {"NONE", "none"}
  ⇒ Input-node(undef,signal.idToNode.**s2**-String,*Mapping*(assignment-spec),
      undef, undef, Transition(*Mapping*(effect), $Mapping_{trg}$(target.idToNode)))
)

If the trigger event of a «Transition» Transition is a ChangeEvent, the transition is mapped to a *Continuous-signal*. The changeExpression maps to the *Continuous-expression* of the *Continuous-signal*. The effect property maps to the *Graph-node* list of the *Transition* of the *Continuous-signal*. The priority maps to the Priority-name.

Transitions that are triggered by a ChangeEvent are mapped to *Continuous-signals*. The changeExpression is a boolean expression that is mapped to a corresponding SDL expression.

Mapping(
  Transition(*,Trigger(ChangeEvent(changeExpression)),*,effect,*,target,priority)

⇒ Continuous-signal(undef,*Mapping*(changeExpression),*Mapping*(priority),
Transition(*Mapping*(effect), $Mapping_{trg}$(target.idToNode)))
)

If the «Transition» Transition has an empty trigger property and a non-empty guard property, the Transition is mapped to a *Continuous-signal*. The guard maps to the *Continuous-expression* of the *Continuous-signal*. The *effect* property maps to the *Graph-node* list of the *Transition* of the *Continuous-signal*. The priority maps to the *Priority-name*.

Transitions without trigger but with guard are mapped to *Continuous-signal*s. In this case, the guard defines the condition of the *Continuous-signal*.

Mapping(
Transition(*,undef,guard!,effect,*,target,priority)
⇒ Continuous-signal(undef,*Mapping*(guard),*Mapping*(priority),Transition(
Mapping(effect), $Mapping_{trg}$(target)))
)

If the «Transition» Transition has an empty trigger property and an empty guard property, the Transition is mapped to a *Connect-node*. The effect property maps to the *Graph-node* list of the *Transition* of the *Connect-node*. If the source of the Transition is a ConnectionPointReference, this maps to the *State-exit-point-name*. If the source is a State the *State-exit-point-name* should be empty.

Transitions without trigger and guard are mapped to *Connect-node*s. The exact mapping depends on the source of the transition. The informal description is imprecise with regard to how a ConnectionPointReference is mapped to a *State-exit-point-name*. In the formalisation, this is defined precisely as the name of the exit Pseudostate associated with the ConnectionPointReference.

Mapping(
Transition(*,undef,undef,effect,source,target,*)
provided source.idtoNode ∈ ConnectionPointReference
⇒ Connect-node(source.idToNode.**s2-**Pseudostate.idToNode.**s-**String,undef,
Transition(*Mapping*(effect), $Mapping_{trg}$(target)))
| Transition(*,undef,undef,effect,source,target,*)
provided source.idtoNode ∈ State
⇒ Connect-node(undef,undef,Transition(*Mapping*(effect),
$Mapping_{trg}$(target.idToNode)))
)

If a «Transition» Transition has a non-empty trigger property and non-empty guard property, the guard is mapped to the *Transition* as follows. A *Decision-node* is inserted first in the *Transition* with a *Decision-answer* with a Boolean *Range-condition* that is the *Constant-expression* true and another *Decision-answer* for false. The specification property of the guard property of the Transition maps to Decision-question of the *Decision-node*. The false *Decision-answer* has a *Transition* that is a *Dash-nextstate* without **HISTORY**. The effect property of the Transition maps to the *Graph-node* list of the *Transition* of the true *Decision-answer*.

Guards in UML and enabling conditions in SDL represent the same concept, but have incompatible semantics (see [2]). Mapping guards to SDL is therefore not straightforward, except in the case of continuous signals (see above). To express UML-style guards in SDL, the transition is modified, inserting a decision node with the guard as condition as the first action of the transition.

Mapping(
Transition(*,Trigger(SignalEvent(signal)),guard!,effect,*,target,*)
provided signal.idToNode.**s-**String ∈ {"NONE", "none"}
⇒ Spontaneous-transition(undef,undef,Transition(< >,
Decision-node(*Mapping*(guard),undef,
{ Decision-answer(Range-condition(Constant-expression(false)),
Transition(< >,Terminator(Dash-nextstate(undef)))),
Decision-answer(Range-condition(Constant-expression(true)),
Transition(*Mapping*(effect),$Mapping_{trg}$(target.idToNode))) },
undef)))
)

Mapping(
Transition(*,Trigger(SignalEvent(signal)),guard!,effect,*,target,*)
provided signal.idToNode.**s-**String ∉ {"NONE", "none"}
⇒ Input-node(undef,signal.idToNode.**s2-**String,*Mapping*(assignment-spec),
undef, undef, Transition(< >,Decision-node(*Mapping*(guard),undef,
{ Decision-answer(Range-condition(Constant-expression(false)),
Transition(< >,Terminator(Dash-nextstate(undef)))),
Decision-answer(Range-condition(Constant-expression(true)),
Transition(*Mapping*(effect),$Mapping_{trg}$(target.idToNode))) },
undef)))
)

A target property that is a State maps to a *Terminator* of the *Transition* (mapped from the effect) where this *Terminator* is a *Nextstate-node* without *Nextstate-parameters*, and where the qualifiedName of the State maps to the *State-name* of the *Nextstate-node*.
A target property that is a ConnectionPointReference maps to a *Terminator* of the *Transition* (mapped from the effect) where this *Terminator* is a *Nextstate-node* with *Nextstate-parameters*, and where the qualifiedName of the state property of the ConnectionPointReference maps to the *State-name* of the *Nextstate-node*, and the qualifiedName of the entry property Pseudostate of the ConnectionPointReference maps to *State-entry-point-name*.

Mapping of states is ambiguous. A state can either be mapped as a state owned by a statemachine, or as the target of a transition. The former is a state in SDL, the latter a terminator. To differ between these mappings, we introduce a mapping function $\mathit{Mapping}_{trg}$ to map states as targets of transitions.

$\mathit{Mapping}_{trg}$(
  State(name,*,*,*,*,*)
  ⇒ Terminator(Named-nextstate(name,undef))
  | cpr=ConnectionPointReference(PseudoState(name,*,*),*)
  ⇒ Terminator(Named-nextstate(qualifiedName(state(cpr)),
    Nextstate-parameters(< >,name)))
)

A target property that is a Pseudostate maps to the last item of the *Transition* (a *Terminator* or *Decision-node*) as defined in section 8.6..

Transition targets that are pseudostates are mapped to corresponding terminators in SDL in a straightforward manner. Pseudostates of kind choice are mapped to decision nodes. The mapping to decision nodes is more complicated, because the UML Profile for SDL encodes the decision question in the guard expressions of the outgoing transitions, as the first operand of a two operand guard. The decision question is selected from the guard of a random transition by Constraint.**s**-Expression.**s**-Expression. The second operand, selected by Constraint.**s**-Expression.**s2**-Expression, defines the range condition of a transition originating from the choice pseudostate.

$\mathit{Mapping}_{trg}$(
  PseudoState(*,deepHistory,*)
  ⇒ Terminator(Dash-nextstate(**HISTORY**))
  | Pseudostate(name,junction,*)
  ⇒ Terminator(Join-node(name))
  | Pseudostate(*,choice,outgoing)
  ⇒ Decision-node(
    Mapping(outgoing.take.**s**-Constraint.**s**-Expression.**s**-Expression),
    undef,

```
    {Decision-answer(
      Mapping(t.s-Constraint.s-Expression.s2-Expression),
      Transition(Mapping(t.s-Activity),
      Mapping_trg(t.s2-Vertex-Identifier.idToNode)
    ) | t ∈ outgoing},
    undef)
  | Pseudostate(name,exitPoint,*)
  ⇒ Terminator(Named-return-node(name))
  | Pseudostate(*,terminate,*)
  ⇒ Terminator(Stop-node())
)
```

7 Conclusions

In this report, we have presented an approach for the formalisation of the UML Profile for SDL. This approach applies techniques from the formalisation of the static semantics of SDL, for which tool support is available. As a survey, we applied the formalisation approach to the transition stereotype from the profile, which differs syntactically and semantically from SDL transitions. Successfully applying our approach to transitions indicates its feasibility for less complicated cases.

The formal definition of the transformations, mappings and meta-model constraints helps detecting errors, omissions and ambiguities in the informal specification of the Z.109 standard. This became especially apparent with the meta-model constraints, where an error and an ambiguity were detected in the five informal constraints of the transition stereotype. The formalisations of the mappings and transformations lead to a number of suggested improvements concerning ambiguities in the informal semantics.

References

[1] Grammes, Rüdiger: Formal Operations for SDL Language Profiles. In: Gotzhein, Reinhard (Hrsg.) ; Reed, Rick (Hrsg.): *SAM 2006: Language Profiles - 5th International Workshop on System Analysis and Modelling (SAM 2006), Kaiserslautern, Germany* Bd. 4320, Springer, 2006 (LNCS), S. 51–65

[2] Grammes, Rüdiger ; Gotzhein, Reinhard: Towards the Harmonisation of UML and SDL - Syntactic and Semantic Alignment - / Department of Computer Science, University of Kaiserslautern. 2003 (327/03). – Forschungsbericht

[3] Grammes, Rüdiger ; Gotzhein, Reinhard: Towards the Harmonisation of UML and SDL. In: Frutos-Escrig, David de (Hrsg.) ; Núñez, Manuel (Hrsg.): *Formal Techniques for Networked and Distributed Systems - FORTE 2004, Madrid, Spain* Bd. 3235, Springer, Januar 2004 (LNCS), S. 61–78

[4] Grammes, Rüdiger ; Gotzhein, Reinhard: SDL Profiles - Definition and Formal

Extraction / Department of Computer Science, University of Kaiserslautern. 2006 (350/06). – Forschungsbericht

[5] ITU: *Recommendation Z.100 (03/93): Specification and Description Language (SDL).* Geneva, 1993

[6] ITU: *Recommendation Z.109: SDL combined with UML.* Geneva, 2000

[7] ITU: *Recommendation Z.100 (08/02): Specification and Description Language (SDL).* Geneva, 2002

[8] ITU Study Group 10: *Draft Z.100 Annex F2 (11/00).* 2000

[9] ITU Study Group 17: *Recommendation Z.119: Guidelines for UML profile design.* Geneva, 2005

[10] ITU Study Group 17: *Recommendation Z.109: The UML Profile for SDL.* Geneva, 2006

[11] Object Management Group: *Unified Modeling Language Specification, Version 1.3.* 2000. – www.uml.org

[12] Object Management Group: *Unified Modeling Language: Superstructure, Version 2.0.* 2005. – www.uml.org

[13] Object Management Group: *Meta Object Facility (MOF) Core Specification, Version 2.0.* 2006. – www.omg.org

[14] Object Management Group: *Object Constraint Language (OCL), Version 2.0.* 2006. – www.omg.org

[15] Prinz, Andreas ; Löwis, Martin von: Generating a Compiler for SDL from the Formal Language Definition. In: Reed, Rick (Hrsg.) ; Reed, Jeanne (Hrsg.): *SDL 2003: System Design* Bd. 2708, Springer, 2003 (LNCS), S. 150–165

[16] Werner, Constantin ; Kraatz, Sebastian ; Hogrefe, Dieter: A UML Profile for Communicating Systems. In: Gotzhein, Reinhard (Hrsg.) ; Reed, Rick (Hrsg.): *SAM 2006: Language Profiles - 5th International Workshop on System Analysis and Modelling (SAM 2006), Kaiserslautern, Germany* Bd. 4320, Springer, 2006 (LNCS), S. 1–18