Z.109 Tutorial

Rick Reed

TSE Ltd

ITU Recommendation Z.109 - UML profile for SDL-2000.
Compliant UML 2 models can be mapped/used.

Rick Reed has been involved with ITU working on language standards since 1979. At that time he was working with and represented GEC Telecommunications (later GPT, then Marconi Communications, now part of Ericsson), which he had joined as student apprentice before going to university in 1967. At first the ITU involvement was with the CHILL programming language related to his responsibility for software development tools and including his own work on a compiler, but from the mid-1980's his primary area of interest in ITU languages became the Z.100 Specification and Description Language. For this language Rick initially made a major contribution to the data model of SDL-88.

In 1986 Rick became the technical project coordinator from a large EU project SPECS involving various telecommunications companies including Alcatel, France Telecom, GPT, IBM and Philips concerned with the specification and programming environment needed for communication software. This also involved collaboration with other European projects in related software areas such as the required offline support platform and the online run-time environment. It is largely as a result of this EU work that SDL-92 was created, making the language more object oriented. The SPECS project continued into 1993. During this period Rick left GPT in 1991, and formed TSE Ltd to sell his services as an independent consultant using his experience from GPT. He continued in the key role in SPECS, including acting as the main editor for a book giving an overview of the SPEC project results published by North-Holland.

In the late 1980's, as well as representing his employer at ITU, Rick took on the role of being head of the UK representation in the ITU software languages and applications Study Group 10. After forming TSE Ltd in 1991, he continued to represent the UK in this area. Subsequent to the merger of Study Group 10 and Study Group 17 in 2001, this role included the studies previously under Study Group 7: Data networks and open system communications (including the currently important topic of system security).

During 1996 to 2000 Rick managed and contributed to the development of the SDL-2000. Since then he has contributed to and edited the Z.104, Encoding of SDL data, and Z.109, SDL-2000 combined with UML. He is currently ITU rapporteur for Q.11/17:Specification and Implementation Languages.

Overview

1. Why has Z.109 been created
2. How is the Z.109 UML profile defined
3. What is in the Z.109 UML profile
4. Use of Z.109
5. Conclusion

This tutorial assumes familiarity with:

> OMG (Object Management Group) Unified Modeling Language (UML) at least to the level of Class diagrams, Object diagrams, and State Machine diagrams;
>
> The ITU-T Specification and Description Language at least to the level of System diagrams, Block diagrams and Process diagrams with states with inputs for transitions leading to other states.

Z.109 is an ITU-T standard that defines how these two languages should be used together, by defining a UML profile that maps to SDL-2000 semantics so that UML can be used in combination with SDL. The current Z.109 (06/07) SDL-2000 combined with UML, replaces Z.109 (11/99) SDL combined with UML. The 1999 version was based on UML1.4, and the 2007 version the profile was completely rewritten starting from UML2.0, which was adopted by OMG in 2004 (and finalized in mid-2005). However, the OMG had been working on UML2.0 for some time, and work on the profile started in 2002. A work item was raised at ETSI in late 2002 on a UML profile for Communicating Systems sufficiently related to the ITU profile that a revised Z.109 could be based on ETSI results. Work was done within a voluntary ETSI task force but progress was slow due to a number of factors, several of which (such as illness and participant company reorganizations) unrelated to the objectives, and a separate ITU-T expert group started drafting the revised Z.109. In mid-2005 it was agreed that the ETSI and ITU-T experts would join forces to produce a single profile document to replace Z.109. Other experts from the SDL Forum membership contributed and the work was largely completed by the end of 2006. At the April 2007 ITU meeting Z.109 was consented for the approval procedure with formal approval June 2007.

The objectives of this tutorial is to explain why there is a need for a UML profile for SDL-2000, to explain the UML profiling technique used in Z.109, give a walk through Z.109, explain how Z.109 will probably be used, and finally to consider where to go from here. The main part is the walk through the Z.109 profile.

Why do we need profiles?

- ITU-T System Design Languages & methodology
  - ASN.1, MSC-2000, SDL-2000, TTCN-3, *URN, eODL, GDMO*
- The completeness of UML
- Component engineering

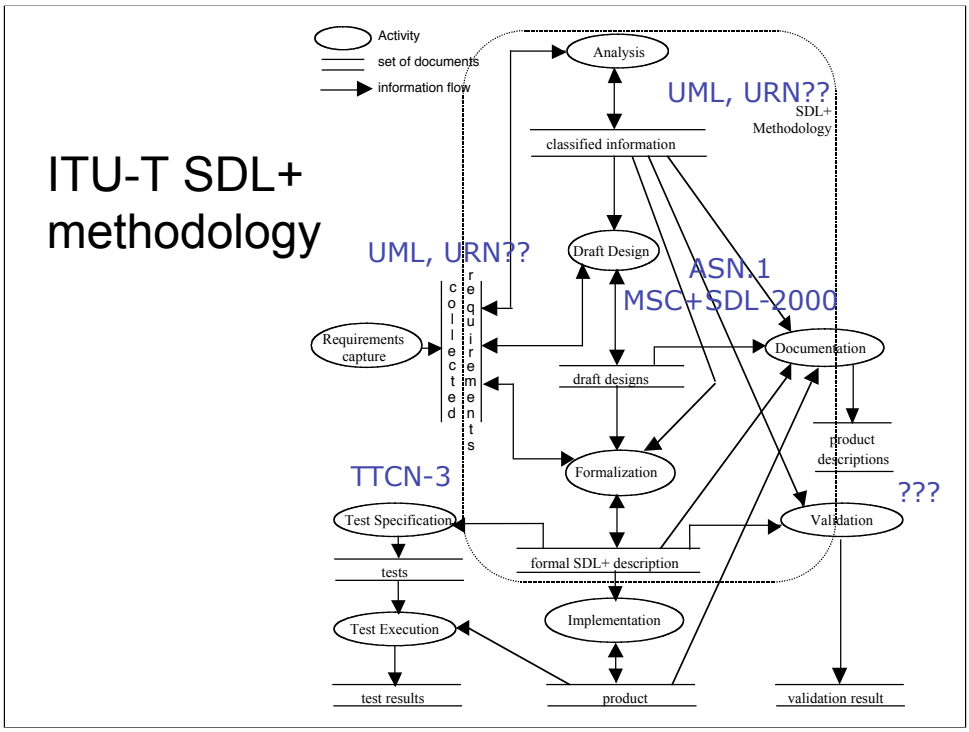
The ITU-T has defined a number of System Design Languages that are used for engineering, the main ones being :

ASN.1	for defining data items to be communicated;
MSC-2000	to describe interaction sequences;
SDL-2000	to specify and describe how objects behave;
TTCN-3	to specify and implement 'black box' tests.

These languages have been used together to engineer systems for some time, but to a large extent these languages have been developed separately. Although the use of ASN.1 with SDL-2000 and TTCN-3 is defined, there is (for example) no direct link with MSC-2000. In reality links between the languages are defined by the tool sets used by engineers or by custom engineering on individual projects. If each of the System Design Languages is expressed as a UML profile, UML (and/or the metamodels) can be used to integrate descriptions in the different System Design Languages.

Why not just use UML? The ITU-T System Design Languages have a long history and the languages and associated tools have been evolving for four decades and are therefore individually well suited to the application areas. There is a legacy of investment in these languages both in terms of experience and products. Moreover (as will be expanded later) UML2.0 is not complete and needs to be profiled in some way before it is usable.

The profile for SDL-2000 enables components written using UML to be integrated with components written in SDL-2000. It also provides a basis for the conversion of UML model components into SDL-2000 model components that can then be further elaborated as implementations.



Z.100 Supplement 1 gives a framework methodology for the use of ASN.1, MSC, SDL and TTCN. The general framework of activities is shown in the diagram above.

The SDL+ methodology is not comprehensive: it does not (for example) cover deployment issues, and is a little out-of-date because all the languages have evolved since its publication in 1995. At the time UML did not exist, but the methodology suggests to use OMT in the requirements collection phase. OMT was one of the main object oriented notations that was *unified* in UML. The Class and Object diagrams of UML are probably its most widely known and used feature. These are based on OMT.

The methodology assumes that the **collected requirements** are converted into some **classified information** before conversion into SDL+ (ASN.1 + MSC-2000 + SDL-2000) as a **draft design** or **formal design**. With today's tools and notations the **classified information** would probably be UML possibly supplemented by URN. The draft design adds information and if the SDL+ languages have UML profiles the linking or converting classified information to the draft design does not require a major paradigm shift or recoding of material (thus avoiding a source of error).

The SDL+ methodology highlights some the co-ordination issues between ASN.1, MSC-2000, SDL-2000 and TTCN-3. For example, ASN.1 enables sets of values to be defined, but does not define any operations so that expressions such as x+1 do not have any meaning in ASN.1. When ASN.1 is used with SDL, some operations are implicitly defined (for example giving meaning to x+1), but further issues arise with compatibility of values, adding operators that are not implicitly defined and inheritance. MSC-2000 and TTCN-3 have some diagrams that are very similar, but currently this languages have no common basis. UML profiles for these two languages would make it easier to determine if the meanings of these diagrams are the same or different.

Note: the implementation language (usually C, C++, Java …) is largely irrelevant these days because most code is generated automatically from the **formal design**.

UML2 completeness

To use the UML2 Superstructure needs

- The notation to be fully defined;
- Binding of notation to the metamodel;
- Binding of semantic variation points.

  A “Semantic Variation Point” section explicitly identifies the areas where the semantics are intentionally under specified to provide leeway for domain-specific refinements of the general UML semantics (e.g., by using stereotypes and profiles).

UML2 has various levels of compliance

The current OMG formal document for UML2 is Unified Modeling Language: Superstructure version 2.1.1 formal/2007-02-05 - February 2007. The Z.109 profile references this document. A version 2.2 is being progressed.

The specification of notation in UML2 is mixed.

For some constructs that are specified textually a specific BNF description is given. But for much text that appears in a UML diagram will be actions or expressions in some specific language supported by a tool and there are many instances of the phrase “No specific notation” in the UML2 document. BNF is not always used, even when it could be.

To provide a useful language for executable models, actions and expressions are essential, and so extra ‘language specific’ notation needs to be introduced. The constructs that are introduced need to be bound to the corresponding UML2 metamodel elements.

The binding of notation to the metamodel is not only needed for introduced notation, but also for notation that is defined for UML2. Much of this is not defined explicitly, either for the BNF or graphical syntax.

There is no equivalent of the BNF for graphical notation (that is no graphical meta grammar). Instead graphical notation is given by example. Like the BNF, the binding to the metamodel elements is often not defined explicitly, but examples do not have element names that can imply binding to elements with similar names.

UML2 has a number of semantic variation points, which are deliberately undecided.

Finally there are different degrees of compliance: there are four levels with increasing language coverage and at each level there is abstract syntax: concrete syntax and diagram interchange compliance. Portability from one tool to another is not certain.

UML2 BNF (example)

<multiplicity-range> ::= [<lower> '..'] <upper>
<lower> ::= <integer> | <value-specification>
<upper> ::= '*' | <value-specification>

But <value-specification> is not defined.

Could be Expression | OpaqueExpression

Expression:
"special notations permitted, including infix"

OpaqueExpression:
"text strings in particular languages"

The UML2.1.1 defines the notation for expression and opaque expression using the following two paragraphs (respectively):

> By default an expression with no operands is notated simply by its symbol, with no quotes. An expression with operands is notated by its symbol, followed by round parentheses containing its operands in order. In particular contexts special notations may be permitted, including infix operators.
>
> An opaque expression is displayed as text strings in particular languages. The syntax of the strings are the responsibility of a tool and linguistic analyzers for the languages.

So for an expression the bracketed notation is not specified syntactically and other notations are allowed, and for an opaque expression no limitation or bracketing is given for the text string which would cause syntactic ambiguities (for example if a <lower> contained '..').

The linking of the BNF syntax elements to the UML2 metamodel elements is usually left to intuition. In the case of the multiplicity range of a multiplicity element, the <lower> and <upper> are obviously related to lower attribute and upper attribute. However, it turns out that <lower> and <upper> are actually the denotations for the associated lower and upper value specifications and the respective attributes are derived.

In this particular example, it is not too difficult to work out what the relationship between the BNF notation and the metamodel elements should be. In other cases it is less obvious, and in general for a particular concrete language used for expressions and actions the binding needs to be defined.

Presentation Options

UML2: *Concrete syntax compliance does not require compliance to any presentation options*

- Tools may omit or use by default
- Portability assured by XMI support
- Tools often have other presentations
- Recognizably the same model?

There is no requirement to comply with presentation options. Some tools may implement the defined options by default or at a user option, where other tools may do things in a different (maybe very different way). The tool can still claim compliance to the abstract syntax of UML.

If two tools comply to the same level of abstract syntax, it should be possible to move the model from one tool to another by an information transfer. The correctly claim compliance to the abstract syntax requires a tool can input and output XMI for that level.

Similarly if two tool comply with the concrete syntax of UML at a level, they should be able to interchange XMI. But this only applies to any basic defined concrete syntax, not to presentation options and (as noted before) there are many areas where the concrete syntax is poorly defined, or where the is "*no specific notation*".

The reality is that there is strong incentive for a tool maker to be able read any XMI output by another tool maker, but very little incentive (unless pressed by customers) to provide good XMI from the tool that can be read by a competing tool. It is therefore no surprise that the track record for XMI is not very good.

Even if it is possible to transfer a model from one UML tool to another UML tool, there is still a question of whether the users will accept the transported model. Experience from the ITU languages is that for user acceptance the criteria is the presentation should be "recognizably the same". Some variations in items such a fonts, icons, symbol shapes will probably be tolerated, but if information is collapsed and hidden or the layout or paging is changed users might not accept the presentation model as "the same", despite its formal meaning being identical.

UML2 semantic variation

Some examples of the many semantic variations:

- Compatibility of Redefined & Redefining elements
- Determining method invoked by call operation
- Ordering of the events in the input pool

Many (not all) associated with action semantics

The variation points are resolved by:

- Using a particular tool (in a particular configuration)
- Applying a profile that binds the variations

Execution requires the variations to be bound

In the three examples given:

RedefinableElement: *Various degrees of compatibility between the redefined element* and the redefining element is concerned with the static semantics;

CallOperationAction: *The mechanism for determining the method to be invoked as a result of a call operation is unspecified*, and could be required to be static or could depend on dynamic semantics.

BehaviouredClassifier: *The ordering of the events in the input pool is a semantic variation*, which is concerned with the dynamic semantics.

UML2 explains executables from different tools behave differently:

> *In case of tools that generate program code from models or those that are capable of executing models, it is also useful to understand the level of support for the run-time semantics described in the various "Semantics" subsections of the specification. However, the presence of numerous variation points in these semantics (and the fact that they are defined informally using natural language), make it impractical to define this as a formal compliance type, since the number of possible combinations is very large.*

UML2 has to have the semantic variations bound to produce a language for executable models or implementations. A tool that binds the points, implements a language. A profile that binds the points, defines a language and a tool that supports that profile implements the language.

There are numerous semantic variation points throughout the UML2 Superstructure document at all levels of the language. It would be tedious to bind each semantic variation when the feature including it is used, so it naturally leads to the situation where profiles are used to bind some (or all) points before use.

UML action language

- Concrete syntax from outside UML
- Binding defines how objects behave
- Libraries from ‘host’ language
- Executable UML and SDL-2000
- Z.109 = SDL-2000 action semantics

UML2 does not define either the complete concrete syntax or the dynamic semantics for actions. To a large extent the concrete syntax is left to be instantiated by some ‘host’ action language (typically C, C++, Java or some domain specific implementation language).

Although UML2 constrains some of the dynamic semantics for actions, it also leaves many semantic variation points to be bound before a model can be executed. The intention has always been that the UML2 Superstructure can be stereotyped and profiled to a language of choice.

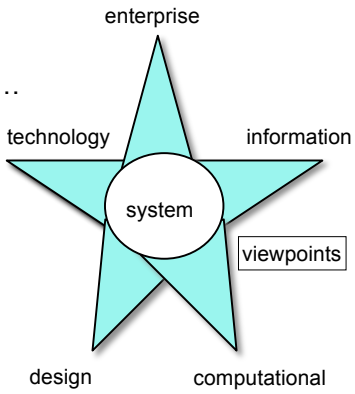
A language of choice usually includes a library of predefined data types, operations and methods. Although these do not change the theoretical basis of the language for actions, in practice the libraries available have a major impact on how the language is used and become part of the language: particularly where these are part of a standard, such as the *Predefined* package of SDL-2000.

The history of software engineering is of raising in levels of abstraction. UML that is Executable is the next evolutionary step. Rather than translate an analysis product into a design product and then write code, application developers can (with suitable tools) use the same ‘language’ to model the abstract application and elaborate it into a delivered product. If the action language is (for example) SDL-2000, it gives UML this capability, while maintaining a actions sufficiently abstract so that they can be emulated and validated with a virtual (SDL-2000) machine, or translated to C, C++ or Java (as an intermediate language - seldom looked at), or to the machine code of the target system.

The Z.109 profile provides SDL-2000 action semantics to UML, therefore acting as a requirement specification for such a tool.

Subsets of UML2

- UML2 - 13 diagrams types
  - Structure(6): Class, Package ...
  - Behavior(7): Sequence, State-machine …
- Give different views, overlap
- Most used/supported
  - Use-case, Sequence, Class, Package, Composite-structure, State-machine
- Executable models (Z.109)
  - Class, Package, Composite-structure, State-machine



UML2 has 13 diagram types:

> Class, Object, Package, Composite-structure, State-machine, Sequence, Use-case, Communication, Interaction-overview, Activity, Component, Deployment, Timing

Each of these is a view on a system, and in a tool they should give different views of a single model of the system in a repository.

Past work has suggested five views of systems: enterprise, information, computational design and technology. Though this is a bit simplistic, it does suggest that different views are required at different stages during the engineering of a product. As illustrated, this does not mean that different underlying models are required, or that different notation are necessarily needed for different models.

Because the UML diagrams are different views of the same model, there tends to be some redundancy of information between the various different diagrams. For example, Sequence and communication diagrams both show the order in which messages can occur. Often the preference of one diagram type over another depends on the background of the user or the tools available.

UML2 adds very little to what can be expressed in ITU-T System Design Languages, but has the advantages of integration and (some of it) being more widely known.

On the other hand, each of the ITU-T languages has a more limited domain of application than the complete range of UML diagrams. A profile for an ITU-T language, therefore includes only those parts of UML with diagrams concerned with the same domain. For example, a profile for MSC-2000 would be concerned only with sequence diagrams and the necessary other diagrams required to define items used in the sequence diagrams.

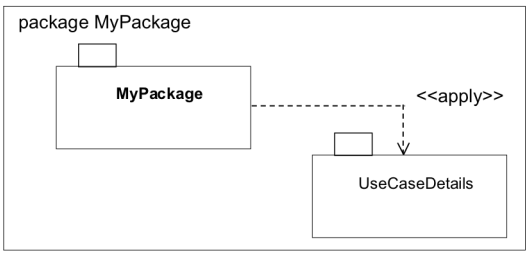
Guidelines UML profile design Z.119

A UML profile defines

- Extra info associated with model elements
- Additional concepts (based on UML ones)
- Constraints on UML (limit model to map)

by a «profile» package with stereotypes

- Tagged values (attributes of stereotype)
- ‘extend’ an existing metaclass



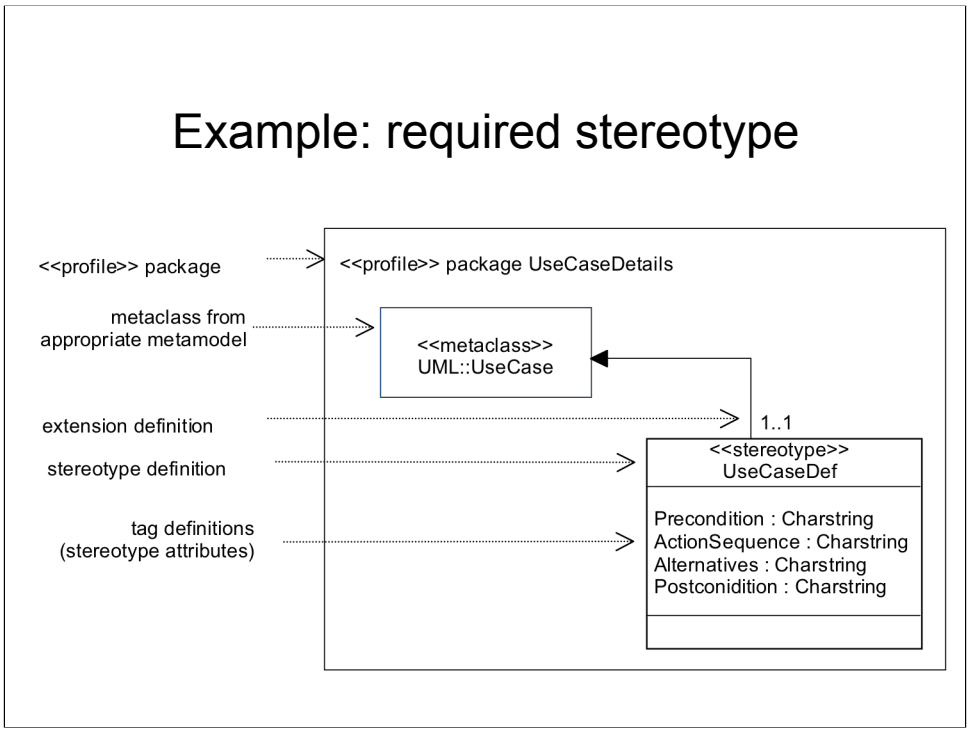
The purpose of a UML profile is essentially the following:

- To define how to associate extra information with model elements in a UML model;
- To define additional concepts that are not defined by UML, but that can be based on the existing UML concepts;
- To constrain UML so that the model can be mapped to the ITU T Language.

From a pragmatic point of view a profile is simply a UML package stereotyped with keyword «profile». The «profile» package contains stereotypes that define:

- Tagged values (the attributes of the stereotype);
- What model element to extend (the ‘extends’ relation to a metaclass).

Formally in UML a profile is applied to a package using a dashed arrow notation with the keyword 'apply' within guillemots as in the example where MyPackage has the «profile» package UseCaseDetails. However, tools usually supply simpler ways of applying profiles, such as the user selecting a name in a user interface.



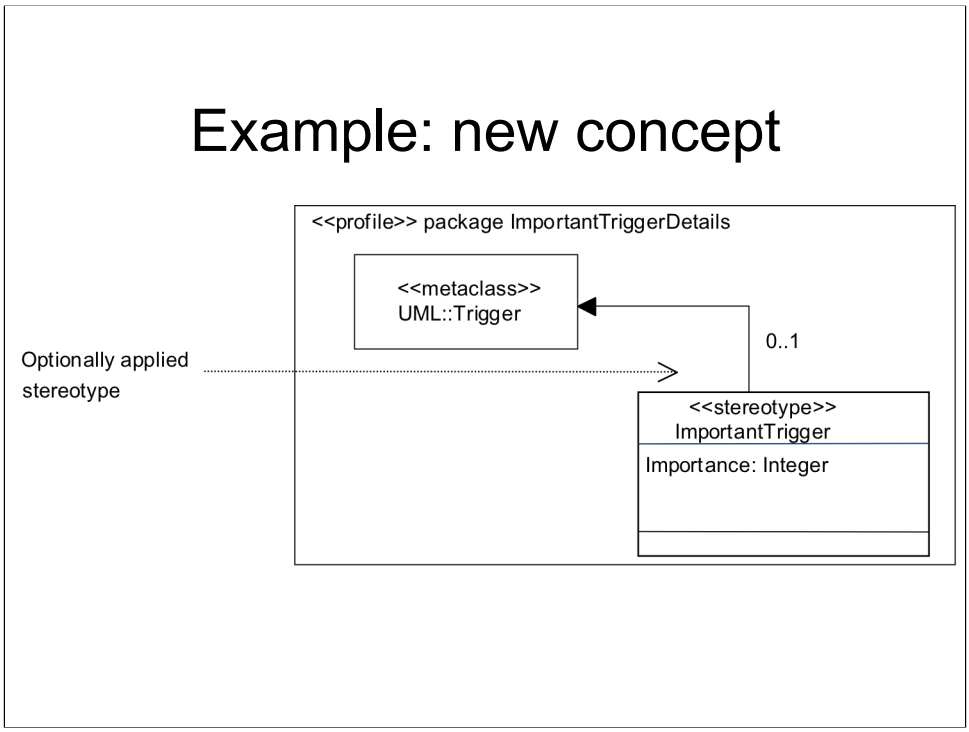
In this example a stereotype called UseCaseDef is defined that extends the built-in UML concept UseCase. The purpose of this is to add some extra items to all use cases (the concept defined by UseCase). In this case the extras are Precondition, ActionSequence, Alternatives and Postcondition attributes of the stereotype.

The stereotype extends the metaclass with multiplicity [1..1]. This is essential because it will enforce that (from a users point of view) whenever a use case is created in a package where the profile is applied, the use case will have the specified properties. Moreover, if the stereotype has the same name as the metaclass (in this case UseCase instead of UseCaseDef), in application models the name UseCase always refers to the extended definition.

This is the mechanism that should be used in ITU T Language profiles whenever there is a need to add extra information to the built-in UML concepts. It is the approach taken in Z.109.

Occasionally the same UML concept has two (or more) alternative extensions, one of which should be applied. In this case, the original UML concept should be extended with the stereotype multiplicity [1..1] with the same name as the original concept, and this stereotype is extended with the alternative extensions with multiplicity [0..1] with different names. The stereotype of the original UML concept shall have an additional constraint that every extension shall have the constraints and properties of one (and only one) of the alternative extensions

Example: new concept



This example introduces a new concept called "ImportantTrigger".

From a user's point of view it is visible as a stereotype that optionally is applied to triggers. The key point to notice is that the multiplicity of the extends relation is "0..1". This means that in some cases a Trigger will be an ImportantTrigger, but not every case.

Example Stereotype Definition

7.12 Property

The stereotype Property extends the metaclass Property with multiplicity [1..1].

NOTE A property is an attribute that corresponds to variables and agent instance sets in SDL, or fields of a structure.

7.12.1 Attributes

Stereotype attributes:

- initialNumber: UnlimitedNatural [0..1] defines the initial number of instances …

-referenceSort: Boolean determines the treatment of a variable or field as …

7.12.2 Constraints

- The aggregation shall not be shared.
- If a <<Property>> Property has aggregation that is composite, …

7.12.3 Semantics

If isreadOnly is false and has an aggregationKind that is none and type is a <<PassiveClass>> Class … the <<Property>> Property is mapped to a *Variable-definition*.

…

7.12.4 Notation

UML standard syntax is used with the following extensions. The property type <prop type> shall …

7.12.5 References

SDL: 9 Agents

12.3.1 Variable definition

…

UML SS:

7.3.32 MultiplicityElement

7.3.44 Property

stereotype Property - names the Property Stereotype

multiplicity [1..1] - all Property items shall match the stereotype Property

initialNumber, referenceSort - attributes introduced in the stereotype

<<Property>> Property - a UML Property stereotyped with <<Property>>

<<PassiveClass>> Class - a UML Class stereotyped with <<PassiveClass>>

aggregation - an attribute (of a stereotype or a UML metaclass)

Variable-definition - Z.100 Abstract syntax element

NOTE: In main section of a stereotype: Informal definition of the purpose of the stereotype

Attributes - attributes for extra information that is needed to capture the Z.100 concept, or 'No additional attributes'.

Constraints - that are needed for the stereotype so that the UML model will be well formed with respect to the Z.100 concept it is intended to capture. For example, shared aggregation has no equivalent in SDL so is forbidden.

Semantics - Usually only specifies the mapping to Z.100 elements, and how the element behaves is defined by Z.100 series standard. The mapping is often complex, so that the mapping chosen may depend on attributes and relationships to other values. One UML element, may map to more than one Z.100 element.

Notation - where UML already contains a sufficient notation 'UML standard syntax is used', otherwise for text BNF is given or if graphical symbols are needed these are described. How concrete syntax relates to the metamodel is defined.

References - the corresponding paragraphs of the UML2 Superstructure and

UML2 coverage

Included UML2 chapters
- Classes
- Composite Structures
- Common Behaviors
- Actions
- Activities
- State Machines

Stereotyped elements

Non-empty properties

Excluded UML2 chapters
- Components
- Deployments
- Use Cases
- Interactions
- Auxiliary Constructs
- Profiles

Notation is guideline only

Meta-model elements defined in included chapters are included if they are specifically mentioned in Z.109.

Any meta-model element of the UML Superstructure specification that is not mentioned in Z.109 is not included in the profile.

A meta-model element that is generalization of one of the included meta-model elements (that is, it is inherited) is included as part of the definition of the included meta-model element. Other specializations of such a generalization are only included if they are also specifically mentioned.

If an included meta-model element has a property that is allowed to be non-empty, the meta-model element for the property is included. However, if the property is constrained so that it is always empty, such a property is effectively deleted from the model and therefore does not imply the meta-model element for the property is included.

Names and name resolution

A name maps to
- a *Name* in definition context;
- an *Identifier* in use context.

Sometimes the qualifiedName is needed (<identifier>)

```
<name> ::=
      <underline>+ <word> {<underline>+ <word>}* <underline>*
   | <word> <underline>+
            [ <word>{<underline>+ <word>}* <underline>* ]
   | <decimal digit>* <letter> <alphanumeric>*
<word> ::=
   <alphanumeric>+
<identifier> ::=
   [ <containing namespaces> ] <name>
<containing namespaces> ::=
   [<name separator>] { <name> <name separator> }+
<name separator> ::=
   <colon> <colon>
```

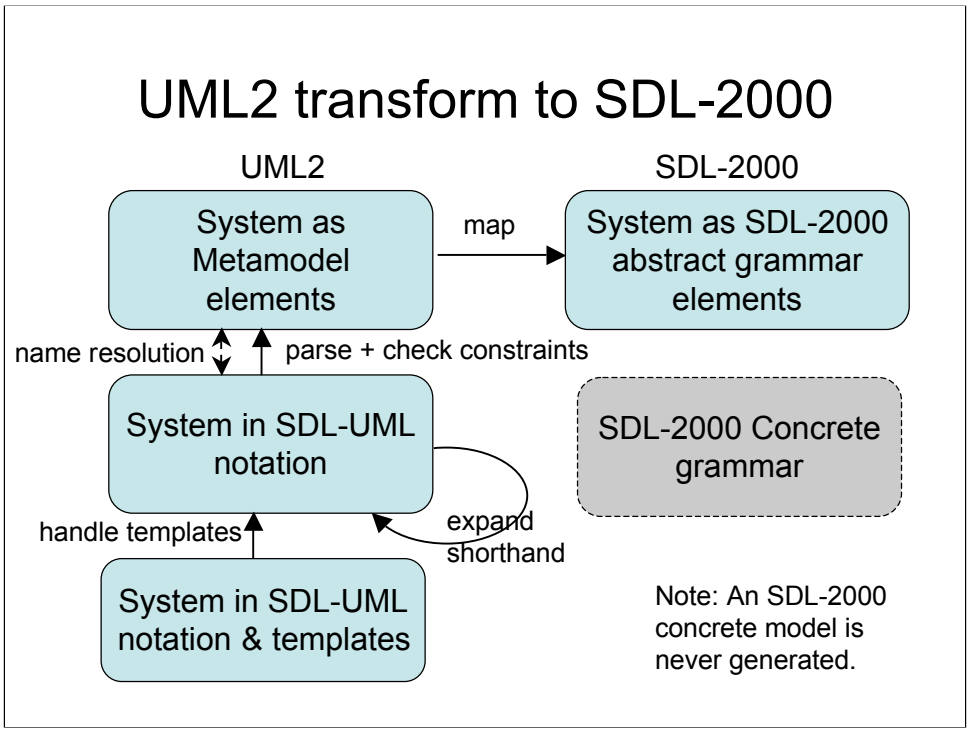
A SDL-UML name must contain at least one underline or at least one letter. It is not allowed to contain full stops. This enables it to be distinguished from an integer or real number.

The <name separator> is a lexical unit, so that <colon> <colon> is always treated as <name separator> outside the lexical context.

Any item that inherits from NamedElement and maps to SDL abstract syntax requiring a *Name* (usually a definition context) shall have a name. Any such name shall have a non-empty String value of characters derived from the syntax. No item shall have the same *Name* as another item of the same entity kind in the same defining context.

Whenever a *Name* is required in the SDL abstract syntax (usually for the definition of an item), the *Name* is mapped from the name of the appropriate item derived from NamedElement. Whenever an *Identifier* is required in the SDL abstract syntax (usually to identify to a defined item), the *Identifier* is mapped from the name of the appropriate item derived from NamedElement.

When a <name> occurs in syntax that defines a name, the qualifiedName is derived from the defining context. Otherwise, a name shall be bound according the UML name binding rules and if necessary the name is qualified by containing namespaces.



The language defined by the profile is called SDL-UML.

Template parameters are expanded according to UML expansion rules before application of the profile. For example, ownedTemplateSignature, templateBinding, owningParameter and templateParameter are expanded.

The concrete syntax of a system is parsed according to the SDL-UML concrete grammar. Where the concrete grammar defines shorthand notations, these are expanded during the parsing process before the corresponding meta-model items are generated.

Names are resolved according the SDL-UML meta-model. The parsing of the concrete grammar is therefore be done in parallel with parsing the concrete syntax and generating the meta-model. If the concrete model does not conform to the concrete grammar (syntax and static conditions for the concrete syntax, including the use of names) of SDL-UML, the model is not valid.

The meta model is generated from the concrete model according to the relationship between the concrete grammar and the meta model. If the model expressed as meta model elements does not conform to the abstract grammar (meta-classes, associations and constraints) of SDL-UML, the model is not valid. Conforming to the meta-model rules of SDL-UML is a necessary (but not sufficient) condition for a model to be a valid model.

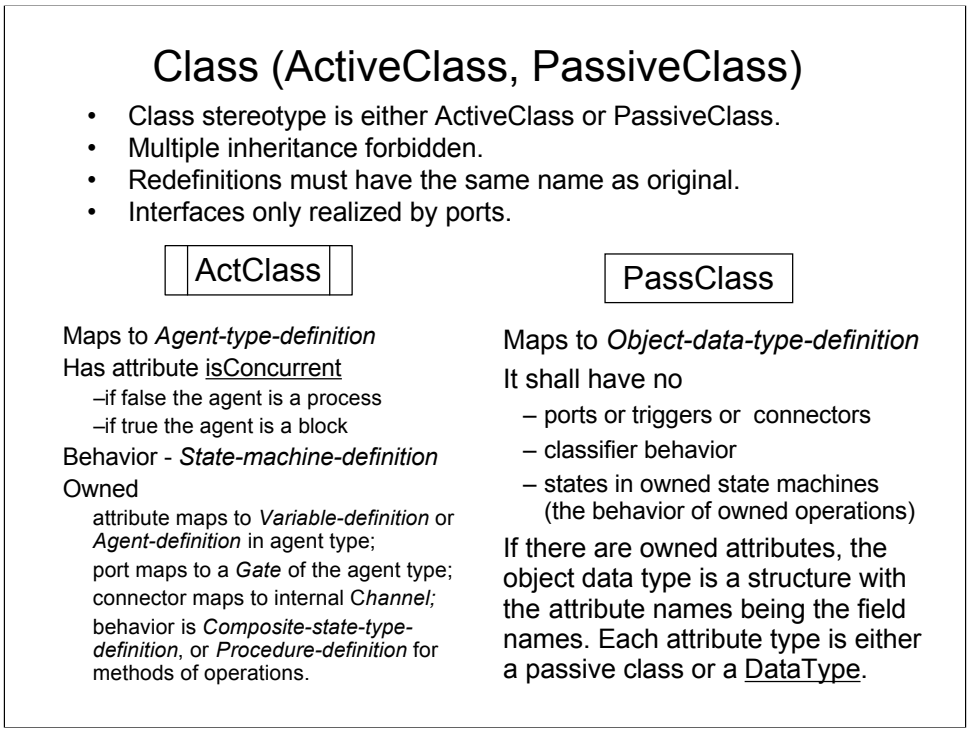
The model expressed as SDL-UML meta-model elements is mapped to a model in the abstract grammar of SDL-2000. How the system behaves is determined by the semantics defined for the SDL abstract grammar. The static conditions of SDL-2000 are reflected in the constraints of the SDL-UML meta-model. However, if during interpretation of the model expressed in the abstract grammar of SDL-2000, any dynamic condition of SDL-2000 is not met, the model is not valid.

Structure

UML2 packages	Metaclasses from UML2
- Communications - Constructs (from Infrastructure library) - Dependencies - Interfaces - InternalStructures - Kernel - Ports - PrimitiveTypes	- Class - Connector - DataType - Enumeration - Interface - Operation - Package - Port - PrimitiveType - Property - Signal - Timer

The structure of a system is defined mainly in class and composite structure diagrams.

Communication is by signals passed from a port on one element to a port on another element where the ports are joined by a connector.



The concept of an active class (a class with isActive true) is separated from passive class (a class with isActive false) to distinguish the classes for executable agents that map onto SDL agent types.

ActiveClass

An ownedAttribute that is visible outside the <<ActiveClass>> Class (public visibility) and that has a type that is a DataType or <<PassiveClass>> Class is the *Variable-definition* for an exported variable and also maps to an implicit *Signal-definition* pair for accessing this exported variable in the defining context of the *Agent-type-definition*.

If the <<ActiveClass>> Class has a classifierBehavior, it shall be a StateMachine. The StateMachine that is the Behavior of the optional classifierBehavior maps to the *State-machine-definition* of the *Agent-type-definition*. The name of the optional classifierBehavior is mapped to the *State-name* of the *State-machine-definition*. The *Composite-state-type-identifier* of this *State-machine-definition* identifies the *Composite-state-type* derived from the StateMachine that is the classifierBehavior. The UML StateMachine maps to the behaviour of an SDL composite state type, and the *State-machine-definition* references this behaviour.

Each Behavior of the ownedBehavior maps to an element of either the *Composite-state-type-definition-**set*** or the *Procedure-definition-**set***. If the owned Behavior is the method of an Operation, it is an element of the *Procedure-definition-**set***, otherwise it is an element of the *Composite-state-type-definition-**set***.

PassiveClass

An ownedBehavior maps to a *Procedure-definition* in the *Procedure-definition-**set*** in the nearest enclosing scope that contains the *Object-data-type-definition.*

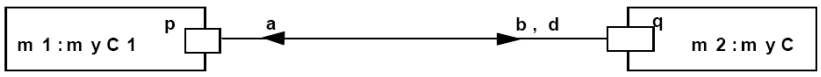
Connector

Maps to a *Channel-definition*.

There shall be an associated InformationFlow, from which the *Signal-identifier-set* of the *Channel-path* is derived.

A connector shall have 2 end properties that end in ports, which are the channel gates.

Has a delay attribute.



The conveyed InformationItem set of each InformationFlow defines the *Signal-identifier-**set*** of the *Channel-path*. If the InformationItem set is omitted then the *Signal-identifier-**set*** is computed based on the realized and required interface of the attached Port. If the InformationFlow conveys an Interface then the *Signal-identifier-**set*** is computed according to the transformation rules of Z.100 (see Interface section).

The role of a ConnectorEnd that is part of the end property maps to an *Originating-gate* or *Destination-gate* in each *Channel-path*. If the role corresponds to the source of the InformationFlow for the *Channel-path* the role maps to an *Originating-gate* otherwise it maps to a *Destination-gate*. The *Gate-identifier* is derived from the name of the Port given by the role.

If the partWithPort is non-empty, *Gate-identifier* contains as its last path-name (before the name of the gate) the name of the part identified with partWithPort.

DataType, PrimitiveType, Enumeration

Maps to *Value-data-type-definition*.

ownedAttribute shall have a PassiveClass or DataType.

If a DataType has an ownedAttribute it is a structure:

Each owned attribute name, names a field.

An ownedOperation maps to a *Static-operation-signature*.

An ownedBehaviour maps to a *Procedure-definition*

in the scope enclosing the *Value-data-type-definition*.

If it is an Enumeration

each owned literal defines a *Literal-signature;*

each owned operation a *Static-operation-signature*.

If it is a PrimitiveType maps to a predefined data type.

A <<DataType>> Datatype is a PrimitiveType (which captures the non-parameterized predefined data types of SDL) or an Enumeration (which corresponds to types defined by a set of literal names) or a value type (typically a structured data type, but could be simply a collection of operations). If it is a value type with at least one ownedAttribute, it is a value structure type (see also the definition of an object structure type by a <<PassiveClass>> Class with an ownedAttribute set that is not empty). A value type without an ownedAttribute that is neither a PrimitiveType nor an Enumeration is a collection of operations. If some of these operations have a result of the type, these denote values of the type. For example, the basis of a type for imaginary numbers could be a type called Imaginary with a operation

`makeImaginary(Integer,Integer)->Imaginary`

together with other appropriate operations

and `makeImaginary(-1,2)` would denote a value of the type.

A data type cannot contain agent instance sets or variables. A data type with an ownedAttribute is a value structure type and each ownedAttribute is field of the structure.

The ownedOperation items are mapped to the *Static-operation-signature-**set*** of the *Value-data-type-definition*.

An ownedBehavior maps to a *Procedure-definition* in the *Procedure-definition-**set*** in the nearest enclosing scope that contains the *Value-data-type-definition*.

A <<Datatype>> Datatype with an ownedAttribute set that is not empty represents a structure and each ownedAttribute represents a field. An ownedAttribute maps to operations in the *Static-operation-signature-**set*** in the SDL abstract syntax for the field operations. These operations are determined and corresponding items implied in the SDL-UML model in the same way as the field operations for a <<PassiveClass>> Class with an ownedAttribute set that is not empty (see PassiveClass). If ownedOperation associations are defined, the defined operation signatures are added to the *Static-operation-signature-**set***. The contained *Data-type-definition-**set***, *Syntype-definition-**set*** and *Exception-definition-**set*** are empty.

Interface, Port, Signal (Timer)

Interface maps to *Interface-definition*.

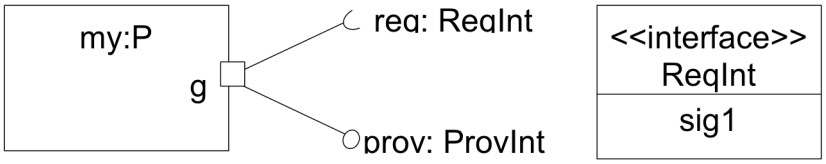
- ownedAttribute expanded as Z.100 remote variable
- ownedOperation expanded as remote procedure
- nestedClassifier shall be Signal
  - Maps to member of interface *Signal-definition-**set***

Port maps to *Gate-definition*

- requiredInterface maps *Out-signal-identifier-**set***
- providedInterface maps *In-signal-identifier-**set***

Signal maps to *Signal-definition*

- ownedAttribute corresponding *Sort-reference-identifier*
- Timer is Signal subtype with default Duration
  - Can be be used in Set, Reset, Active



An interface defines public features that are used to communicate with an object. In SDL-UML these are signals, remote variables and remotes procedures. Accesses to remote variables and calls of remote procedures are signal exchanges in the SDL abstract grammar, so the components of a SDL-UML interface map to signals in the corresponding *Interface-definition*.

An SDL-UML port defines an SDL *Gate*. The required interfaces characterize the requests from the classifier to its environment through the port and therefore define the outgoing signals for the *Gate*. The provided interfaces of a port characterize requests to the classifier that are permitted through the port and therefore define the incoming signals for the *Gate*.

A signal represents the type for communication message instances and maps to a *Signal-definition*. A timer is a specialized signal, with an attribute for the default value of the time and a mapping to an SDL Timer that gives additional behaviour. A timer can also be used in expressions that require a timer.

Operation

In an Interface expands to signals (see Interface).

In an ActiveClass maps to *Procedure-definition*.

In a PassiveClass or Datatype is a operator and maps to:

- *Operation-signature* and identified anonymous *Procedure-definition* in scope of type

Each *Procedure-formal-parameter* and *Result* of the *Procedure-definition* are derived in an obvious way from each ownedParameter including its direction (in, inout, out, return) and the type of the Operation.

The Behavior identified by the method property defines the *Procedure-graph*, *Data-type-definition-**set***, and *Variable-definition-**set*** of the *Procedure-definition*.

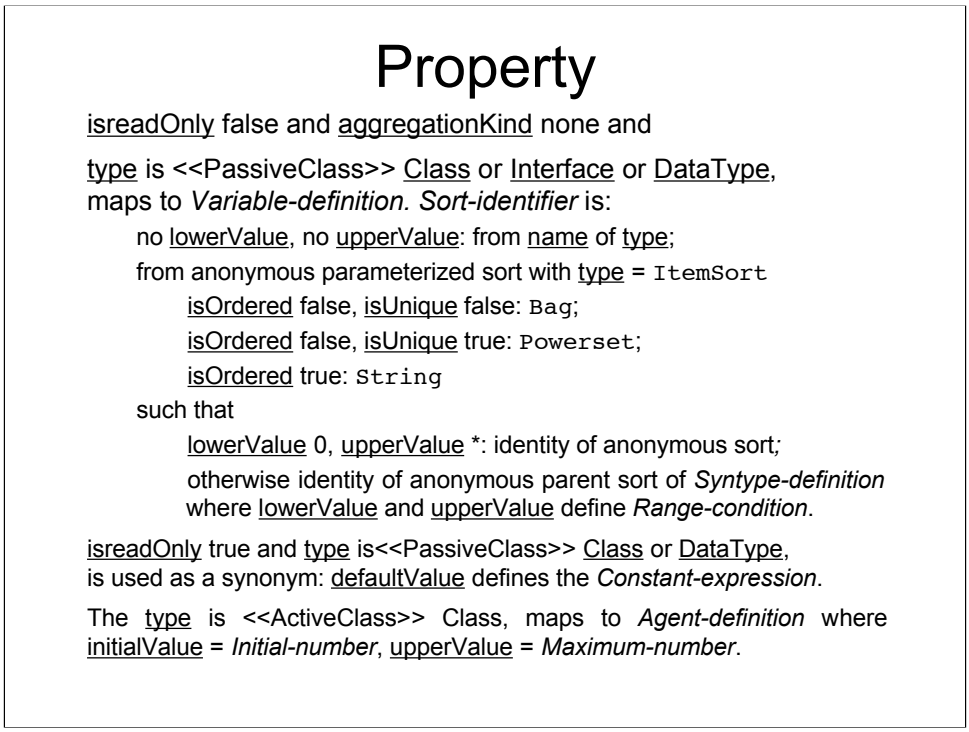
The properties isQuery, bodyCondition, precondition and postcondition are ignored.

An operation is a feature that determines how an object behaves as described by its method. If the operation is contained in an agent (that is an <<ActiveClass>> Class), the method has to be a state machine and maps to a procedure. An operation contained in an interface is treated as a remote procedure. Otherwise, the operation has to be an activity and maps to an operation of the SDL data type for the <<PassiveClass>> Class or <<DataType>> DataType that contains the operation.

An <<Operation>> Operation directly contained in an <<ActiveClass>> Class is mapped to a *Procedure-definition*. The name defines the *Procedure-name*.

An Operation directly contained in a <<PassiveClass>> Class or DataType is mapped to an *Operation-signature* and an anonymous *Procedure-definition* identified by the *Identifier* of the *Operation-signature,* in the same context as the data type of the passive class or data type. In each ownedParameter that does not have a return direction, the type and multiplicity together define (in order of the parameters) a *Formal-argument* of the *Operation-signature* with a type determined in the same way as in a Property. The type of the Operation defines the *Result* of the *Operation-signature*.

If the Operation maps to a *Procedure-definition* (named or anonymous), each ownedParameter that does not have a return direction defines (in order) a *Procedure-formal-parameter* where the name and type (including the multiplicity) of the ownedParameter define respectively the *Variable-name* and the *Sort-reference-identifier* of the *Parameter*. The *Sort-reference-identifier* is determined in the same way as for a Property. The direction (in, inout, or out) of each ownedParameter that does not have a return direction determines (respectively) if the corresponding *Procedure-formal-parameter* is an *In-parameter* or *Inout-parameter* or *Out-parameter*. The type of the Operation defines the *Result* of the *Procedure-definition*. The Behavior identified by the method property defines the *Procedure-graph*, *Data-type-definition-**set***, and *Variable-definition-**set*** of the *Procedure-definition*.



If isreadOnly is false and has an aggregationKind that is none and type is a <<PassiveClass>> Class or an Interface or a DataType (which includes PrimitiveType and Enumeration), the <<Property>> Property is mapped to a *Variable-definition*. The name defines the *Variable-name*. The defaultValue defines the *Constant-expression*. The *Sort-reference-identifier* is the *Sort-identifier* of the sort derived from the type property. The *Sort-identifier* is determined as follows:

- If there is no upperValue and no lowerValue, the name of the type maps to the *Sort-identifier*;
- Otherwise, the *Sort-identifier* identifies an anonymous sort formed from the SDL predefined `Bag` (if isOrdered is false and isUnique is false) or `Powerset` (if isOrdered is false and isUnique is true) or `String` (if isOrdered is true) datatype instantiated with the sort given by the type as the `ItemSort`. The anonymous sort is a *Value-data-type-definition* or *Syntype-definition* in same context as the *Variable-definition*. If the upperValue value is omitted or the lowerValue value is zero and the upperValue value is unlimited (* in the concrete syntax), there are no size constraints and the anonymous sort is a *Value-data-type-definition* with its components derived from the instantiated predefined data type. Otherwise the lowerValue value and upperValue value map to a *Range-condition* of the anonymous sort, which is a *Syntype-definition*. The *Parent-sort-identifier* of this *Syntype-definition* is a reference to another anonymous sort that is the *Value-data-type-definition* derived in the same way as the case with no size constraints.

If isreadOnly is true, the type is required to be either a DataType (which includes PrimitiveType and Enumeration) or a <<PassiveClass>> Class. When isreadOnly is true, the Property is mapped to a *Constant-expression* defined by the defaultValue each time the Property is used in an expression.

If the type is an <<ActiveClass>> Class, the <<Property>> Property is mapped to an *Agent-definition*. The name defines the *Agent-name*. The type property defines the *Agent-type-identifier* that represents the type in the SDL abstract syntax. The initialNumber defines the *Initial-number*. The upperValue defines the *Maximum-number*. If the initialNumber is omitted the lowerValue defines the *Initial-number*. If both the initialNumber and lowerValue are omitted the *Initial-number* is 1.

Packages

A <<Package>> Package is mapped to a *Package-definition*.

A <<Package>> Package is mapped to a *Package-definition*.

The name of the package maps to the *Package-name* of the *Package-definition*.

The elements of the ownedMember composition define the contents of the package, that is the *Package-definition-**set***, *Data-type-definition-**set***, *Syntype-definition-**set***, *Signal-definition-**set***, *Agent-type-definition-**set***, *Composite-state-type-definition-**set*** and *Procedure-definition-**set***. Each ownedMember that is a nestedPackage maps to an element of the *Package-definition-**set*** of the *Package-definition*. An ownedMember that is not a nestedPackage is mapped as defined in other sections to a *Data-type-definition*, *Syntype-definition*, *Signal-definition*, *Agent-type-definition*, *Composite-state-type-definition* or *Procedure-definition* element of the corresponding set of the *Package-definition*.

The UML ElementImport and PackageImport (which are not stereotyped in this profile) define the import and visibility of elements of the package and define the name resolution of imported package elements. The resolved items map to *Name* and *Identifier* items in the SDL abstract syntax.

State Machines

Metaclasses from package BehaviorStateMachines:

FinalState	no name in StateMachine of an ActiveClass = *Stop-node*, else no name in StateMachine (not procedure) = *Action-return-node*, else non-empty name = *Named-return-node* with *State-exit-point-name*
Pseudostate	initial in region for composite state *Start-node* initial in region for procedure *Procedure-start-node* deepHistory is mapped to a *Nextstate-node* junction is mapped to a *Free-action* choice is mapped to a *Decision-node* entryPoint is mapped to a *Start-state-node* exitPoint is mapped to a *Named-return-node* terminate is mapped to a *Stop-node*
Region	of Statemachine with specification = *Procedure-graph* single of Statemachine w/o specification = *Composite-state-graph* multiple of Statemachine w/o specification = *State-partition*
State	is mapped to a *State-node*
StateMachine	w/o specification maps to *Composite-state-type-definition* with specification maps to *Procedure-graph*
Transition	- see separate description -

When a FinalState is reached the containing graph completes. In SDL-UML a graph for a procedure will complete with a <<Return>> ActivityFinalNode. In this case, there is no mapping to the SDL abstract syntax for FinalState, because the return node terminates the graph. A FinalState that is not in a procedure graph maps to an *Action-return-node* or *Named-return-node* for the enclosing composite state.

A Pseudostate is used instead of a state before initial or state entry point transitions, when there is a junction of transitions, when there is a decision to make a choice of transitions, when the transition leads to a history nextstate, or after a transition to lead to a state exit point or terminate the state graph. They allow more complex transitions between states to be built from simpler, shorter transitions that end or start (or start and end) in a Pseudostate. They map to start, next state (with history), decision, join and free action, return and stop nodes in the SDL state transition graph.

A region contains states and transitions and is mapped to the definition of how a procedure or a composite state behaves. For the composite state mapping of a StateMachine, a single region maps to a *Composite-state-graph*, whereas two or more regions map to a *State-aggregation-node*. A region in SDL-UML is always part of a StateMachine and is never part of a State, because the region of a State is constrained to be empty.

A state represents a condition where an object is waiting for some condition to be fulfilled: usually for an event to occur. A state in SDL-UML maps to an SDL state.

An SDL-UML StateMachine either maps to the graph of an SDL procedure or an SDL composite state type. The two cases are distinguished by whether or not the StateMachine has a specification. If it does then it is the procedure case, otherwise it is a composite state type. Because there are two different mappings some constraints on StateMachine are dependent on whether there is a specification or not.

Transition

With TransitionKind local expanded as * state.

AnyReceiveEvent trigger expanded as * input.

CallEvent trigger expanded as remote procedure call.

With Signal name NONE maps to *Spontaneous-transition*.

effect maps to *Graph-node* list of *Transition* of *Spontaneous-transition*

With other Signal name maps to *Input-node* where

effect maps to *Graph-node* list of *Transition* of *Input-node*

With empty trigger, empty guard maps to *Connect-node* where

effect maps to *Graph-node* list of *Transition* of *Connect-node*

With trigger and guard maps to *Decision-node* for guard

with false *Decision-answer* a *Dash-nextstate* without HISTORY and

effect maps to *Graph-node* list of *Transition* of true *Decision-answer*

Terminator of the *Transition* is in the case of a target

ConnectionPointReference: *Named-nextstate*

Pseudostate: a *Terminator* or *Decision-node* at the end of *Transition*

A transition is the part of a state transition graph that defines what happens when the object goes from one graph vertex another. Each vertex is usually a state, but may be a pseudostate. Signals are used to trigger transitions. Standard UML notation and semantics are used.

If the trigger event of a 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*. The effect property maps to the *Graph-node* list of the *Transition* of the *Spontaneous-transition*.

If the trigger event of a Transition is a SignalEvent and the name of the Signal is neither "none" nor "NONE" (so it does not map to *Spontaneous-transition*), 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> 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*.

A target that is a State maps to a *Terminator* of the *Transition* (mapped from the effect) where *Terminator* is a *Nextstate-node* that is a *Named-nextstate* without *Nextstate-parameters,* and where qualifiedName of the State maps to the *State-name* of the *Named-nextstate*.

A target that is a ConnectionPointReference maps to a *Terminator* of the *Transition* (mapped from the effect) where this *Terminator* is a *Nextstate-node* that is a *Named-nextstate* with *Nextstate-parameters*, and where the qualifiedName of the state property of the ConnectionPointReference maps to the *State-name* of the *Named-nextstate,* and the qualifiedName of the entry property Pseudostate of the ConnectionPointReference maps to *State-entry-point-name* of the *Nextstate-parameters*.

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

Actions and activities

Packages	Included metaclasses
BasicActions	Activity
BasicActivities	ActivityFinalNode
BasicBehaviors	AddStructuralFeatureValueAction
CompleteActivities	AddVariableValueAction
CompleteStructuredActivities	CallOperationAction
FundamentalActivities	CreateObjectAction
IntermediateActivities	ConditionalNode
IntermediateActions	LoopNode
StructuredActions	OpaqueAction
StructuredActivities	SendSignalAction
	SequenceNode

Describe how the model behaves: the action control flow and handling of value expressions and changing instance values.

An activity is used to describe how the model behaves, for example the control flow of actions in an operation body or a transition. When invoked, each action takes zero or more inputs, usually modifies the state of the system in some way such as a change of the values of an instance, and produces zero or more outputs. The values that are used by an action are described by value specifications (see ValueSpecification), obtained from the output of actions or in ways specific to the action. The UML specification contains a framework for dealing with actions, but does not provide syntax. In the stereotypes below, the syntax is given for actions, and these actions are mapped to the UML framework.

An activity defines the effect of a transition or the body of an operation.

AddStructuralFeatureValueAction is used to define an assignment to structural features of a Class or other Classifier. AddVariableValueAction is used to define assignment to local variables of compound statements. <<Break>> OpaqueAction is a break action within a loop that terminates the loop. A call operation action maps to the call of a procedure in the SDL abstract grammar. Every ConditionalNode is either a <<Decision>> ConditionalNode or an <<If>> ConditionalNode. A <<Continue>> OpaqueAction represents a continue action within a loop that causes a jump to the next iteration of the loop or termination of the loop if already in the last iteration. A create object action is used to create instances of agents and store a reference to the created instance in a variable. An <<Empty>> OpaqueAction represents an action that does nothing. A <<Decision>> ConditionalNode is used to define textual switch statements and maps to a *Decision-node* in SDL. There is no graphical notation, but a PseudoState with kind choice (which has no textual form) also maps to a *Decision-node*. An <<ExpressionAction>> OpaqueAction represents an action that only contains an expression. A LoopNode stereotyped by <<For>> is represents a traditional programming language for loop. An <<If>> ConditionalNode is used to define a textual if statement and maps to a *Decision-node* in SDL. Every LoopNode is a <<For>> LoopNode or a <<While>> LoopNode. Every OpaqueAction is one of the subtypes: <<Break>> OpaqueAction or <<Continue>> OpaqueAction or <<Empty>> OpaqueAction or <<ExpressionAction>> OpaqueAction or <<ResetAction>> OpaqueAction or <<SetAction>> OpaqueAction. The reset action cancels a timer and removes any corresponding timer signals. A return represents the action to return from a procedure (in the SDL abstract grammar) to the point where the procedure was called. A sequence node is a sequence of actions and is either a node of an activity or describes the body of a compound node. A send signal action outputs a signal from the executing agent, optionally specifying the target agent and the port used to send the signal. The set action gives a timer an expiry time. A stop represents the action to terminate the enclosing <<ActiveClass>> Class instance (the enclosing agent). A LoopNode stereotyped by <<While>> represents a traditional programming language while loop.

Value Specification

Metaclasses from Kernel package:

Expression, InstanceValue, LiteralInteger, LiteralNull, LiteralString, LiteralUnlimitedNatural, ValueSpecification.

Gives a concrete syntax for expressions (like Z.100).

Literals are defined for the basic types.

A ValueSpecification is an Expression or InstanceValue or LiteralSpecification, and the mapping to the abstract grammar is determined by these metaclasses.

<expression> ::=
    <expression0>
  | <range check expression>

<range check expression> ::=
    <operand2> **in type** { <sort identifier> <constraint> | <sort identifier> }

<constraint> ::=
    constants <left parenthesis> <range condition> <right parenthesis>
  | <size constraint>

<size constraint> ::=
    size <left parenthesis> <range condition> <right parenthesis>

And so on.

Lexical Rules and Predefined Data

- The lexical rules are basically as Z.100
  - Some symbol change (e.g. assignment)
  - Different treatment of names
- Predefined matches Z.100 Predefined

<lexical unit> ::=
<name>
| <integer name>
| <real name>
| <character string>
| <character>
| <hex string>
| <bit string>
| <note>
| <composite special>
| <special>
| <keyword>
| <quoted name>

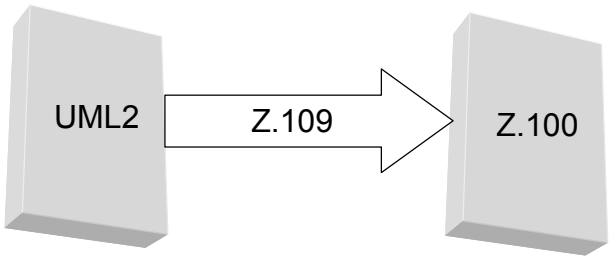
NOTE: The syntax rules for <name>, <alphanumeric>, <letter>, <uppercase letter>, <lowercase letter>, and <decimal digit> are given in .

<integer name> ::=
<decimal digit>+

<real name> ::=
<integer name> <full stop> <integer name>
[{ e | E } [<hyphen> | <plus sign>] <integer name>]

<quoted name> ::=
<apostrophe> { <quoted name character><quoted name character>+
| <reverse solidus><any character except btnfr> } <apostrophe>
| <apostrophe> <apostrophe>

Use of Z.109



Conclusions

- What Z.109 does not cover
- Other profiles
- Implementations
- Further work and the future