SDL-2010 Revised 2016

On the 29th April 2016, ITU-T approved a revision to SDL-2010 (Specification and Description Language): ITU-T Recommendations Z.100 (04/16), Z.101 (04/16), Z.102 (04/16), Z.103 (04/16), Z.104 (04/16), Z.105 (04/16), Z.106 (04/16) and Z.107 (04/16). The revision updates the previous 2011/12 version of SDL-2010. It consolidates the updates previously published in versions of the Specification and Description Language implementer's guide: version 2.0.1 – Z.Imp100 (09/13), and version 2.0.2 – Z.Imp100 (04/15). The Open Issues listed in the implementer's guide have been considered and revised so that in the current Z.Imp100 (03/16) has an empty list of Open items.

A revision of the formal definition of the Specification and Description Language in Z.100 Annex F was approved in January 2015 that is updated to bring it more in line with SDL-2010. However, it is not yet fully aligned and as of May 2016 there is an ongoing work item for further work on Z.100 Annex F.

The following details the significant changes made to SDL-2010: minor defect corrections and clarifications are omitted.

1 Object data

In the 2011/12 version of SDL-2010, object-oriented data was left for "further study". This study was concluded in April 2012 with the issue of Z.107 (04/12) – Object oriented data in SDL-2010. The April 2016 versions of Z.100 to Z.106 have therefore been updated to reference Z.107 for object-oriented data.

Every value sort has a unique value denoted by `null`, which is only valid as an operand of an equality expression or the right hand side expression of an assignment. To be valid in an equality expression both sides of the expression have to be sort compatible and have a REF aggregation kind. To be valid in an assignment, the left hand side of the assignment has to have a REF aggregation kind. Otherwise *null* is invalid and raises the predefined exception `InvalidReference`. There is a predefined literal operator for each value sort with the name `Null` that has `null` as its result, with the consequence that `null` and `Null` can be used interchangeably.

When the left hand side of an assignment has a REF aggregation kind, the right hand side (normally a sort compatible variable or field) is associated with the variable for the left hand side. If the right hand side is "undefined" then the left hand side becomes "undefined". If the right hand side is `null`, the left hand side becomes `null`. If the variable on the left hand side is subsequently accessed where a PART aggregation is required, a copy of the value of the right hand side of the assignment is made (except if the left hand side variable was "undefined" or `null`, in which case an exception is raised).

2 C language binding

In October 2012 an amendment to Z.104 (12/11) was approved that replaced Annex C language binding, that allows C programming language syntax to be used as an alternative to the native SDL-2010 data language. This amendment is now consolidated into Z.104 (04/16).

Contact: Rick Reed
TSE, UK
Tel: +44 153 948 8462
Email rickreed@tseng.co.uk

3 Lexical rules for <equals sign> and <not equals sign>, equality abstract grammar

These lexical units ("=" and "/=") are no longer treated as an alternative of <infix operation names> as this conflicted with their use as part of <equality expression>. They can still be used as quoted operation names: for example "="(a, b) and therefore are added to <quoted operation name>.

It is not allowed to define a new operation with two parameters and an <operation name> that is <quotation mark> <equals sign> <quotation mark> or <quotation mark> <not equals sign> <quotation mark>.

To distinguish between equality expressions for the <equals sign> and <not equals sign>, the abstract grammar is changed so that *Equality-expression* is either a *Positive-equality-expression* (for <equals sign) or *Negative-equality-expression* (for <not equals sign).

4 Structure data type `Make` operator

The nullary `Make` operator is no longer defined by default.

If the structure data type is not a specialisation (that is, one that inherits from another structure data type), and a there is no explicitly defined `Make` operator, there is a default `Make` operator with an argument for each field of the structure (except optional fields or fields with default initialization). Alternatively one or more `Make` operators can be explicitly specified, one of which could be a nullary operator.

If the structure data is a specialisation, it inherits the `Make` operator (or operators) of the parent type.

5 For loop refinements

The <loop variable definition> syntax is extended to include an <aggregation kind> before the <variable name> for the loop variable, and <finalization statement> is extended to allow <compound statement> as an alternative to <statement>.

6 Octet encoding rules (OER) added

The language is extended to include OER (octet encoding rules) as an alternative to the previously defined encoding rule names (such as BER).

7 General Input and Output

The language is extended with a notation to refer to the last signal input saved in an implicit **signal** variable, which can then be used to output the signal. The advantages of this are: the values received can be retrieved from the **signal** without explicitly storing them in variables, and the **signal** can be simply output without change.

This does not require change to output, because <expression output> already allows the output signal to be specified by an expression provided that expression has the sort of a choice data type where each choice field name corresponds to an outgoing signal carried from the local agent by one of the gates of the agent. The **signal** <imperative expression> is defined as an implicit variable that has a choice data type that can hold any of signals that can be received by the agent. This choice data type can be denoted by an <as interface> of the form **`as interface`** `agent1`, where `agent1` is the name of the agent. The **signal** variable is initialized as undefined, and updated when the *In-choice* of an *Input-node* is denoted by **signal**.

8 Reset all timers

Two new short hand notations are added:

If `t1` and `t2` are defined as timers and there are the set statements:

```
timer t1, t2 (Integer);
set (1.0, t1);
set (2.0, t2(1));
set (3.0, t2(2));
```

The statement:

```
reset (t2 *);
```

resets both the timer `t2(1)` and the timer `t2(2)` and any other timer with the name `t2`.

The statement:

```
reset (t1 *);
```

resets the timer `t1` and therefore has the same meaning as **`reset`** `(t1)`.

The statement:

```
reset *;
```

resets all timers.

9 Access values of signals in the input port

The language is extended to provide a way of accessing the name and the parameters of a signal instance in the input port before the signal instance has been received and without changing the contents of the input port. It applies only to signal instance that are "available" (availability time <=now). The signals in the input port are treated as a read-only string of values of the choice data type corresponding to the valid input signal set. The choice data type has the same data type the **signal** above.

The language is extended with the <imperative expression> **`signallist`** that is treated as a read-only implicit variable that accesses the input port.

The syntax of <basic sort> is extended with an alternative <as signallist> that is denoted **`as signallist`**. This represents the string sort of the of the signal instance values in the input port.

If the name of the agent is `agent_name`, it has an implicitly defined interface with the name `agent_name`. This interface has the valid input signal set of the agent as its *Signal-identifier-**set***.

There is a choice data type defined for the interface with a choice of sort for each distinct *Signal-definition* identified by the *Signal-identifier-**set*** of the *Interface-definition*. Given that the name of the agent is `agent_name`, the anonymous implicit name of this choice data type is denoted by "**as interface** `agent_name`".

The signal instances in the input port are stored in order of arrival in the implicit read-only string variable **`signallist`**. This has the data type:

```
value type AnonSignalString
   inherits String < as interface agent_name > ( empty = emptystring )
endvalue type AnonSignalString;
```

where `AnonSignalString` is an anonymous unique name denoted by **`as signallist`**.

`length(`**`signallist`**`)` is the number of available signals (signal instances with availability time > `now` are ignored) and has the value zero if there are no available signals in the input port.

`signallist` `= empty` if and only if there are no available signals in the input port.

`first(`**`signallist`**`)` is the first signal instance to be considered in a state, and has the same meaning as **`signallist`**`[ 1 ]`. Whether this signal instance is consumed depends on whether for the current state this signal saved, or whether there are inputs with higher priority.

`signallist``[ n ]` where `n` is a Natural expression is the `n`th available signal in the input port – a choice value.

`signallist``[ n ]!Present` gives a literal that is the name of the `n`th available signal in the input port and one of the literals for the anonymous data type with the field names of the choice sort **`as interface`**, and can be used to make a decision based on the signal name.

Once the name of the `n`th signal in the input port has been determined, the parameters of the signal can be extracted using a field name (if one was given in the signal definition) or field number. For example:

```
signal signal1 ( Pid, Integer);
signal signal2 (b2 Boolean, i2 Integer);
value type sig3struct { struct c3 Charstring; } endvalue type sig3struct;
signal signal3 struct sig3struct;
dcl agent1 Pid; dcl n Integer ( >= 1) ;
dcl routeNumber2 Integer;
dcl sig3copy sig3struct;
```

Given the definitions above

```
if signallist[n]!Present= signal1 /* is it a signal1 signal */
then agent1 := signallist[n]!as signal signal1!1;
     /* yes, determined by PresentExtract() of the choice value,
        from the choice for signal1 – selected by <as signal>
        (must use <as signal> if signal1 not a unique signal name) -
        extract the first struct field (the Pid field) using <field number>,
        and assign the Pid value */
/* */
if signal2Present (signallist[n]) /* is it a signal2 signal */
then routeNumber2:= signallist[n].signal2!i2;
     /* yes, determined by testing the choice value for signal2
        from the choice for signal2 – selected by unique signal2
        (assumes signal2 is a unique field name) -
        extract the named i2 field of the struct value for the signal,
        and assign the Integer value */
/* */
if signal3Present (signallist[n]) /* is it a signal3 signal */
then sig3copy:= signallist[n]!signal3;
     /* yes, determined by testing the choice value for signal3
        assign the choice for signal3*/
```

As well as having a signal type and the actual parameters of the instance, a signal instance in the input port has additional information associated with it: the sender `Pid`, the availability time (always <= `now`), the signal priority and the arrival gate. None of these are accessible using *Signal-expression*. In a particular application it is suggested to place this information in signal parameters to make it available – probably the best way if ASN.1 and encoding is being used.

10 Simpler initialization of systems and dynamic routing

When a signal is output, where the signal is delivered is constrained by the communication paths from the sending agent and further constrained by a <destination> and a <via path>. Typically there are several agents to receive the signal according to the communication paths, possibly more than one agent instance set and more than one process within a specific agent instance set. The <destination> and <via path> are therefore important if the signal has to be received by a specific agent. The destination identifies a specific agent instance to receive the signal by means of a pid value denoted by an <expression0>. In some cases the pid value needed is simply available from stored values of items such as **`sender`** or **`offspring`**. In other cases, particularly when initializing a system or a transaction, the pid value needs to be derived from something else such as an application equipment number or routing code, and the derivation of pid values from the other factors needs to be set up when the agent instances are created.

The language is extended with a notation for the pid value of an agent instance of an agent that has a static number of instances (assured by making the initial, maximum and minimum the same number). When agent instances are created dynamically, **`offspring`** can be used.

When a signal is sent without a pid value <destination> to a process instance set or via a communication path, the result is often that there is an arbitrary selection of the agent instance. In this case, the issue is how to make the instance selection specific (or at least less arbitrary) to control the dynamic routing.

Each agent instance has a unique `Pid` value: the agent instance pid value. To handle agent instance `Pid` values in models, what usually has to be done is to rely on some tool provided mechanism, or to dynamically record the agent instance `Pid` values that are allocated in some data store(s) within the model.

The dynamic recording approach requires all processes to communicate with the data store(s) and this is an issue with initializing the system. Real systems are often re-initialized to overcome error situations, and the time taken for the system to restart should usually be as short as possible. In a system with a large number of statically allocated agents, recording the agent instance `Pid` value in the data store(s) could take a significant time, and it is probably best not to handle environment signals until the data store(s) have been set up.

For the initial agent instances in a system, allocation of agent instance `Pid` values is able to done before the system interpretation. In fact, like initial memory allocation for static data, allocation of the initial agent instance `Pid` values can be done when the initial system image is generated, so that restarting the system requires only the system to be reset to its initial state, which may only require some dynamic data to be re-initialized.

In the definition of an agent, visibility restricts the agent to only agents at the same or higher level or immediately contained agents. An agent or state machine at the system level does not have visibility of agents within other agents at the system level. For a data store to be initialized with agent instance values, the instances need to be visible. For initializing the data stores the normal visibility rules need to be relaxed.

It is a requirement that each `Pid` value uniquely identifies one agent instance. Each block/process agent instance is a member of an instance set directly contained within another block/process agent instance or is a member of the system instance. Any particular instance set is either the system or is identified by a list of agent instances hierarchically from the system until the instance set is identified. All block/process instance set names have to be unique at each level. An agent instance is identified in its agent instance set by name by an instance number within that set represented by a `Natural` value.

The keyword **`system`** identifies the system instance. Other agent instance `Pid` values are denoted in a hierarchical way from the system ignoring visibility rules. The denotations of agent instance `Pid` values can be used anywhere in the model. The syntax is

<agent instance pid value> ::=
system [< name>]
[**value** < agent instance> **endvalue**
| <left curly bracket> <agent instance> <right curly bracket>]

<agent instance> ::=
{ < agent instance> }*
[**block** | **process**] <agent name>
[<left square bracket> <Natural expression> <right square bracket>]

where <agent name> is an agent instance set name, and the value of the <Natural expression> identifies the agent instance in the order of creation in the agent instance set.

The leftmost items in an <agent instance> list in <agent instance> are allowed to be omitted if the the agent is otherwise unambiguous (similar to omitting the leftmost items in a qualifier).

If the optional bracketed <Natural expression> of an <agent instance>, this is the same as a value of "1".

The syntax for <expression0> is extended to include <agent instance pid value>.

A pid variable can be initialized in a <variable definition> by a <constant expression>. A <constant expression> is defined as a <constant expression0> "that does not contain any <active primary>, or a <value returning procedure call>". If all the <Natural expression> items of the <agent instance> items of an <agent instance pid value> are a constant expressions, the pid value is evaluated only once when the system is initialized and is a <constant expression>.

An <agent instance pid value> is a pid value of the implicit pid sort for the type of the agent instance set and is compatible with the predefined `Pid` sort.

For the system and each agent instance set where the minimum, maximum and initial number of instances are the same ("static" instance sets) the number of instances never changes, and each <agent instance pid value> represents a unique `Pid` value for a given <Natural expression> value.

If the agent instance set is not static, and an instance of set ceases to exist, the instance number for the next and all subsequent agent instances are reduced by one. Therefore if `myagent [2]` terminates, the agent instance pid values denoted by the (constant) expression **`system`** `{myagent [3]}` would be the (dynamic) expression by **`system`** `{myagent [i+2]}` where `i` has the value `0`.

10.1 Examples

Most initial agent instance sets will contain more than one agent instance, and therefore the agent instance `Pid` values for an agent instance set need to be stored in a variable that can be indexed to provide the different values. A suitable data type is:

```
value type PidString
   inherits String < Pid > ( empty = emptystring )
endvalue type PidString;
```

Assume that the system directly contains an agent instance set named `agent1` that has three initial instances. The expression to initialize a `PidString` is illustrated for values for the three instances of `agent1`:

```
dcl agent1Pidstring PidString:=
mkstring( system { agent1 [1] } ) //
mkstring( system { agent1 [2] } ) //
mkstring( system { agent1 [3] } ) ;
```

An alternative data type in this case is:

```
value type Agent1String
   inherits String < agent1 /* implicit pid sort */ > ( empty = emptystring )
endvalue type Agent1String;
```

Because `agent1` is the name of the agent instance set, it is also the name of the pid sort for the agent interface.

```
dcl agent1pids Agent1String:=
mkstring( system { agent1 [1] } ) //
```

```
    mkstring( system { agent1 [2] } ) //
    mkstring( system { agent1 [3] } ) ;
```

If `agent1` contains an agent instance set named `agent11` with 4 initial instances that contains an agent instance set named `agent111` with 5 initial instances, there are at least 76 initial agent instances (including the system, but ignoring any other instances directly contained by the system). For the 4th instance of `agent111` in the 3rd instance of `agent11` in the 2nd instance of `agent1` the `Pid` value is denoted:

```
    system { agent1 [2] agent11 [3] agent111 [4]} ) ;
```

11 Routing to an instance of a visible agent

SDL-2010 currently allows a <destination> to be an <agent identifier>. If there is only one instance of the agent this uniquely identifies the destination. But more typically there is more than one instance of the agent. The language is extended to allow a Natural expression after the <agent identifier> to select a particular instance.

```
<destination> ::=
                    <pid expression0>
            |       [ system | block | process ] <agent identifier> [ <destination number> ]
            |       this

<destination number> ::=
                    <left square bracket> <Natural expression0> <right square bracket>
```

The optional keyword (system, block or process) before an <agent identifier> shall match the agent kind.

The agent instances are numbered consecutively from 1 when the destination is interpreted, in the order in which the instances were created: this allows for changes in numbering due to instances terminating. If the destination number is zero or greater than the number of instances in the set of agent instances, the signal is discarded.