

# SDL-2000 for new millennium systems

BY RICK REED

*SDL is the premier language for specification, design and development of real time systems, and in particular for telecommunication applications. SDL-2000 became the international standard in force in November 1999, replacing the previous version. This paper gives an overview of SDL-2000 and fills the gap between previously published tutorials and the current SDL standard.*

## 1 Introduction to SDL

The success of SDL [1, 2] can be attributed to its graphical presentation form. This makes it easy to understand specifications and designs expressed using SDL. They are good for communication even to anyone that has little knowledge of the language. Another factor is the conceptual suitability of the basis of the language: the notion of an extended finite state machine (EFSM). SDL offers a practical way of specifying systems with several communicating EFSM instances. An SDL system consists of one or more communicating agents. There is one outermost agent: this communicates with the environment. In agents, there is definition of behaviour by EFSM, hierarchical structure with agents containing agents, data variables (owned by agents) of value or reference data types, and communication based on asynchronous message exchange.

When systems are specified or designed (in the rest of this article the verb “specify” should be taken to include design), the usual starting point is some kind of top level picture showing the connection between components of the system and the environment. Such pictures usually take the form of labelled boxes joined by labelled lines. SDL can be used, even at this level, to start turning sketches into a formal system description: the names on boxes become the names of SDL components and the names on lines can become the names of SDL channels or associations. Such descriptions are abstract models of real systems. Of course, as an object oriented language, SDL can also be used bottom-up, based on a set of components, or “middle-out”.

The SDL specification for a system is a set of diagrams. Each diagram has one or more presentation “pages”, and each page has:

- a frame (often with some information attached on the outside);
- the diagram heading giving the kind and identity of the item described by the diagram in the top left corner;
- the page name and number of pages in the top right corner.

**Keywords:** SDL, SDL-2000, SDL-92, SDL-88, current SDL, MSC, ITU-T, Z.100, Z.105, Z.106, Z.107, Z.109, ASN.1, Z.120, Conformance, Standards, Recommendations, simplification, exceptions, SDL combined with UML, data methods, data objects.

### 1.1 Simple structure

A very simple example is shown in Figure 1. This system agent diagram contains two process agents. A channel ( $\rightarrow$ ) conveys signals between two agents or between an agent and the environment of a diagram. The signal names are listed in the [ ] symbol near the arrowhead, which gives the direction. Channels can have names, but these are omitted here, as they are not needed. A system diagram can contain process agents ( $\square$  symbol), or block agents ( $\square$  symbol). The system itself is the special case of the outermost block agent.

The essential difference between a block (or system) agent and a process agent is that the instances of agents within a block agent behave concurrently and asynchronously with each other, whereas instances within a process are scheduled one at a time. A block agent can contain process agents or block agents. A process agent can only contain other process agents.

As well as containing other agents, agents can contain a state machine, data variables and procedures. An agent SDL diagram is the definition of a set of agent instances. Each agent instance of such a set is created either when the instance containing the set is created or by a create-action in another agent instance. The system agent is created when the system is initialized.

Agent diagrams act as scope units hiding internally defined items. These include the items mentioned above, signals for communication and locally defined types of data. Items defined in enclosing agent diagrams are visible in inner agents. Thus, the signals *0* and *1* are visible inside *send\_bits* and *receive\_bits*. On the other hand, items defined inside these process agents are not visible at the system level.

The  $\square$  symbols containing the names (and similarly  $\square$  symbols containing names) are links (called “references” in Z.100) to other diagrams considered to be defined where the symbol occurs (**process** *send\_bits* is defined in the **system** *bitstuff\_transmission*). The defining context and kind of the entity (such as **block**, **process**, and **signal**) is part of entity’s identity. Complete identities must be unique, but names need not be unique.

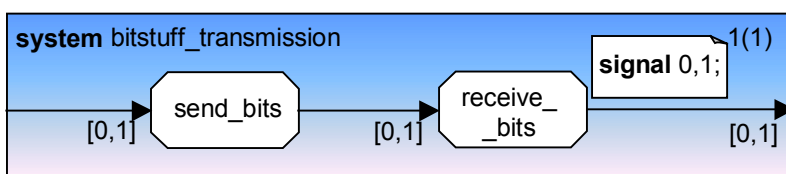


Figure 1 Example simple system model – Bit-stuffing one-way transmission.

This system consists of a *send-bits* transmitter and a *receive-bits* receiver. The transmitter inserts (“stuffs in”) bits so that there are never *n* bits the same. The receiver removes the inserted bits. This technique is used in real systems to protect against “stuck at zero or one” or (for example in Signalling System 7) to allow flags that consist of *n* ones or *n* zeros to be inserted without the risk that they are imitated by signals.

#### Names and underlines

Names consist of letters, digits and underlines; names that only contain digits are allowed. However, an underline character at the end of a line is a continuation and not part of a name.

### Uniqueness and qualifiers

A name is usually sufficient to identify an entity, but the full identifier includes a qualifier that gives the context where the entity is defined.

The qualified signal `<<system bitstuff_transmission>> 0` is distinct from the Integer data item called `0` or a signal `<<send_bits>>0` (that is, a signal `0` defined in `send_bits`). In practice, these qualified names (`<<context path>>` is a “qualifier”) need only be used when necessary, which occurs rarely.

### 1.2 Simple behaviour

An agent diagram, such as Figure 1, has the possibility to show the interaction between the contained agents, and is called an interaction diagram. An agent that only contains one state machine (typically a **process**) can have the behaviour graph in the agent diagram (otherwise, it has to be linked to a **state** diagram for the state machine graph).

In Figure 2, the `send_bits` process contains a finite state machine that has:

- a start (○ symbol) where it starts;
- states (□ symbols) containing state names: *initial*, *0*, *00*, *0000*, *1*, *11*, *111* and *1111* with associated inputs (▷ symbols) for the stimulus signals *0* and *1*;

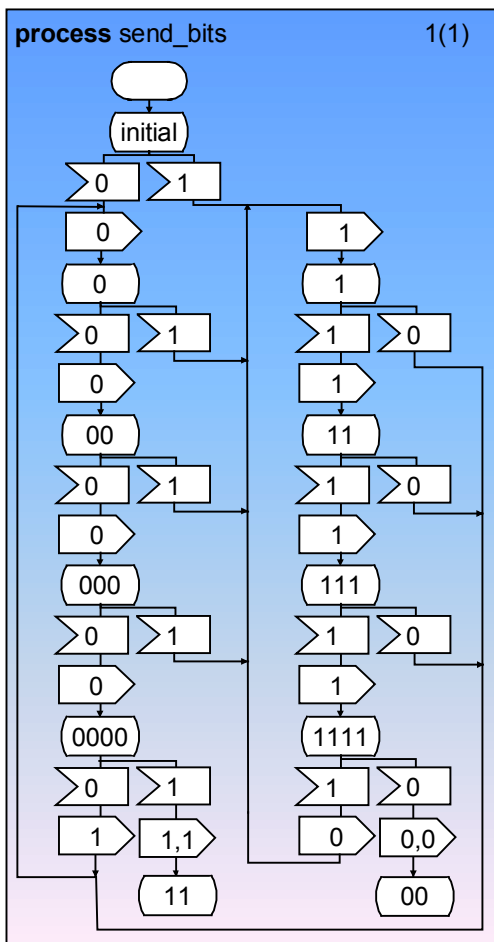


Figure 2 The `send_bits` process as a finite state machine.

- transitions to the next state with outputs (□ symbols) for the response signals *0* and *1*.

The response of the state machine is determined by following the flow from state to state in the diagram. The start leads to a state, possibly via other symbols. Once at a state, the machine waits until one of the signals that can be consumed in the state is available. This is immediately if the first signal queued in the agent’s input port can be consumed, otherwise the machine will wait. Each input leads to other states via other symbols (such as outputs) to the next state.

An output symbol may contain more than one signal (in the example `1, 1` meaning that two `1` signals are sent). The next state can be indicated by a □ symbol with the state name (in the example `11` after the output of `1, 1`), which in this case acts as a connector.

SDL extends the finite state machine paradigm in two important ways:

1. Each agent has an input port that queues received signals on a first-in-first-out basis, so that the signals are (normally) processed in the order they are received;
2. Data can be received in signals, stored in variables, manipulated, used in expressions, used to decide how the agent will behave, and passed in output signals.

The `receive_bits` process in Figure 3 uses data, and therefore the

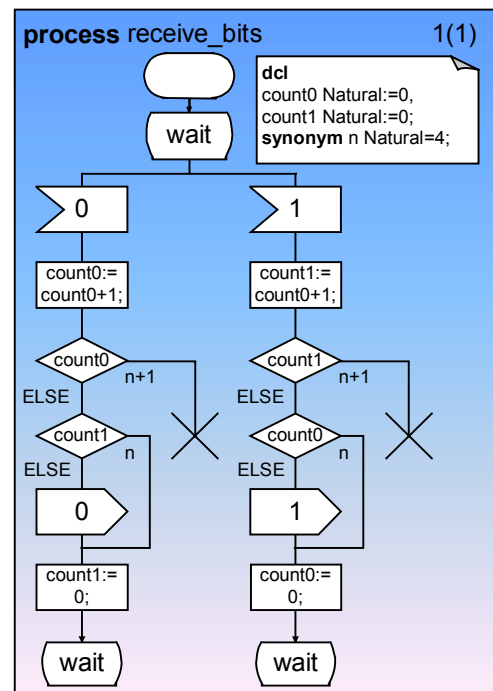


Figure 3 The `receive_bits` process as an extended finite state machine.

number of explicit states is reduced to one and the specification allows  $n$  to be easily changed to any value. The data declaration (**dcl**) introduces two variables, *count0* and *count1*, and a **synonym** relation is defined between  $n$  and a constant value 4. The *receive bits* process also has:

- decisions ( $\langle \diamond \rangle$ ) that can have two or more alternatives, one of which can be **else** – the path taken after a decision is the one labelled with a value that matches the expression in the symbol;
- tasks ( $\square$ ) that contain one or more statements – typically assignment statements, but can include textual loops, textual procedure or method calls, textual if, and textual decision statements;
- text ( $\square$ ) symbols that are used to contain data definitions, signal definitions and other textual definitions;
- stops ( $\times$ ) for terminating the state machine and in this case the process agent.

Note that the stops are unreachable in the example if the *send\_bits* process works correctly.

## 2 Basic communication and timers

As seen in the example in the previous section, signals are the primary means of communication between state machines (see 7 for other means). Timers provide a real time element to SDL, and generate associated timer signals.

### 2.1 Signal communication

Signals can be defined with or without parameters, and the paths used are shown by the lists attached to channels and gates as shown throughout the figures in this article. An output using the signal name generates an instance of the signal. When a signal instance arrives at the destination agent, it remains in the input port until it is consumed, at which time the instance ceases to exist. On output, parameters of a signal can be given the values of expressions listed in parentheses after the signal name. On input, the parameters of a signal can be assigned to variables listed in parentheses after the signal name.

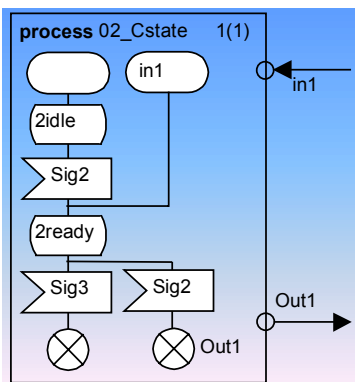


Figure 4: The state 02\_Cstate.

When there is more than one path, communication can be directed in the output to specific destinations by a processing identity (Pid) value, an agent name or **via** path. If there is more than one path, an arbitrary one is used.

These values can be stored in variables for use later. In Figure 5, *X* can only take path *c1*, but *Y* can take *g1* or *c1*. *Y via c1* ensures the signal goes to *p2*. *Y to sender* or *Y to kid* directs the signal to a specific destination but on either path.

Four Pid expressions are available to each agent for communications:

- self** an agent's own identity;
- parent** the agent that created the agent – Null for initial agents;
- offspring** the most recent agent created by the agent – Null initially or if creation fails because the maximum number of instances already exists;
- sender** the agent that sent the last signal input- Null before any signal received.

### 2.2 Timers

An agent can have timers defined. A timer is created by a definition, such as

```
timer t4 := 10.5;
```

A timer can be started with a **set** and cancelled with a **reset**. When the timer is **set** it becomes active and will expire when the time specified in the set has been past.

The expression

**active (t4)** tests if the timer *t4* is active.

**set(now+3.2, t4)** – sets the timer to 3.2 from the current time.

**set(t4)** – sets the timer *t4* to the duration (optionally) given in the timer definition from the current time, which for *t4* is now+10.5, see Figure 6.

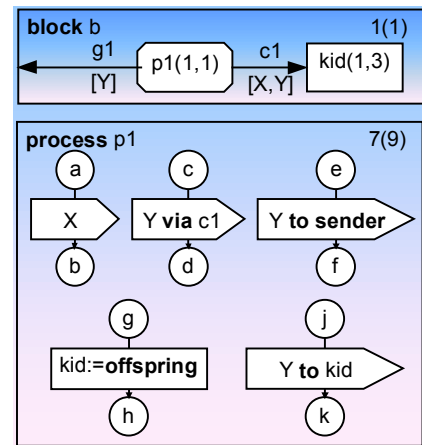


Figure 5: Number of instances; signal directions in output.

If the timer expires then a signal of the same name (in this case *t4*) is put in the input port of the agent. It is quite usual to have a **reset** (*t4*) before the timer expires in which case it is cancelled, or if the signal is already in the input port, it is removed.

A typical use of a timer is shown in Figure 6.

Timer definitions are NOT allowed in state diagrams or procedure diagrams (outlined in 6.3 and 6.4 respectively).

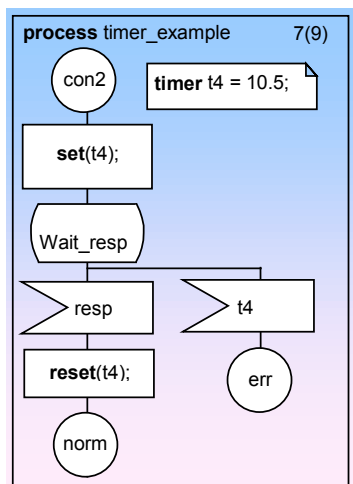


Figure 6: Typical timer use.

### 3 System engineering

Although the state machines are essential to specify behaviour (that is, what responses are given to particular stimulus sequences), complex systems often involve several levels of decomposition before state machines are reached. After producing a top-level diagram, the next step is often to determine the various attributes and structures of each component rather than designing state machines. Some would argue that recognising the “objects” in the system, their attributes and the relationships between objects should be the first step.

Rarely are engineers given such a simple case as in Figure 1. More likely the case would be more complex as indicated by the following informal statement: “The message transfer part of our system has some control transfer functions that interface with link control functions. Link control uses data signalling links defined by the following standards ... Design the Link Control Function (LCF) to support the Message Transfer Part (MTP) with the following characteristics... LCF is expected to ...”.

In most cases, engineering involves domain and requirements analysis as well as application specification, design and implementation. For analysis, knowledge and experience are important factors, but natural languages have proved inadequate to complete the task effectively and efficiently [3]. Well-defined notations are needed to provide common understanding of the object and property models and to

enable the models to be checked (before too much money is spent).

The essential models for analysis are use scenarios with use sequences (these can be captured in MSC-2000 [4, 5]) and the object model. SDL-2000 uses the same object model notation as UML [6] for this purpose. A feature of engineering is that the diagrams change and evolve and there may be many different versions, even if only one is retained at the end. The final object model can be a traceable evolution of the initial analysis model.

In the rest of this article, an example has been taken from ITU Recommendation Q.703: Signalling System No. 7 - Message Transfer Part – Signalling Link, otherwise known as level 2. Of the several functions of level 2, the signal unit delimitation, alignment and error detection are considered, which interfaces with level 1, the signalling data link. For delimitation, an eight-bit flag 01111110 is inserted into messages after “bit-stuffing” to ensure six ones cannot otherwise occur. On reception, the flags are removed, and the messages “unstuffed”.

The initial model of a system would normally be considered a “context model” showing the main objects and interfaces. This is usually the initial version of the final top level specification, which for the example is the SDL diagram in Figure 7, the details of which will be described subsequently.

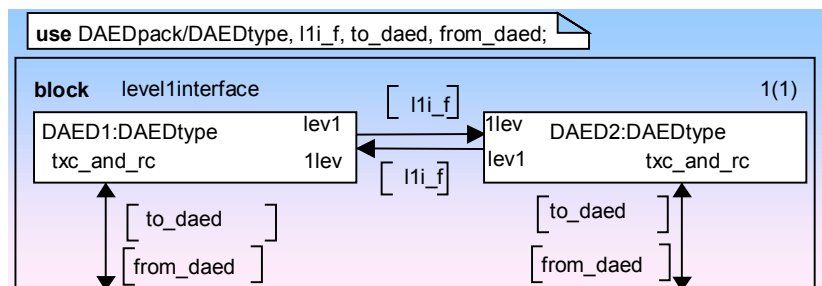


Figure 7 The level 1 interface for Q.703, re-using the same BLOCK for both ends. The communication carried by the channels is defined by attached interface names: *l1i\_f*, *to\_daed* and *from\_daed*.

Analysis of the small part of Q.703 results in the diagrams in Figure 7, Figure 8, Figure 9 and Figure 10 containing:

- an interface *l1i\_f* for transmission and reception of Bits from level 1;
- two interfaces with the rest of level 2, *to\_daed* and *from\_daed*;
- two agents *DAED1* and *DAED2* of type *DAEDtype*, each containing agents for:
  1. “delimitation, alignment and error detection (transmission)” *DAEDT*;
  2. “delimitation, alignment and error detection (receiving)” *DAEDR*;
  3. if error handling is included a “signal unit error rate monitor” *SUERM*, see Figure 10.

### 3.1 Structure and types

The block *level1interface*, Figure 7, uses the **block type** *DAEDtype* from **package** *DAEDpack* (for packages and their use see 3.5). End to end signal unit transport has two DAED units connected by level 1. In Figure 7, the type *DAEDtype* is used twice as the basis for *DAED1* and *DAED2*. The diagram that contains the types, in particular the definition of *DAEDtype*, (often called an “object model”) corresponding to the analysis is shown in Figure 14. Note that for illustration in this article, it is assumed that two systems for the level 1 interface are defined: one without and one with error rate monitoring. Therefore, two versions of the *DAEDR* agent are provided in Figure 14. These two different specifications could (for example) be used as the basis for different conformance tests.

*DAED1* and *DAED2* are linked by the name *DAEDtype* to the diagram in Figure 9, which is linked by the *daedtype* and *daedrtype* in **block type** (□) or **process type** (⊖) symbols to the diagrams that defined these agent types.

The labelled arrows outside the frame in Figure 9 are gates. Channels are connected to these inside the frame, and when the type is used, channels are connected to the gates from outside the relevant symbol. For example, *daedtype* has a gate *txc* that consumes *signal\_unit* signals and generates *transmission\_request* signals. Interface names could have been used instead of signals.

For a system that consists of a single block or process, enclosing digrams are not essential, therefore in Figure 7 there are no connections to the channels outside the frame. Normally a gate or a channel name would have to be shown. In general channel and gate names can be omitted from diagrams if there is no need to refer to the channel. No name is needed on the channels inside the frame, as the communication is clear from the interface names given for

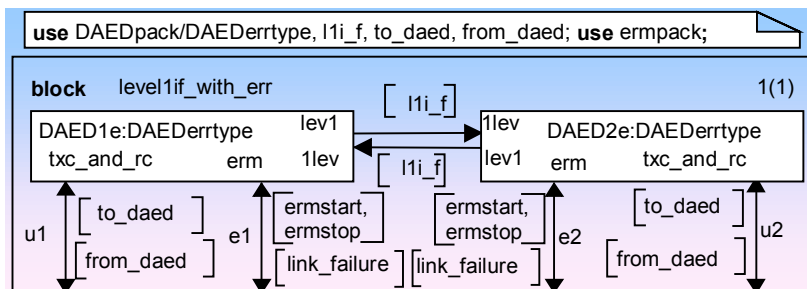


Figure 8 The level 1 interface for Q.703, with error handling.

each direction.

Even when names are not needed by SDL, it is sometimes useful to put them in. In the alternative version of the system (Figure 8), the channels have been named (*u1,e1,e2,u2*) so that it is possible to distinguish between the two sides.

In simple systems such as Figure 1, the object instances are shown as SDL definitions (such as a **block** or a **process**) that have an implied type definition. If several objects have the same properties, using explicit types makes the SDL simpler.

A type definition can be re-used in several places in the SDL specification, and its properties can be inherited to make specialisations of the type. For example in two-way systems, it is quite usual for the transceiver description to be re-used at both ends. In a system with several kinds of termination, a general type of termination can be specialised for each case.

Types have to have fewer context dependencies, so that they can be used in different contexts, and context independence means that types can be used as components in different systems.

### 3.2 Inheritance and virtuality

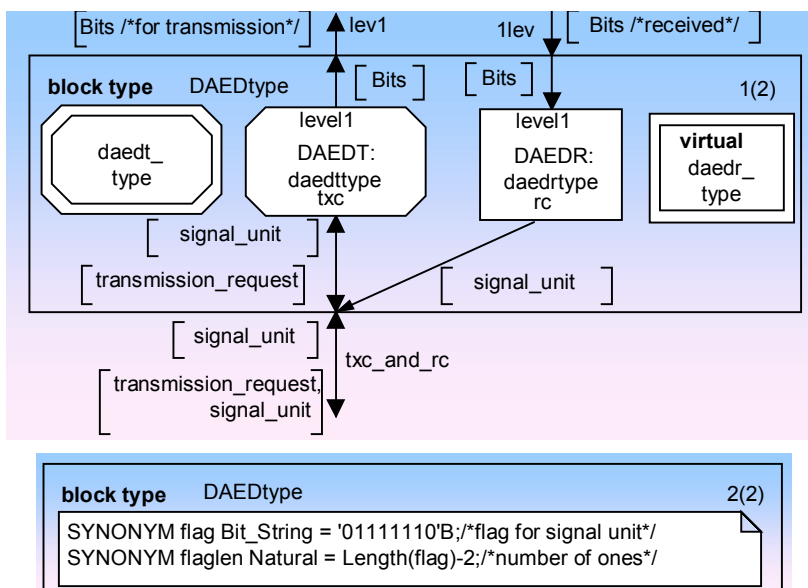


Figure 9: The diagram for *DAEDtype* consisting of two “pages”.



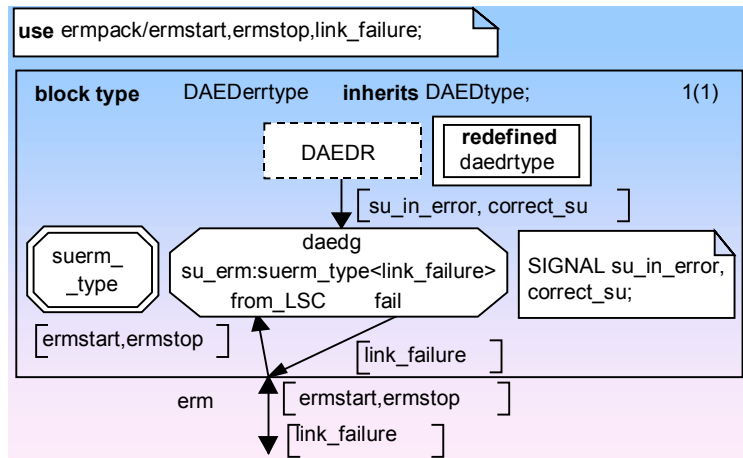


Figure 10 The error handling version inherits the basic version.

When a type simply inherits from another type, it has the same set of properties as the original type, but a distinct identity. More typically, additional properties are also specified at the same time. For example, *DAEDerrtype* in Figure 9 is inherited by the *DAEDRtype*, which handles errors in Figure 10. The extra process agent *su\_erm* is added, based on an extra type *suerm\_type*.

Inheritance is a general mechanism that applies to interaction diagrams, to behaviour diagrams and to data types. In behaviour diagrams new transitions can be added leading to new states. In data types, new operations can be added.

However, it is not always sufficient to add new properties to a type: it may be necessary to redefine some existing properties. For example, in Figure 10 the additional signals needed are generated by *DAEDR* based on the **redefined** *daedtype*.

SDL clearly distinguishes those parts that are **virtual** and can be **redefined**. All other parts are inherited unchanged and cannot be changed: these are “finalized”. The fact that the properties defined by the unchanged parts can be relied upon in sub-classes, is a major advantage over languages where any property of a super-class can be changed in a sub-class. A **redefined** item is virtual, and can be **redefined** again if the sub-class (here *DAEDerrtype*) is inherited again. On the other hand, a virtual or redefined item does not have to be redefined in a sub-class, in which case the definition from the super-class is used. When redefinition is given, an item can also be made **finalized**, so that it then cannot be changed in sub-classes.

The agent type *daedrprocess*, defined in Figure 11 is redefined to generate the extra signals (see Figure 12), but otherwise the structure and behaviour of the rest is the same as in the original *daedtype* in *DAEDtype*. Symbols with dashed lines indicate the use of existing items defined in a super type. The examples here are the existing process (⋯) and existing block (⋯).

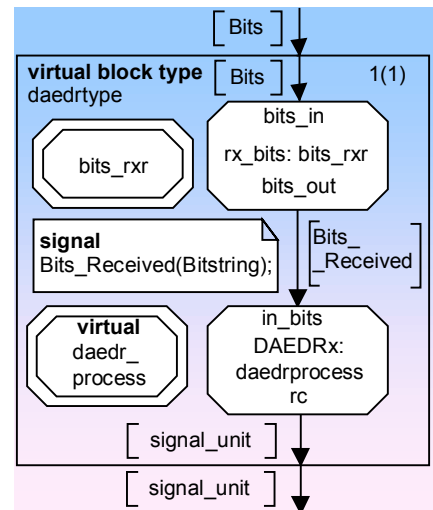


Figure 11 The virtual block type daedtype

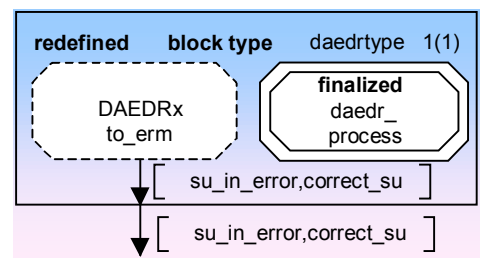


Figure 12 The redefined block type.

### 3.3 Context parameters

Specialisation of types can also be done using context parameters, for which actual parameters must be given before a type is used. As an example, *suerm\_type* has been defined (Figure 13) to have a signal parameter for the *failure* signal, so that the actual signal output can be changed. The actual parameter, *link\_failure*, is given after the use of *suerm\_type* in Figure 10.

Formal context parameters are given in a type definition after the name of the type and enclosed in < and >. The actual parameters are given after the use of the type name enclosed in < and >.

As well as being a **signal**, a context parameter can be a **block**, a **process**, a data variable, a **synonym**, a gate, an interface, a procedure, an exception or timer; or a **type** for a **block** or **process** or data.

### 3.4 Constraints

Context parameters, virtual types and gates can have constraints. A constraint limits the actual parameters, type redefinition and gate connections (respectively). By default, a virtual type is constrained to be a sub-class of the base type (the one with **virtual**). For example, any redefinition of *daedrtype* in Figure 11 must by default be a sub-class of

*daedrtype*. However, it is permitted to specify that the constraint is **at least** some other type, in which case a redefinition can use **inherits** to explicitly inherit another type.

There are no defaults for context parameters, but these can also be constrained by an **at least**. Similarly, gates can normally be connected to any channel that conveys the appropriate signals, but a constraint restricts connections. In Figure 13, gate *daedg* must be connected to a block based on *daedrtype*.

### 3.5 Packages

A package groups several type definitions together and allows them to be used in several systems. Packages can also be used within other packages, and it is quite usual to have a hierarchy of dependencies between packages, which can be shown diagrammatically (not illustrated here for reasons of space).

Figure 14 supports both the systems defined in Figure 7 and Figure 8. Each **interface** contains the definition of the relevant signals, or links to signal definitions by **use** (see *from\_daed*). Interfaces can also include definitions or uses of two other ways of communicating between processes: remote procedures and remote variables (see 7.1 and 7.2).

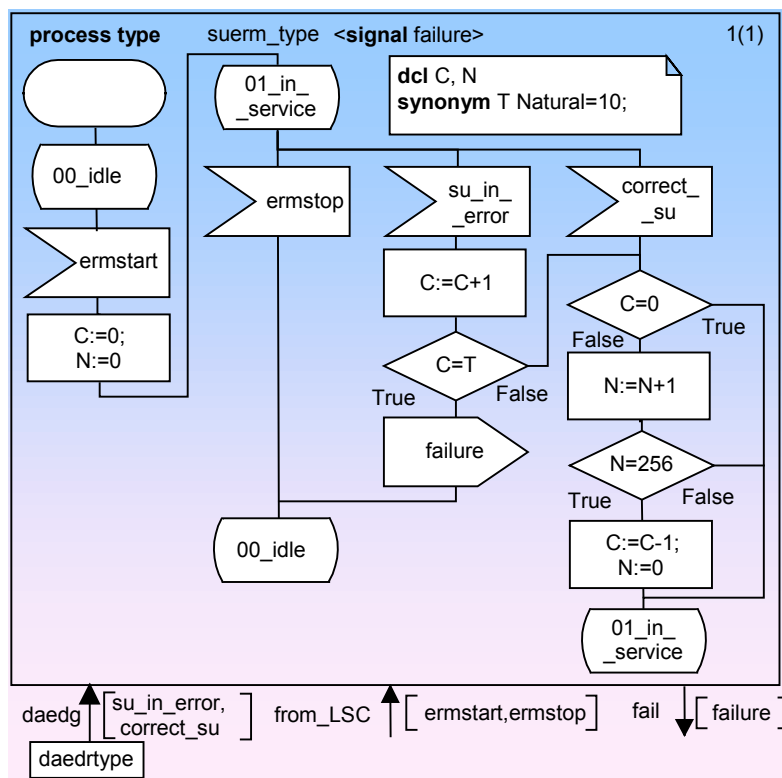


Figure 13 The Q.703 signal unit error rate monitor, adapted with a context parameter.

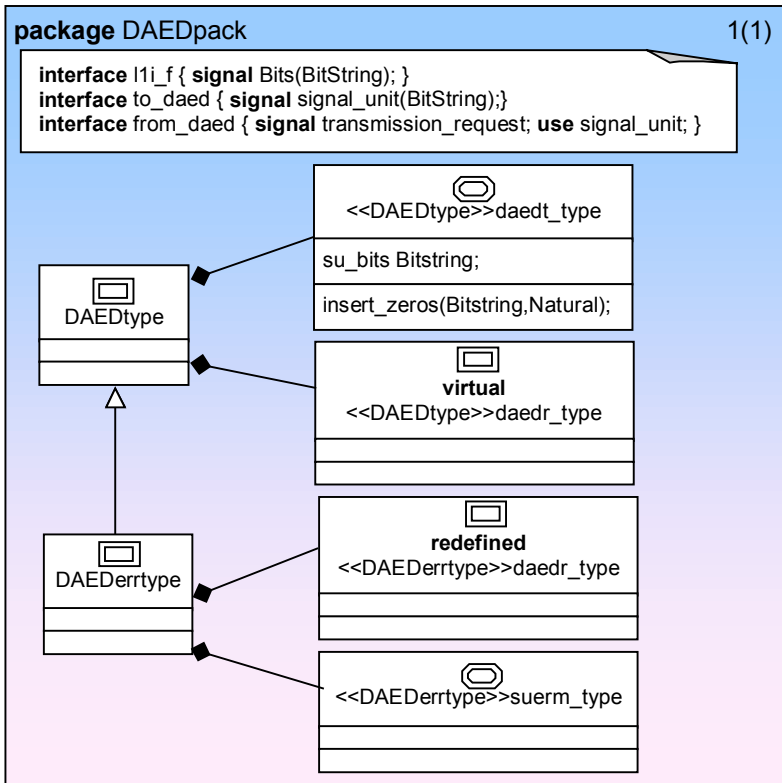







Figure 14: An object model of part of ITU Recommendation Q.703, as an SDL package.

The three compartment class () symbols are linked to type definitions. The top compartment contains the kind, here **block type** () or **process type** (), and identity of the type, such as *DAEDtype*. Where the types are actually defined elsewhere, a qualifier is put before the name: *daedt\_type* is defined in `<<DAEDtype>>`. The specialization relation ( $\rightarrow$  symbol) indicates that *DAEDerrtype* inherits *DAEDtype* so that it includes the properties of *daedt\_type*. The **block type** *daedr\_type* is also inherited, but is **redefined**, which is possible because the original is **virtual**. The **process type** *suerm* is added.

The lower two compartments of a class symbol optionally give a definition of some properties of the linked type, so that a reader does not have to refer to another diagram for them. The middle compartment can contain attribute properties, such as the variables if the linked type is an agent type. The lower compartment can contain behaviour properties such as a procedure name and its parameter sorts or a signal used in the inputs of the linked object. In Figure 14, this use of the class symbol is illustrated only for *daedt\_type*, which has a variable attribute property, *su\_bits* and a procedure behaviour property, *insert\_zeros*. During engineering, it would be quite normal to fill in some properties in the compartments first, and elaborate the linked type later. The real property definition is in the linked type, but tools can assist in copying or checking consistency.

An association () is a form of annotation – it makes no difference to the SDL meaning if it is removed. However, associations are intended to have meaning in UML, and it is

expected that tools will do some checks between associations and the SDL. Associations can be given meaningful names and have attributes at each end (role name, multiplicity range, **ordered**, private or restricted or public visibility). The line can be plain, or with  $\blacklozenge$  or  $\diamond$  at one end indicating “composition” or “aggregation”. A  $\rightarrow$  at either end shows that the end is “bound”. The terms “composition”, “aggregation” and “bound” are not further defined by SDL.

If no properties are included, the class symbols can be “iconized”: that is replaced by the identity inside the type symbol, (such as ) for a **block type**). For examples see *daedttype* and *daedrtype*, used in Figure 9 for the process and block *DAEDT* and *DAEDR* respectively.

The text box at the top of Figure 10 makes **use** of the signals *ermstart*, *ermstop* and *link\_failure*, all defined in a package, *erm-pack*. Also the whole object model is enclosed in a package called *DAEDpack* in Figure 14.

One package named *Predefined*, is an integral part of the language. It defines the data types: Boolean, Character, String, Charstring, Integer, Natural, Real, Array, Vector, Powerset, Duration, Time, Bag, Bit, Bitstring, Octet and Octetstring. Some of these have context parameters that need to have actual parameters to create new data types before they can be used to declare variables (some examples follow).

## 4 Data

SDL data is strongly typed. A data type can be a **value type** that represents a set of values, or can be an **object type** that represents object references. Each sort of data is distinct. An element of one **value type** cannot be assigned where another **value type** is required. An element of one **object type** cannot be assigned where another **object type** is required that is not a sub-type of the first. A **value type** element can be assigned to an **object type** if they are based on the same sort of data: an Integer value can be assigned to an **object** Integer.

```
value type astring inherits String <Natural>;
```

defines *astring* as a data type that is a string of Natural elements.

```
value type chlookup inherits Array <Character, Integer>;
```

defines a data type that is mapping for Character values to Integer values.

```
value type c_array10 inherits vector <mystruct, 10>;
```

defines *c\_array10* as a data type indexed with an Integer in the range 1:10 that gives *mystruct* values (where *mystruct* is a defined data type).

**package** *Predefined* is implicitly part of every SDL model. The example uses Bitstring, which is a string of Bits. Note that Bitstring is indexed from zero to be compatible with ASN.1: all other strings in SDL (including Octetstring) are indexed



from 1. Bit values are 0 and 1. Bitstring values can be '0'B, '1'B, '00'B, '01'B etc. or (for example) 'B3'H meaning the same as '10010011'B. The bit '...'B and hexadecimal '...'H notations are also valid for Integer.

The Pid (processing identity) and Any data types are considered as defined in **package** *Predefined*. Pid has a special role in the language for referencing agents or interfaces to agents, and therefore has a Null to indicate no reference. An Any variable can be assigned a value or reference of any other data type, and is therefore fully polymorphic.

Each of the data types defined in *Predefined* has a set of operations. Some of these provide the normal infix notations for Boolean, Integer and Real (such as **and**, **or**, **+**, **-**, **\***, **/**). Other *Predefined* operations (such as *mkstring*) are operators that use functional prefix notation.

String, Vector and Array based types can be indexed to give an element.

```
dcl a1,a2 c_array10, I Integer;
      /*allows assignments */
      a2:=a1;          /* the whole array */
      a1[3]:=a2[1+1]; /* an element */
```

#### 4.1 User data types

For data types beyond simple types such as Integer, user-named types are defined either using Predefined types with parameters (see *astring*, *chlookup*, and *c\_array10* above), or by constructing new data types. Constructed data types are enumerated with a list of literals, or a **structure**, or **choice** type.

An example of an enumerated list is:

```
value type rgb {literals blue, red=0, green}
```

and has operators **<**, **<=**, **>**, **>=**, **first**, **last**, **succ**, **pred** and **num**. Each literal must have a unique number. Literals without numbers are given (left to right) the lowest available Natural number, so blue=1 and green=2.

A **structure** has any number of fields, each of which can be any named type including other structures, strings, vectors or arrays.

```
value type S { struct
  a      Integer;
  b      Charstring optional;
  c      Character default 'd'; }
dcl s1 S, I Integer, X Character;
s1:= ( 3, '21', 'e' );          /*structure value */
s1.b:= mkstring(s1.c);         /*field access */
```

The presence of the **optional** field *b* can be tested by *s1.bPresent*, which gives a Boolean value. A field that has not been assigned a value is undefined unless it has a **default** value.

A choice is similar to a structure, but can only contain one field at any one time, and assigning one of the choices makes all other choices undefined.

```
value type C { choice
  hue     rgb;
  bs     Bitstring;}
```

```
object type Slist {struct
  elem     S;
  next Slist;
operators make(S)->Slist;
methods add(S);
operator make(s S)
  {return (. s, Null .);}
method add(s S) {
  dcl last S;
  for (      last:=this,
          last.next/=Null,
          last.next);
  last.next:= make( s );
  } }
```

Figure 15: An **object** type for a linked list of *S* elements..

A named data type can be defined that **inherits** the properties of another data type, and properties can be added including operations. An operation can be either an **operator** or a **method**. An operator has a list of parameters and produces a result. A method acts on a variable of the data type (and may change it), and optionally takes a list of arguments, and may produce a result. An operator uses functional prefix notation: *f(a,b)*, whereas a method uses dot notation: *var.methodname(c,d)*. The body of an operation can be defined using a textual algorithm (for example see Figure 15) or by a linked diagram.

A synonym type (**syntype**) can be defined for any data type that is assignment compatible with the parent type. Though this could be used just to give the type another name, this is usually combined with some limitation of the values of the parent type. Values of a **syntype** can be assigned to the parent type, but only those values defined by the **syntype** can be assigned to a **syntype** variable or parameter. A common use is to limit the range of Integer.

```
syntype Int16 = Integer constants 0:65535;
```

For types that have a Length operator (such as strings), a **syntype** can include a **size** constraint. For example **size(0,10)** means the length must be zero or 10.

#### 4.2 Support for ASN.1

Several of the predefined data types have a direct equivalence in ASN.1 [7]. SDL adds operators to ASN.1 data types, so that the values can be manipulated in expressions. Bit, Bitstring, Octet, and Octetstring were added to specifically support ASN.1.

Other mappings from ASN.1 to SDL are defined in Z.105 [8]. This allows an ASN.1 module to be used with SDL, so that the data types defined in ASN.1 are equivalent to data types defined in SDL.

A value assignment in ASN.1 is mapped to a **synonym**.

```
myvalue INTEGER ::= 100;
is mapped to
synonym myvalue Integer = 100;
```

A constrained type in ASN.1 is mapped to a **syntype**, so that

```
T ::= INTEGER(1..10)
    is mapped to
syntype T = Integer constants 1:10;
```

An ASN.1 SEQUENCE (or SET these are treated the same) is mapped to a structure type in SDL, which allows a variable to have a number of fields. For example, the ASN.1

```
S ::= SEQUENCE {
    a INTEGER,
    b CHARSTRING OPTIONAL,
    c CHARACTER DEFAULT 'd' }
```

is mapped to *S* as defined in 4.1 above, and similarly CHOICE is mapped to the SDL **choice**. The corresponding mapping for SEQUENCE values is to omit the field names and convert the value to a structure value.

```
seqval S ::= {a 22, b 'pqr', c 'x'}
    is mapped to
synonym seqval S = (. 22, 'pqr', 'x' .);
```

SEQUENCE OF, and SET OF, are mapped to the String and Bag data types respectively. ENUMERATED types are mapped to types with **literals**. In addition, Z.105 gives mappings for ASN.1 parameterized types, object classes, objects and object sets.

## 5 Agent creation

In most systems, there are multiple instances of various agents, and some agents are created dynamically, particularly when these are realised as software rather than hardware or firmware. The definition of an agent therefore includes how many initial instances of the agent there will be, and the maximum number of instances. The default is one initial instance and no limit on the maximum, and applies if explicit numbers are not given by parentheses after the name. In Figure 5 (also used in 2), **process** p1 is defined to have 1 initial instance and a maximum of 1 instance, and **block** b2 is defined to have a maximum of 3 instances.

Agents can be created by other agents in a create request (≡ symbol) as part of a transition. One instance of the agent definition identified in the request is created each time the request is interpreted. Values can be passed to the agent in

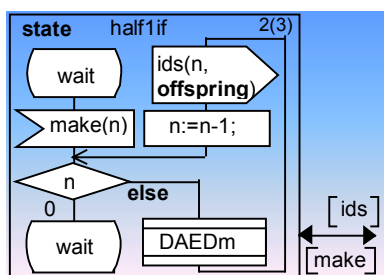


Figure 16: Description of a state machine in a block that creates instances.

parameter variables. A create request can also be used with an agent type, in which case an instance is created as a member of an agent set (implicitly created if one does not exist) of the agent type in the scope surrounding the creator. Creation can be indicated by create line (→ symbol) originating from the creator and with its arrowhead at the created agent.

In Figure 17 the state machine of **block** half1if creates (multiple) instances of *DAEDm* and Figure 16 shows the create request in the state machine. Note that the state machine of the block is represented by a single state symbol, linked to the state diagram in Figure 16.

## 6 State machine diagrams

The state machine diagrams determine the behaviour of Agents. They define what happens in each state and the transitions between states. States are defined by both the □ symbol and the attached symbols such as ∑ describing the handling of stimuli in the state. Transitions are defined by the symbols between the symbols for states, and the next state symbol. Input is part of a state, not part of a transition, though the signal mentioned triggers one.

### 6.1 Stimulus handling

Other symbols that can be attached to a state □ symbol to describe stimulus handling are:

- ▱ save symbol that contains the names of signals that are not consumed in that state;
- < > continuous signal symbol that contains a Boolean expression – if there is no signal that can be consumed and the expression is true, the attached transition is triggered;
- ∑ immediately followed by < > containing a Boolean expression making a signal with an enabling condition – the signal named in ∑ is consumed and the attached transition entered, only if the expression is true (the expression cannot depend on the signal parameters);
- ∑ containing the keyword **none** indicating a spontaneous event – the attached transition can be entered at any time while waiting in the state.

The save ▱ is particularly important, because the channels leading to the state machine determine signals that are valid for all states in the machine. If a signal is not mentioned in any

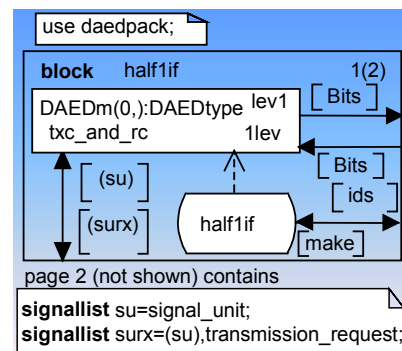


Figure 17: State machine in block that creates instances of an inner block.

of the  $\square$  for the state, it is implied that it can be consumed and there is an empty transition back to the state. Defining a signal as saved in that state by using  $\square$  prevents this from happening.

Whether a signal is saved or consumed is defined for each state independently. If the machine enters a state and a signal is saved in that state, all instances of that signal remain in the input port and are not consumed as long as the machine remains in that state. The next transition is triggered by the first signal instance in the input port that can be consumed (that is, not saved and not inhibited by an enabling condition being false). If the triggered transition goes to a new next state, this next state defines the sets of consumed and saved signals.

## 6.2 Transitions

The components of a transition, such as output ( $\square$ ), task ( $\square$ ) and decision ( $\diamond$ ) seen in 1.2, are called actions. Other actions and symbols that can occur within a transition are:

- $\square$  procedure call – see 6.3;
- $\otimes$  return symbol – see 6.3;
- $\square$  create request – see 5;
- $\square$  raise exception – see 6.6;
- $\circ$  connector – this contains a label.

When a transition ends in a next state  $\square$  that does have any stimulus handling attached, this symbol acts as a connector to the  $\square$  symbol that defines the state. Although, this means that it is possible to avoid connectors, they are sometimes necessary. Out connectors have arrows pointing to them at the end of the flow lines leading to the connectors. An in connector can only have one flow line leading from it. Logically, connectors are a continuation of flow. Figure 18 has an example of a connector and procedure call.

Two other symbols are also introduced in Figure 18, though they can be used generally:

- $\square$  text extension symbol – this can be attached to any symbol and allows continuation of the text inside the symbol. So the task containing the comment */\*generate flags\*/* also logically contains *su\_bits:=flag/su\_bits/flag*;
- $\square$  comment symbol – contains comment text and can be attached to any symbol, such as the initial *transmission\_request* output with the comment *DAEDT -> TXC For first su*;

## 6.3 Composite states

In all the above examples, there has been either no explicit state machine or just one. If no explicit state machine is given for the agent, an implicit one exists. If the agent contains other

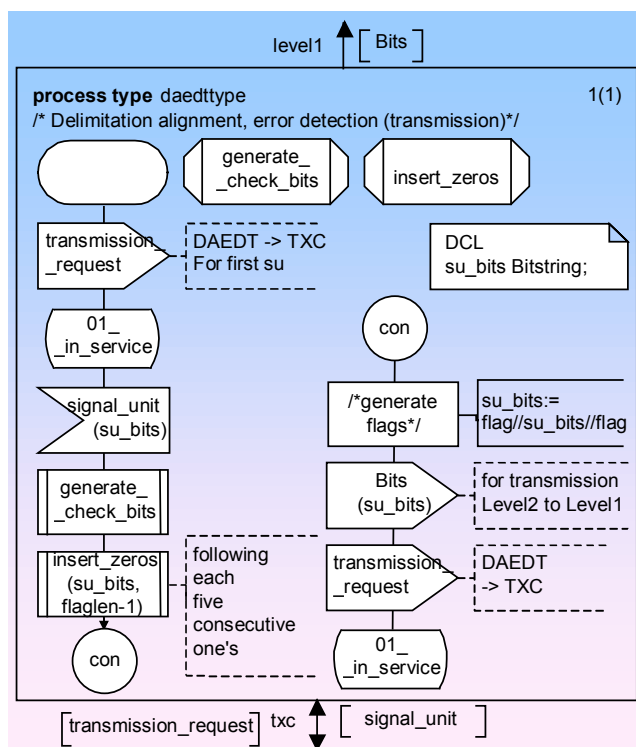


Figure 18: daedtype with connectors and procedure call.

agents, the explicit state machine must be given as in Figure 17 and Figure 16. There can be channels connecting the  $\square$  symbol with the other agents and the environment. The  $\square$  links to a composite state description that can be either a state aggregation diagram, or a state machine diagram.

The state aggregation (see Figure 19) is similar to an agent diagram containing a number of agents, except that it contains  $\square$  links to composite states instead of agents and no channels are allowed. The linked composite state can be considered as a partitioning of the state machine of the agent into state machines that are interpreted in an interleaving manner: only one machine can be in a transition at any one time. When that transition reaches a state node, one of the state machines that can enter a transition is scheduled. If no machine is ready, the agent waits for a stimulus. Each of the aggregated state machines must handle a different set of inputs. An aggregate state only terminates when all the contained states terminate.

A state machine diagram, which is linked from an agent diagram containing other agents, has the same form as an agent state machine diagram. This is the case in Figure 16, and another example (**state ATM**) is given in [9].

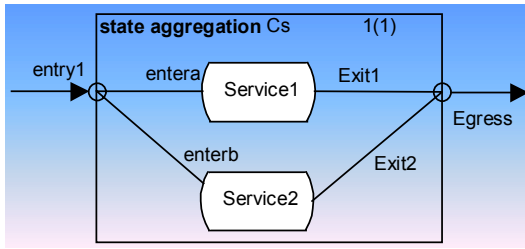


Figure 19: The specification of a machine that is partitioned into two interleaved machines. Cs can be entered without giving an entry point or giving entry1. If no entry point is given both Service1 and Service2 are entered via the start transitions without names. If entered via entry1, Service1 is entered via entera Service2 via enterb. If Service1 terminates at Exit1 and Service2 terminates at Exit2, Cs will exit via Egress. There can be more than one exit, and if the terminations are inconsistent, an arbitrary one is used. Named entry and exit point are only meaningful if Cs is a composite state in a state machine diagram with named entries and exits.  $\rightarrow \circ$  entry and  $\circ \rightarrow$  exit connection points are optional.

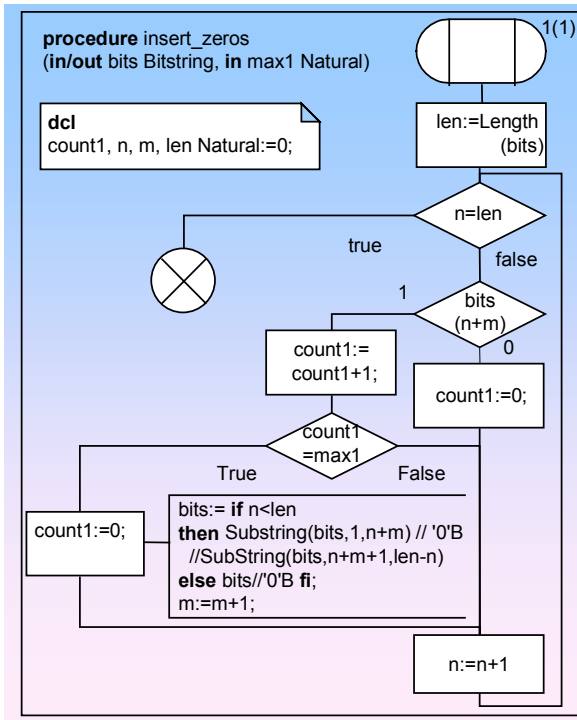


Figure 21: The procedure definition for zero\_bits.

A state machine diagram can also specify composite sub-states of a state in another diagram. Figure 20 and Figure 4 show a simple example. When 02\_Cstate is entered, the process remains in this composite state until either one of the returns is reached, or a Sig1 is received, which forces the sub-state to terminate. Return via the unlabelled return (⊗) takes the unlabelled transition from the 02\_Cstate in composites. Return via the labelled return (⊗) Out1 leads to the transition to 03\_state. If 02\_Cstate is entered via in1 the start symbol containing in1 is used.

As well as state diagrams, state types can also be defined, which allows composite states to be reused many places, like procedures

## 6.4 Procedures

Figure 21 shows a procedure diagram. It is similar to a state machine diagram except that it starts with a procedure start (□) and ends with a return (⊗). The procedure link (□) containing the procedure name shows where it is defined.

A procedure is part of a state machine diagram that is separated out and encapsulated, providing a level of abstraction and a component for re-use. Procedures can have dynamic parameters. A procedure can return a result, and such a procedure can be used in an expression. Procedures can contain states and can be recursive.

A procedure is a type. The calls of the procedure are instances of the type. As well as dynamic parameters for variables, procedures can also have context parameters, for example for signals. A procedure definition can inherit from another definition or can be virtual and redefined in sub-types of the enclosing type.

## 6.5 Textual algorithms

A task (□) contains one or more statements separated by semicolons. These statements are not limited to assignments, and can include:

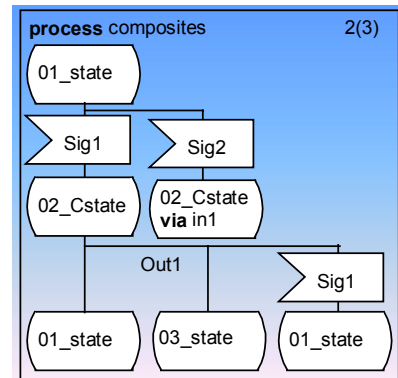


Figure 20: The state 02\_Cstate is a composite state that can be entered via in1 and has an exit Out1.

- compound statement;
- **if** or **decision** statement;
- **for** statement;
- **break** and labelled statements;
- procedure **call** (see 6.3);
- **set** or **reset** action (see 0);
- **raise** statement (see 6.6);
- **export** action (see 7.2);

There are some occasions when graphical description of an algorithm is not the most appropriate form, though this is clearly a matter of opinion. A long and complex procedure without any states might be better written textually. A statement list can be used in a task symbol, as the body of a compound statement, or as the body of a textual procedure or operation in a text symbol. The last three cases all have the list

enclosed in curly brackets `{}`. The *make* and *add* in Figure 15 are defined in this way.

A compound statement (and textual body of a procedure or operation) can have local variables only used in the statement.

An **if** statement takes the form  
**if** (<Boolean expression>)  
    <consequence statement>  
    **else** <alternative statement>;  
where the **else** part is optional.

A decision has the form  
**decision** (<expression>){  
(<range>): <statement>  
(<range>): <statement>  
...  
}

where <range> specifies the constants for the statement to be interpreted. It is exactly equivalent to a graphical decision and there can be one **else**.

The **for** statement does not have one equivalent graphical construct, and therefore is one benefit of using textual algorithms. The general form is  
**for** (<loop variable assignment>,  
    <loop test>,  
    <loop variable step>)  
<controlled statement>

though, for simplicity, options and some alternatives have been omitted here.

If a statement is preceded by a label, the statement can contain a

**break** label

statement, which goes to the label. Note that it is not possible to jump into a statement.

## 6.6 Exceptions

Some checks can only be made dynamically on SDL models. This causes language defined exceptions to occur:

- OutOfRange, when a value is out of the range for a **syntype**;
- InvalidReference, when there is an attempt to use Null to reference an object or a Pid is used in an output to identify a destination process that cannot receive the signal;
- NoMatchingAnswer, when no answer matches a decision value;
- UndefinedVariable, when trying to get the contents of a variable before it has been assigned a value;
- UndefinedField, when trying to get the contents of a field that is undefined;
- InvalidIndex, when an index is out of range;
- DivisionByZero – division by zero;
- Empty – trying to *take* an element from a set (created using a type derived from Powerset) that has no elements.

Handlers can be provided for these exceptions and for user defined exceptions. If the exception occurs and is not handled locally in a procedure, operation or compound statement that item is terminated, and the exception can be passed to the point of invocation in the caller.

An exception handler can be defined in an agent, agent type, procedure or operation. The exception handler ( $\langle \boxtimes \rangle$ ) symbol contains the name, and handles one or more exceptions whose names are in handle ( $\langle \Sigma \boxtimes \rangle$ ) symbols attached to an exception handler  $\langle \boxtimes \rangle$  symbol by lines.

An on exception ( $\rightarrow$ ) has its arrowhead connected to a  $\langle \boxtimes \rangle$  symbol containing a name. Like a next state symbol, this  $\langle \boxtimes \rangle$  symbol may be a connector to the actual definition of the handler, or may be the head of the handler. The other end of the on exception ( $\rightarrow$ ) is either not connected, in which case the handler applies to the whole diagram, or is connected to a specific symbol (such as a start, or state or input) in which case it applies until the end of the transition. An exception attached to an action applies just to that action.

```
exception e1, e2;
```

in a text symbol defines user exceptions *e1* and *e2*.

An exception can be explicitly raised by a **raise** ( $\langle \boxtimes \rangle$ ) containing the exception name. This terminates a transition, and no symbol can follow it.

Figure 22 gives an example. *check\_bits\_correct* has the heading

```
procedure check_bits_correct -> Boolean; raise su_error;
```

## 7 Communications

Communication between agents takes place by signals (see 2.1), remote procedures and variables.

### 7.1 Communication using Remote procedures

One agent can communicate with another agent by a remote procedure mechanism, so that the calling agent waits for a response from the called agent. The agent that offers the communication defines the procedure in the normal way, but

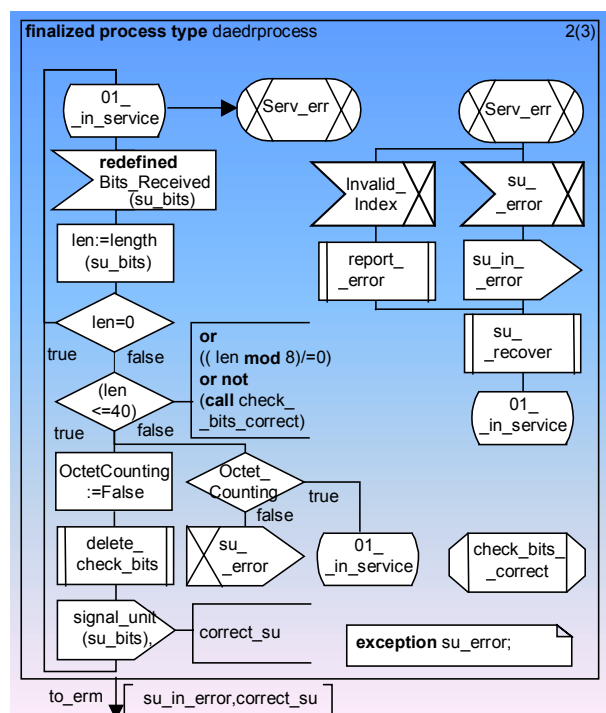


Figure 22: finalized daedrprocess with exception handling.



with **exported** in the heading, such as

```
exported procedure rp(in x xsort)->rsort;
```

There are some restrictions on the parameters and return values of remote procedures. In a scope or interface common to both the called and calling agents, a definition is given

```
remote procedure rp (in xsort)->rsort;
```

The procedure call can be written in the caller in a similar way as any other procedure, but with the added possibility to specify the destination of the call and a timer on the response.

```
myx:= rp (myx) to parent timer ttp;
```

## 7.2 Communication using variables

One agent in a block cannot access the variables of another agent in the same block or any other block. However, there is a notation for exporting the value of a variable from one process to another. If the owning process defines the variable as exported (by **dcl exported** x xsort;), it can have an **export** action that copies the variable value. In a scope or interface common to both the exporter and importer, a definition is given (**remote** x xsort;). The importer can invoke **import** expression to the exporter (myx:= **import** (x);) and obtain the copied value. In this way, the value is safely under the control of the exporter.

A variable of an enclosing agent that has its **dcl** definition directly visible to an enclosed agent, can be read or written by either agent without the need to define a remote variable.

Where the enclosing agent is a process, no special mechanisms are needed to access the variable. This is because the scheduling of state machines within the process is alternating at the transition level, no two state machines can be accessing the variable at the same time.

Where the enclosing agent is a block, the scheduling of state machines within the block can be interleaved at the action level. To ensure safe access to the shared variables of the block, these are accessed by implicit remote procedures of the state machine of the block.

## 8 Learning more

The SDL-2000 Recommendation is 200 pages of concise information and is probably only suitable as a reference document. Obviously, a short article such as this one cannot be comprehensive. At the time of writing no tools have been released and no books has been published, and as far as the author knows this is the first tutorial style article to be published.

However, by the end of the year 2000 the author expects the situation to changed, and the best way of getting up-to-date information on SDL is to access <http://www.sdl-forum.org>.

## References

1. Z.100 (11/99) Specification and Description Language (SDL), ITU-T, 2000.
2. Z.100 (03/93) CCITT Specification and Description Language (SDL), ITU-T, 1994.
3. Bræk, R., et. al., TIME – The Integrated Method version 4.0 – TIME at a glance, SINTEF.
4. Z.120 (11/99), Message Sequence Chart (MSC), ITU-T Geneva, 1999.
5. Haugen, Ø., MSC-2000: interacting with the future, *Teletronikk, this issue*.
6. Z.109 (11/99) SDL combined with UML, ITU-T Geneva, 2000.
7. Willcock, C, A Tutorial Introduction to ASN.1 97 new features and language evolution, *Teletronikk, this issue*.
8. Z.105 (03/95) SDL Combined with ASN.1 (SDL/ASN.1), ITU-T Geneva, 1995.
9. Møller-Pedersen, B, SDL Combined with UML, *Teletronikk, this issue*.