Introduction to SDL-92

Ove Færgemand and Anders Olsen,
TFL, Lyngsø Allé 2,
DK-2970 Hørsholm, Denmark
e-mail: ove@tfl.dk, anders@tfl.dk

July 27, 1992

Abstract

This paper contains an introduction to CCITT SDL - Specification and Description Language as it appears in a draft revised CCITT recommendation. The technical work on the draft has been completed and the draft is expected to be recommended by CCITT, at the CCITT Plenary February 1993. The paper covers main aspects of the new version of SDL in three main areas: behaviour, data and structure. The paper covers in particular the new features for object-oriented structuring in the language. The paper is concluded with an overview of current activities in standards, research and industry.

Contents

1 Evolution of SDL

SDL (CCITT **S**pecification and **D**escription **L**anguage is recommended by CCITT for describing functions etc. of telecommunications systems. It is based on experience of describing telecommunications systems as communicating state machines in the telecommunications industry and CCITT. Since 1976, SDL has been maintained by CCITT, and has evolved from an informal drawing technique to a formal description technique. The most recent recommendation is [1]. Due to its simple conceptual basis and its choice of graphical or textual representation, SDL has maintained its original informal flavour, even though today commercial tools exist [2] which allow to generate code directly from SDL descriptions. The most popular representation form is the graphical one, which it is used in this paper. (Note: for reasons of simplicity, some of the figures do not include all details of the enclosing diagram).

The more recent extensions to SDL are in the area of object-orientation. They are included in SDL-92 [3]. SDL-92 introduces a clear distinction between *types* and *instances*, *specialization* of types into subtypes, and the concept of *generic types*. These concepts are described in 4 below. While SDL-92 has been extended in the area of object-orientation and includes a few other improvements to SDL-88 [1], it is backwards compatible[1]. This strategy protects investments in education, tools and applications based on SDL-88.

2 The Process Model

2.1 Basic Communication Model

A system is modelled in SDL as a number of concurrent, communicating *process* instances interchanging *signal* instances with each other and with the environment of the system. Each process instance is an extended finite state machine. Signals convey the identity of the sending process instance and may convey data values from the sender to the receiver. In case of explicit addressing of a signal (see 2.3.5), the signal also conveys the identity of the receiving process instance. The information content in a signal is illustrated in fig. 1[2]. .

		data items
sender	[receiver]	*0... n*

Figure 1: Information content of signal

Each type of signal is introduced in a signal definition. As an example, the signal ready used in fig. 2 has this signal definition:

signal ready (Integer, Boolean);

The communication is based on asynchronous passing of signal instances from one sender to one receiver (like the postal mail service). This implies that the only temporal ordering derivable from a communication is that the sender has sent a signal prior to the reception by the receiving process instance. During the evolution of SDL, the communication scheme of the language has often been discussed. It seems that the loose, buffered coupling implied by asynchronous communication is a good model for building large systems and for taking an object-oriented view of a system. In addition it fits well with actual possibilities for observation of telecommunications systems. The more advanced techniques for verification and testing work can more easily be applied to system models based on synchronous communication. However some of the techniques, have been adopted to SDL (see 5.2).

Each process instance has an unlimited buffer, *input port*. When a signal arrives at the receiving process instance, it is stored in the input port of the instance. Once the process reaches a state, it will request the next signal in the port. The asynchronous communication is thus realized by adding this input port buffer to each process instance.

It is possible to retain certain signals in the input port dependent on the actual state. This construct is called *save* and is elaborated in 2.3.2.

2.2 Process Types and Instances

A process type defines a template which can be used to define *sets of process instances*. Each set of process instances (object) is defined in a certain context of a system. Individual *process instances* can be *created* to be members of a process instance set.

2.2.1 Process Types

A process type definition consists of:

- Gate definitions,
- Specialization clause,
- Formal parameter definitions,
- Enclosed definitions,
- Process body.

An example of a process type diagram is shown in fig. 2. This process type `p1`, has a gate `g`, a formal parameter (**fpar**) `x` of data type `Integer`, an enclosed definition of a variable `s`, and a process body (a directed graph). The process body defines the behaviour of the process type utilising the information provided in other parts of the process type definition. The definition of some SDL-constructs has no graphical representation, therefore these definitions are shown inside *text symbols* in the diagrams. In fig. 2, the variable definition is shown inside a text symbol.

The gates are the interface of a process type. A gate has a name and is characterized by the signals it allows in a direction. The gate `g`, in fig. 2 allows the signal `event` to `p1`, and the signals `response` and `ready` from `p1`. Gates may have endpoint constraints, indicated in a box at the end of the lines connected to the gate. In this example the endpoint constraint implies that the gate `g` in all process instance sets based on `p1` can only be connected to a communication path leading to an instance set based on the process type `p2`.

[1]except for two minor cases: implicit routing of signals without explicit address and qualification in view and import expressions

[2]The notation used in the figure is not SDL

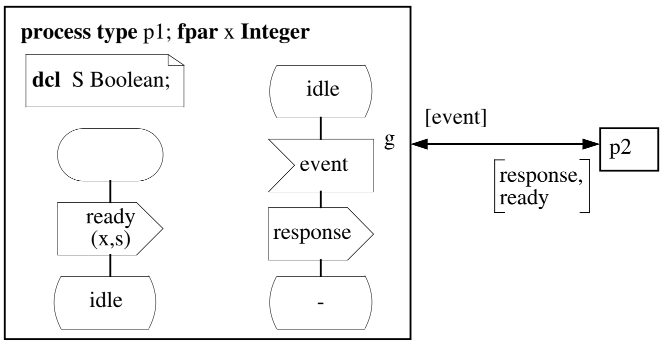


Figure 2: Example process type diagram

More elaborated examples on the use of gates and specialization are provided in 4.

When a process is created, actual parameters can be passed to it. The formal variable parameters assume the value of the actual parameters before interpretation of the process instance begins.

A process definition may contain enclosed definitions. These can be definitions of services (see 4), procedures, signals, timers, data types and variables. Variables can only be defined inside processes, services and procedures and are only accessible within the same process instance, service instance or procedure.

The behaviour of a process is defined by its process body. The process body consists of a start-transition, and a number of states with associated triggers and state-transitions. The beginning of the process body is indicated by a start symbol (the empty symbol below the text symbol in fig. 2). In a state, a process instance is waiting for a trigger to leave the state. When the process instance leaves the state, it interprets a state-transition. After the state-transition, the process instance enters a new state, this is referred to as *nextstate*. The nextstate-symbol is the same as the state symbol, and if convenient, the symbols can be merged.

The process in fig. 2 outputs the signal ready in its start-transition, thereafter it enters a state `idle`, where it waits for a signal `event`. The input of this signal triggers the process to interpret a transition which consists of the output of the signal `response`. After output of this signal, the process (re)enters the same state (`idle`). Re-entering the same nextstate as the originating state, is indicated by a - in the nextstate symbol.

The contents of the process body is elaborated in 2.3.

2.2.2 Process instance set

Fig. 3 shows an example of a process definition based on the process type stated in fig. 2. The two numbers `(1,3)` denote that the process instance set has 1 instance when the system is initiated, and that there can exist at most 3 simultaneous instances in the process instance set. Either number is optional. If the initial number is omitted, the default value is 1. If the maximum number is omitted, there is no limit on the number of simultaneous instances.

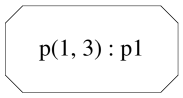


Figure 3: Process reference definition

In addition to the initial creation of process instances indicated in the definition of a process instance set, process instances can be created dynamically, by specifying a *create* action in a transition. This action may also transfer actual parameters to the created process instance. Fig. 5 shows an example of a create symbol, for dynamic creation of an instance in the process instance set `p`.

Process instances are identified using a special predefined data type, called `PId` (Process Identity). The `PId` data type includes a special value, `Null`, which never denotes a process instance. Each process instance is identified by a unique `PId` value. There are four predefined `PId` expressions available in each process instance, useful for communication with other processes:

- **self** - denoting the process instance itself
- **sender** - denoting the process instance from which the process instance has most recently input a signal
- **parent** - denoting the process instance which created the process instance
- **offspring** - denoting the process instance most recently created by the process instance

From this follows, that **self** and **parent** are fixed for the whole lifetime of a process instance, whereas **sender** and **offspring** may vary.

If a process instance has been created as part of system initialization, `parent` is `Null`. If it has not yet input any signals, `sender` is `Null`. If it has not yet created any process instances, `offspring` is `Null`.

2.3 Basic Process

2.3.1 Process instance lifetime

When a process instance has been created, the local variables can be initialized and interpretation of the process body starts by interpreting the state-transition following the start symbol. To indicate that a process instance has completed, a stop symbol is specified. When it is interpreted, the process instance ceases to exist. A minimal process body consisting of a start and a stop symbol is shown in fig. 4.

The example in fig. 5 has been extended so that it - as its only action - creates another instance of its own process instance set `p`, before it stops.

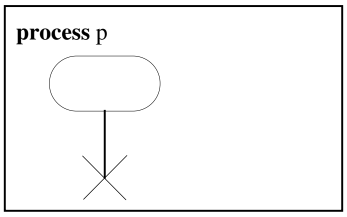


Figure 4: Start and stop symbol

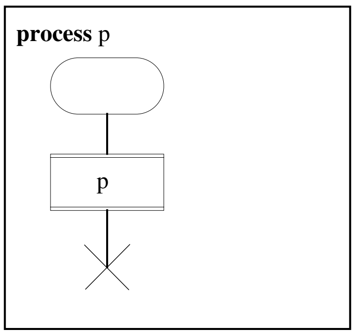


Figure 5: Dynamic process creation

2.3.2 State and triggers

In a state a process instance basically waits for new signals to arrive. A process instance can be activated by reception of any signal in its *valid input signal set*. This set can either be specified directly or be derived from incoming signals on gates or on signal routes (see 4).

Signals can be retained in the input port in a certain state by mentioning them in a *save*. These signals are then available for input in consecutive states. This allows some priority on signal handling in a receiving process, e.g. if a signal is not considered relevant in a certain state, it can be mentioned in a save, and thereby the handling of it be postponed until a later state.

In some cases, all signals which are not dealt with explicitly, can be treated in the same way, e.g. as a default case. For this purpose an asterisk can be placed inside an input symbol. This is a shorthand for denoting all signals not mentioned directly in a state.

In other cases, it is convenient that all inputs not handled directly are saved, thereby building a "persistent" queue. For this purpose, an asterisk can be placed inside a save symbol. It is not allowed to use this shorthand and the input asterisk shorthand in the same state.

If a signal in the valid input signal set is not mentioned in an input or in a save and the state has no shorthand of asterisk input or asterisk save attached, the signal is considered to be implicitly received with no change in the state. That is, the input port only retains all not mentioned signals, if the asterisk save is used.

An input symbol mentions a signal name and a list of variables. If the signal is received, the variables assume the values conveyed by the signal, in addition to the PId expression **sender** assuming the PId value of the process which sent the signal. If a certain position is not filled in with a variable, any corresponding value conveyed by the signal is lost.

The example in fig. 6 shows a state s, where the signal d is saved, and the signals b and c are received. b conveys two values which are stored in the variables x and y. c conveys two values as well but only the second one is stored, in the variable y.

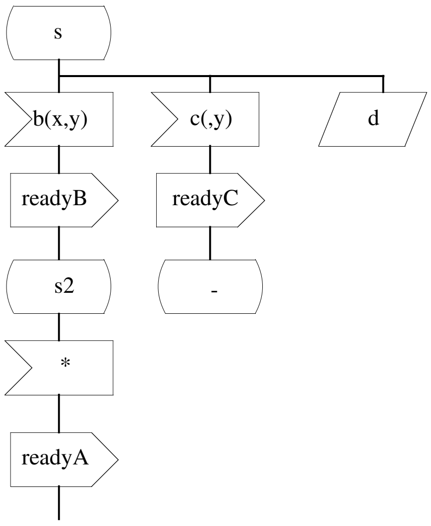


Figure 6: Example of input and save

Assume that the input port holds this sequence of signals, when the process enters the state s:

$$< \texttt{d, b, c} >,$$

where d is the first signal in the input port. Since d is saved in state s, the process will receive the signal b, and interpret the state transition following the input b, that is output readyB. In the next state, s2, where d is included by the asterisk, and not saved, it will be first in the queue and consequently received. That is, the process will output readyA, and then the rest of the transition which is not shown.

In some cases, it is convenient to express, that a certain input or save should be attached to all states of a process. For this purpose, an asterisk can be placed inside a state symbol.

To complete the list of shorthands which makes the writing of the basic state and input symbols easier, it is allowed to write a number of states in the same state symbol, a number of signals in the same input symbol and a number of signals in the same save symbol respectively. These notations expand into a list of state, input and save symbols each containing one item.

2.3.3 Unstable States

A spontaneous transition allows a state transition to be triggered non-deterministically irrespectively of whether there are any signals in the input buffer. This is expressed by writing the keyword **none** in an input symbol.

Fig. 7 shows an example of a state `in_service`, where a process can input a signal `b` and pass it on as signal `a` with same value `x` conveyed. This could be a model of a very trivial protocol machine. However, the state `in_service` also has a spontaneous transition leading to the state `in_error`. To leave this state, the input `fixed` is needed. The process is really a model of an unreliable system.

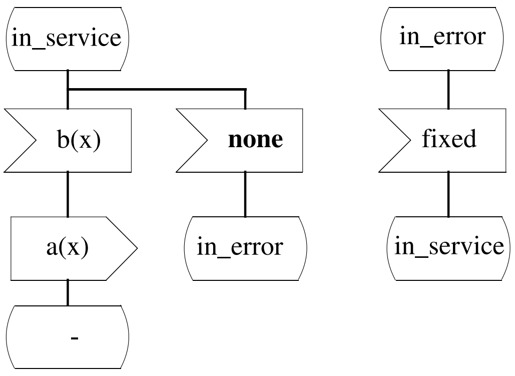


Figure 7: Example of state with a spontaneous transition

Spontaneous transitions are convenient for modelling unreliable system components. In verification of protocols, an explicit model of a communications medium is often inserted between the protocol endpoints to have a formal link between the endpoints. In this context, spontaneous transition is useful for describing different kinds of unreliable communication media (the one in fig. 7 models possible loss of signals).

2.3.4 Conditional Waiting in a State

A continuous signal allows a state transition to be triggered by the fulfillment of a certain condition rather than by signal reception. Fig. 8 shows an example of a state `s` with input of a signal `b`, and a continuous signal with the condition: `x=7`.

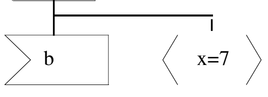


Figure 8: State with input and continuous signal

This notation is extremely powerful when combined with some of the imperative operators (see 3). An imperative operator returns a value which both depends on its actual parameters and the global state of the system. The use of continuous signal can even lead to a "rule-based" style where state transitions (connected to an asterisk state) are triggered based on complex conditions rather than on the process being in a particular state.

2.3.5 Output

A most important feature of a state transition is the ability to output signals, indicated by an output symbol. It contains the name of a signal with possible actual parameters, followed by a possible address and a possible path which must be followed when the signal is output. Fig. 9 shows an output symbol for sending the signal `response` with actual parameter `True`, where the address information is `S` and the path `g` is the name of a gate (assuming the symbol appears inside a type). The address and path are both optional.

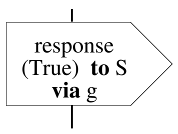


Figure 9: Example of output

The address information may be the identifier of a process instance set or it may be a `PId` expression (e.g. **sender**, to return an answer to the process from which a signal was recently input). If an identifier of a process instance set is stated, the signal is delivered to an arbitrary instance of this set, once it is transported to the process instance set. Of course, in the special case of only one member in the process instance set, the signal will be sent to this process instance.

If no address information is given, the signal delivery is to a member of a process instance set, which includes the signal in its valid input signal set, and which is reachable from the sending process instance according to the system structure (see 4) and the possible path stated.

The path which is given in a **via** clause restricts the choice of communication path for the signal to those paths mentioned in the clause. A communication path expresses a possibility for signal passing in a complete system description. It is further described in 4.

The addressing scheme for outputs with no addressing has been slightly changed as compared to SDL-88. This has been done, because the guarantees of reliable signal delivery imposed by SDL-88 were found complicated to interpret in a distributed context.

2.4 Extending the Finite State Machine

The concept of finite state machine has been extended for processes to

- maintain variables, whose values can be assigned in assignment statements or as part of signal input, and to
- use the values of variables to control branching between different alternatives in a state transition.

2.4.1 Variables and expressions

Variables can only be declared within processes and procedures. Each variable can only be accessed directly within a single process instance. The only way to change a variable in another process instance is by sending a signal to that process, and having agreement with the owning process about updating the variable.

A variable in SDL is declared to be of a certain datatype. The concept of data type in SDL is described in 3. The variable is declared in a text symbol of the enclosing procedure or process diagram, as shown in fig. 2.

2.4.2 Use of data inside the process

Variables are manipulated locally by tasks and their values can be used for branching in decisions.

The task symbol is a rectangle. A task can be formal or informal. A formal task is an assignment statement, whereas an informal task contains informal text, which has no formal semantics.

The decision symbol is a diamond. A decision consists of a question written inside the symbol and an answer on each exit branch of the symbol. A decision can be formal or informal. A formal question is an expression of some data type. A formal answer expresses a range of values of the same data type. Formal and informal questions and answers may be mixed freely.

The use of informal text allows to sketch part of internal behaviour before final data design of the process has been done. Informal text is enclosed by apostrophes.

The example in fig. 10 shows a formal assignment `x := 5` followed by the informal task `'x is 7'`, followed by decision with an informal question `'x=5'` followed by two branches with (formal) answers: `True`, `False`. The formal interpretation of this sequence is that `x` holds the value 5, but the decision is not formally based on this value, since the question in the decision is informal. Therefore, the branching of the transition can only be informally interpreted.

2.4.3 Undecided values and decisions

Sometimes it is convenient to specify that a value of a data type can be an arbitrary value, e.g. it can be part of a protocol to state that an arbitrary Integer value is passed twice with a protocol data unit to check correct

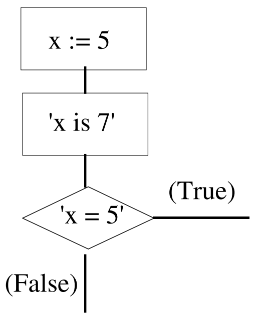


Figure 10: Example of task and decision

transmission. This can be modelled using the **any** operator. The **any** operator takes as argument the name of a data type, and it returns an arbitrary value of this data type.

Based on this construct, an undecided decision is introduced. An undecided decision contains the word **any** in a decision symbol, and no answers on the exits. When interpretation reaches this symbol, an arbitrary exit will be chosen. This can be used to model non-deterministic aspects of behaviour as a supplement to spontaneous transitions.

2.5 Procedure

SDL offers a usual procedure concept, as a convenient shorthand for writing parts of processes which are used several times. A procedure may have local variables and formal parameters.

When a procedure has been called, its local variables are possibly initialized, and interpretation of the procedure body starts by interpreting the state transition following the procedure start symbol. To indicate that a procedure has completed its mission, a return symbol is specified. When it is interpreted, the procedure activation ceases to exist, and control is returned to the calling body. A minimal procedure body consisting of a procedure start symbol and a return symbol is shown in fig. 11. If a procedure returns a value, the expression of the value is shown next to the return symbol.

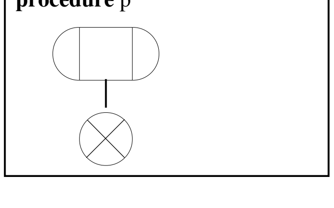


Figure 11: Procedure start and return

A procedure uses the input port and assumes the same valid input signal set for its states as the process in which it is called.

A procedure is a type in itself, and as such can be based on another more general procedure by specialization (see 4).

2.6 Implicit process communication

The basic communication scheme in SDL is signal passing, but for some purposes, other communication schemes may be more appropriate.

SDL-88 allowed for two alternative ways of reading variables in another process: **view** and **import**. View is mainly included in SDL-92 for reasons of compatibility with older versions of SDL and its use is not advised. For new SDL descriptions the exported procedure concept which has been introduced in SDL-92 should be used to get the value of a variable in another process.

In this section, exported (imported) variables and procedures are described. Both schemes are shorthands, which rely on signal passing for their formal semantics.

2.6.1 Exported variable

If a variable has the exported attribute, other process instances than the one where it is declared may read its value. Other processes wanting to read the value, must introduce it locally in an import definition. The (one) exporter and the (one or several) importer definitions are related by all referencing the same remote variable. The value available in the importing process is the value of the exported variable at a certain place, where the exporting process contains an **export** statement. Fig. 12 shows an example of two processes communicating via export/import. In the exporting process, the **exported** attribute in the variable declaration (**dcl**) makes the value available for export. In the importing process, the value is made available via an import-expression. Note that the import-expression is used in a continuous signal to trigger a transition.

2.6.2 Exported procedure

The concept of exported procedure is a generalization of the exported variable concept, mentioned above.

If a procedure has the exported attribute, it may be called from other process instances than the one where it is defined. Other processes wanting to call the procedure, must introduce it locally in an import definition. The (one) exporter and the (one or several) importer definitions are related by all referencing the same remote procedure. The call of an exported procedure is transformed to a signal interchange for parameter- and result-passing and a local call of the procedure in the exporting process instance. The calling process will wait in an implicit state

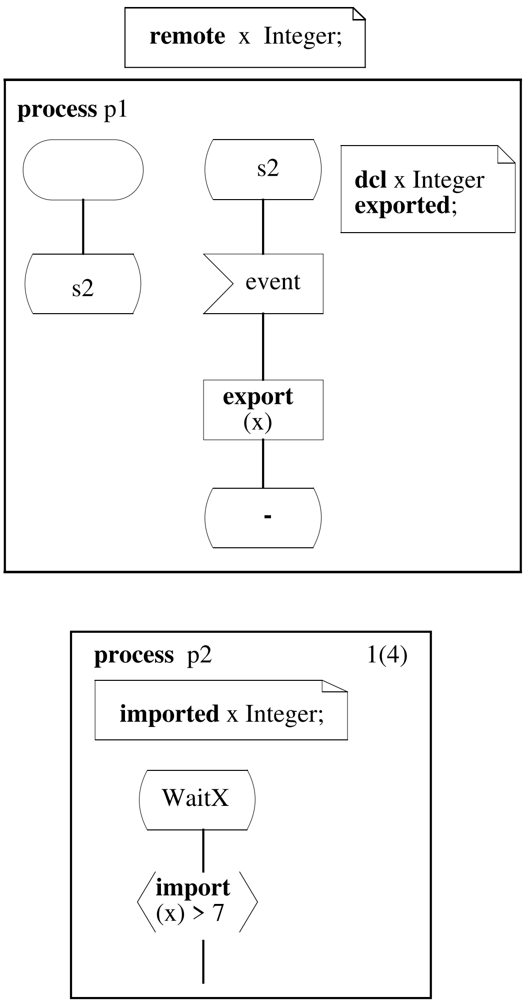


Figure 12: Example of export and import

until the called procedure returns a signal that the call is completed.

A remote call of an exported procedure can be indicated as a state-trigger in the exporting process instance. To specify that the remote procedure is not served in a certain state (serving is postponed), **remote** followed by the procedure name in a save symbol is used. To specify a transition after the local call of the procedure **remote** followed by the procedure name in an input symbol is used.

Exported procedure provides the facility known as remote procedure, e.g. in the OSI-framework for data communications (including network management standards).

Fig. 13 shows an example of a process `p2` with a procedure call of a procedure `p`, which is exported by process `p1`. If `p` is called when `p1` is in state `s1`, a local procedure call of `p` will be made, the result of calling `p` will be returned to `p2`, and `p1` will assume the same state (`s1`). If `p` is called when `p1` is in state `s2`, the local call will be postponed by retaining the activating, implicit signal in the input port. The diagram for procedure `p` is not drawn inside the diagram for process `p1`, rather a procedure reference symbol (see 4) is used.

2.7 Modelling time

2.7.1 Time values

Two predefined data types, `Time` and `Duration`, are available for stating time values. A global actual time can be accessed via the imperative **now** operator. Since SDL models a distributed system, no assumptions on temporal ordering of events in different processes should be based on reading **now**, only on (direct or indirect) communication between the instances.

SDL has no notation for scale of time. Rather, it must be stated in some comment or outside the SDL-description, which scale of time is assumed.

2.7.2 Timers

Time-dependencies can be modelled by means of *timers*. A timer is a watch which can be set with an expiration time. When this time is reached, the timer expiration is signalled to the process instance as an ordinary input. When a timer is no longer needed, it can be reset before its expiration, to avoid unexpected expirations.

Fig. 14 shows an example, where a process uses a timer, `t` to supervise that a signal `late` arrives within 14.5 time units from the current time (**now**). If the signal does not arrive, `t` expires. If a signal, `late` arrives before `t` expires, `t` is reset.

Timers are e.g. useful to model supervision of unreliable media.

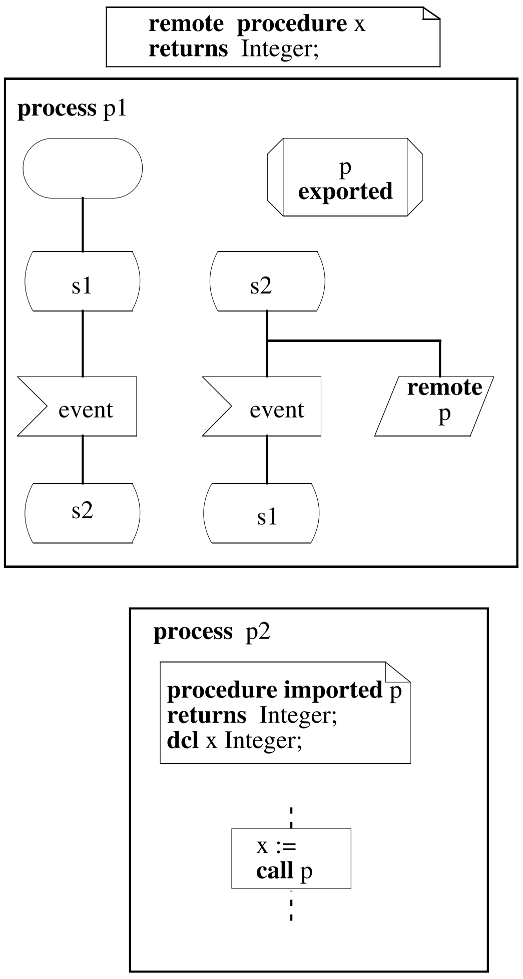


Figure 13: Example of remote procedure call

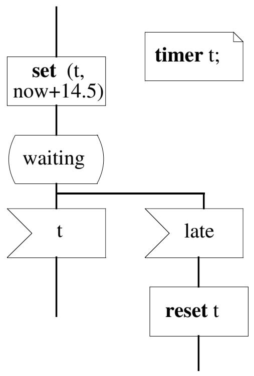


Figure 14: Example of timer

3 Definition of Data

2.1 shows examples of how variables and expressions can be used to influence the behaviour of process instances. Specifically, the behaviour is influenced through manipulation of *values* in a process instance:

- Values are used to determine the answer branch in decision
- Values may be conveyed by signals to other process instances in output
- Values may be passed in procedure call and in dynamic process instance creation
- Values are used to determine the expiration time when setting timers

In many ways, the use of data in SDL resembles use of data in programming languages:

- Expressions evaluate to values;
- A value belongs to a certain data type;
- Variables are used for storing values for later use. A variable can only hold values of a certain data type;
- Predefined data types are available (e.g. Integer) and new data types may be defined;
- Each data type defines names for specific values, e.g. the number `'7'` is a name defined in the Integer data type. The names for values are in SDL called *literals*. Each data type also defines *operators* performing operations on values, e.g. `'+'` is an operator for addition of Integer values.

A data type thus has a strict interface which is the literals and operators for the data type. Data types are defined in text symbols (like for variable definition, see fig. 2).

The predefined data types are listed with their literals and operators in fig. 15.

The way data types are defined in SDL and especially the way the properties of literals and operators are defined, differs considerably from the programming language approach.

Consider for example the definition of the predefined Integer data type:

```
newtype Integer
 literals 0,1,2,3,4,5,6,7,... etc[3];
 operators
   "-"   : Integer,  Integer  ->  Integer;
   "+"   : Integer,  Integer  ->  Integer;
   "-"   :           Integer  ->  Integer;
   "*"   : Integer,  Integer  ->  Integer;
   "/"   : Integer,  Integer  ->  Integer;
   "<"   : Integer,  Integer  ->  Boolean;
   "<="  : Integer,  Integer  ->  Boolean;
   ">"   : Integer,  Integer  ->  Boolean;
   ">="  : Integer,  Integer  ->  Boolean;
   Float :           Integer  ->  Real;
   Fix   :           Real     ->  Integer;
     /*Here the behaviour of the operators is defined*/
endnewtype;
```

[3] A special construct (not described or used here) exists for defining an infinite number of literals.

First, literals of the data type are defined. Then follows the definition of operators and finally the behaviour of the operators are defined (not shown in the example).

Each operator has a *signature*, i.e. a argument data type and a result data type. For example, the "+" operator takes two values of Integer data types as arguments and returns an Integer value as result. The operators for a data type need not be distinct, but operators with same name must have different signatures (e.g. the "−" operator for Integer).

An operator is used by specifying the operator name followed by the arguments enclosed in parenthesis (e.g. "+"(2,3) and Fix(1.2)). However, SDL allows the common names for arithmetic and relational operations to be applied in infix form (e.g. "+"(2,3) can be written as 2+3).

Often, operators or literals with the same name are defined for several data types. For example, the "+" operator is defined for the data types Integer and Real.

In addition, every data type has implicitly the "equal" operator ("=") and the "not equal" operator ("/=") defined .

In most cases, it can be determined from the context which operator or literal a name denotes (e.g. the name "+" in the operator application "+"(1.2,3.4) denotes Real addition since the arguments are Real values). For the rare cases where it is not possible to determine the operators and literals from the context, the language provides a means for qualifying names with the data type in which it is defined.

3.1 Behaviour of Operators

Often a new data type is solely introduced for the purpose of introducing new operators working on existing data types.

Consider for example that a new operator `count`, which counts the number of occurrences of a given character in a string, is to be defined.

Then a new data type defining the operator has to be defined:

Name	Literals	Operators
Boolean	True, False	**not**, **and**, **or**, **xor**, =>
Char	character enclosed by ' '	<, <=, >, >=, Num, Chr
Integer	0, 1, ..	−, +, −, ∗, /, <, <=, >, >=, Float, Fix
Real	0, ... 1, .., 107.86, ..	−, +, −, ∗, /, <, <=, >, >=
PId	Null	none
Duration	as Real	+, −, >, ∗, /
Time	as Real	+, −, −, <, <=, >, >=
Charstring	characters enclosed by ' '	MkString, Length, First, Last, //, (*index*), SubString(*string*, *start position*, *length*)

Figure 15: Predefined data types

```
newtype Stringutility;
  operators count : Charstring, Character -> Natural
  /* Behaviour of count operator */
endnewtype Stringutility;
```

As there are no literals or operators having Stringutility as result type, the data type contains no values. However, it defines the semantics of the count operator.

In general, there exist four approaches for specifying behaviour of operators:

- As informal text
- As axioms (equations) specifying equivalence of expressions
- As actions like the behaviour of processes
- As externally defined data

It is allowed to mix the four approaches.

In the following, the approaches are illustrated using the Stringutility data type.

3.2 Informal Approach

Specifying the behaviour informally is useful during the early development phases. The behaviour can then later be refined into one of the other approaches.

Informal definition of operators is indicated by the keyword **axioms** followed by informal text (enclosed in apostrophes):

```
newtype Stringutility;
   operators count : Charstring, Character -> Natural
   axioms
   'count(str1,char) =
    count the number of occurrences of
    char in str1'
endnewtype Stringutility;
```

3.3 Axiomatic Approach

In the axiomatic approach, semantics is given by specifying equivalence of expressions, i.e. for the count operator, it is specified which natural number any particular combination of arguments corresponds to. The equivalence of expressions is expressed through a number of *equations*, each containing a left-hand expression, the symbol "==" and a right-hand expression. Equations may be *quantified.* A quantified equation introduces a number of typed names which in the equation denotes any value of the given type. This means that the left-hand side expression and right-hand side expression are equivalent if the same value is substituted for every occurrence of the name. The actual value chosen makes no difference, since the two sides are equivalent for all values.

The axiomatic definition of the data type Stringutility is given below:

```
newtype Stringutility;
   operators count : Charstring, Character -> Natural
   axioms
     for all ch1,ch2 in Character
      (for all str in Charstring
        (count('',ch1) == 0;
         count(mkstring(ch1)//str,ch2) ==
         count(str,ch2) + if ch1 = ch2 then 1 else 0 fi))
endnewtype Stringutility;
```

The definition contains one quantified equation which contains another quantified equation. The outer quantification introduces two names ch1 and ch2 of type Character and the inner quantification introduces a name str of type Character. Each of the three names independently denotes any value of their given data type in the two contained equations. The first of the two contained equations express that counting the number of occurrences of a Character in the empty string always yields 0. The second equation expresses that when a character is concatenated to a string, that character should also be taken into account. (mkstring converts a Character to a string and the infix operator "//" concatenates two strings. These operators are defined in the Charstring data type).

Note the use of a *conditional expression* enclosed in **if** and **fi**.

The foundation of the axiomatic approach is called ACT-ONE[4].

3.4 Specifying the behaviour Algorithmically

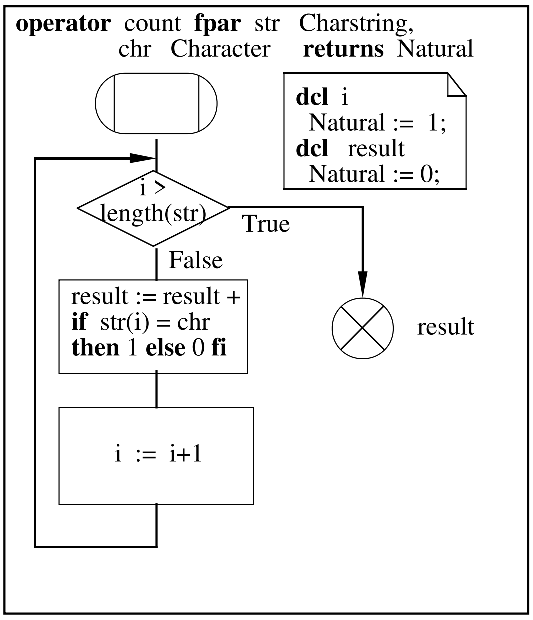
In the algorithmic approach, the operator behaviour is specified in an *operator diagram* using a transition like for a process. However, in an operator diagram, it is not allowed to manipulate the global state, such as process variables, and input. The algorithmic approach is more "implementation oriented" than the axiomatic approach and it is often found easier to use because it resembles specification of processes. However, it has less expressive power than the axiomatic approach (e.g. "basic" operators like extracting a Character from a Character string cannot be defined algorithmically).

The definition of the data type Stringutility with the operator count defined algorithmically is given below:

newtype Stringutility;
  operators count : Charstring, Character -> Natural
operator count **fpar** str Charstring;chr Char
  returns Natural;
    referenced;
endnewtype Stringutility;

The data type definition contains an *operator reference* which is the textual counterpart to the reference symbol described in 4.

The operator diagram for the count operator is given below:



operator count fpar str Charstring, chr Character returns Natural
dcl i Natural := 1;
dcl result Natural := 0;
i > length(str)
True
False
result := result + if str(i) = chr then 1 else 0 fi
i := i+1
result

Figure 16: An operator diagram for the count operator

3.5 Specifying the behaviour in an alternative data formalism

This approach is useful when interfacing SDL with another language such as the data specification notation ASN.1 (Abstract Syntax Notation 1) or when translating SDL to a programming language, thus taking advantage of the features offered by a specific language. The alternative notation is enclosed in the keywords **alternative** and **endalternative**. It should be noted that, from the SDL point of view, the construct is considered as informal text, i.e. SDL does not currently provide any formal relation to other data formalisms.

An example of interfacing with the programming language C is given below.

newtype Stringutility;
operators count : Charstring, Character -> Natural
alternative C
#include "stringutil.h"
#include "stringutil.c"
endalternative;
endnewtype Stringutility;

3.6 Convenient shorthand notations

3.6.1 Generator

A *generator* is an incomplete data type definition that can be parameterized with data types to form a complete data type definition.

The use of generators will be described only for the predefined data type generators shown in figure 17.

Name	Parameters
String	element type, empty list literal
Array	element type, index type
Powerset	member type

Figure 17: Predefined Generators

String corresponds to an indexed list of values of the element type. The predefined data type Charstring is in fact generated from String with the element type Char.

Array corresponds to a mapping from the index data type to the item data type. There are no requirements that the index data type of an array is discrete and limited.

Powerset corresponds to a data type, value of which is a mathematical set.

A number of operators is available for these predefined generators, e.g. indexing an array and insertion of a member into a powerset.

As an example of use of a generator, a data type representing Integer sets can be defined as:

```
newtype Integerset;
    Powerset(Integer);
endnewtype;
```

3.6.2 Inheritance

A new data type definition can be based on existing ones by using the specialization concept (see 4.5).

3.6.3 Ordering

Ordering of values of a data type can be indicated by including the word **ordering** among the operator definitions. **ordering** expands into the operator definitions for the usual ordering operators "<",">","<=" and "<=" and axioms for these operators.

In addition, if the data type includes definition of literals, the literals are implicitly ordered in ascending order.

3.6.4 Record data types

A *structure data type* is a composite data type whose values consist of a list of field values. This concept appears as record, tree, etc. in many languages. An example of a structure definition is the definition of the data type Car:

```
newtype Car;
    struct
        Brand Manufacturer,
        Year Natural,
        Paint Colour,
        Licence_Number Natural,
        Owner Citizen;
endnewtype Car;
```

Car is a structure containing the fields Brand of type Manufacturer, Year of type Natural, Paint of type Colour, Licence of type Natural and Owner of type Citizen.

Accessing a field of a **struct** is done by writing the name of the **struct** followed by '!' and the field name. Assuming the following variables:

```
dcl vehicle Car, joe Citizen;
```

then:

```
vehicle!Owner := joe;
```

assigns joe as the owner of vehicle.

A **struct** value is constructed by enclosing the field values with '(.' and '.)', as in:

```
vehicle := (. Ford, 1970, Red, 17, joe .);
```

It should be noted that the above constructs are shorthands for use of operators implicitly defined for the struct data type.

3.7 Syntype

A *Syntypes* is a special kind of data types which nominate a subset of the values of another data type. Variables of a syntype may only contain the nominated values.

SDL includes a single predefined data type Natural:

```
syntype Natural = Integer constants >= 0;
endsyntype Natural;
```

Variables of the syntype Natural must not contain negative Integer values. For example, if the variable v is of the syntype Natural then the assignment

```
v := -1;
```

yields an error during interpretation

4 Structuring

4.1 Types and Instances

In SDL, there is a clear distinction between *types* and *instances*:

- Instances are the entities in an interpreted SDL *system*, e.g. processes, variables, the signals in the input queue;
- Types define the properties of the instances in the interpreted SDL system, that is, every instance has an associated type defining its properties, but the types themselves are not part of the interpreted system;

Some kind of instances may be created and deleted explicitly by actions in the specification (e.g. call and return for procedures) while others are given an identifiable name in an *instance definition* and created or deleted implicitly when the enclosing instance is created or deleted respectively (e.g. variables).

As the types define the properties of instances, and thereby the semantics of the specification, they play an important role from the specifiers point of view. SDL therefore contains some powerful concepts for specifying types, known from object orientation:

- *Specialization* (see 4.5)
- *Virtual types* and *virtual transitions* (see 4.6).
- *Generic types* (see 4.7)

It is not all kind of types which can be defined as specialized or virtual types and only those kind of types which can be specialized, may be generic types.

Fig. 18 shows the different kind of types in SDL and the allowed operations on them. In the table, the term implicit is used when the instance of a type is implicitly created and deleted as mentioned above.

Note that the instances of data types and syntypes in SDL are variables (including the implicit variables in signal instances holding the conveyed values). [4]

System type, *Block type*, *Process type* and *Service type* are explained below.

4.2 The System Structure

As mentioned above, the interpretation of an SDL system is the interpretation of a number of processes interpreted in parallel. In terms of types and instances, the interpretation of an SDL system is thus the interpretation of a *system instance.*

A *system instance* is a container for a number of *block instances.*

A *block instance* is a container for either a number of *process instances* or a number of (sub)block instances

A *process instance* is either a state machine as explained in 2.1 or a container for a number of alternating *service instances* interpreted sequentially within the *process instance.*

A *service instance* is a state machine.

An SDL specification consists of a number of *diagrams* which all together specify the behaviour of a system instance. The *system diagram* defines the system instance itself, while the rest defines types and instances needed for defining the system instance.

The type associated with an instance is given when the instance is defined. A system type, block type, process type or a service type can be defined by means of a *system type diagram*, a *block type diagram*, a *process type diagram* or a *service type diagram* respectively.

Fig. 19 shows an example of a system type diagram

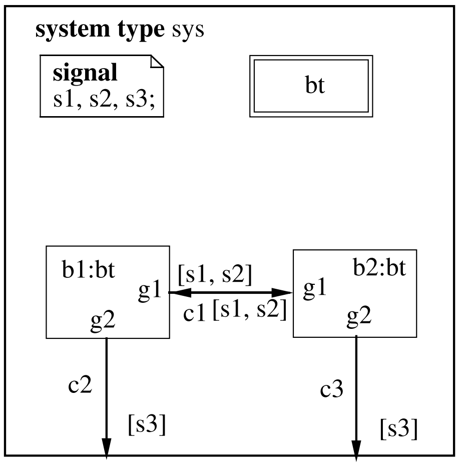
System types are defined in packages (see 4.8).

A system instance is defined in a *system diagram* as shown in fig. 20.

The system instance sysinst is of type sys and consists therefore of two block instances b1 and b2 which are of the block type bt. Block instances are connected by *channels* which are the communication media conveying signal instances between process instances in different blocks.

In the example in fig. 19, there are three channels. Channel c1 conveys signal instances of s1 and s2 between block b1 and block b2. Channel c2 conveys signal instances of s3 from b1 to the *environment* of the system

[4]Another view could be that the instances of data types (not syntypes) are the values themselves (since data types defines properties of values rather than of variables).



[s1]

Figure 19: System type diagram

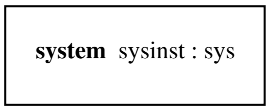


Figure 20: System Instantiation

and c3 conveys signal instances of s3 from b2 to the environment.

The identifiers g1 and g2 inside the block symbols are gate identifiers used for connecting the channels to process instance sets inside the block instances as explained below.

Note that SDL systems are open systems. The behaviour of the environment is not specified, but it is assumed to behave in an 'SDL-like manner', i.e. being able to send and receive signals and to contain process instance(s) identifiable by unique **sender** values (see 2.1).

In the example, the block type bt as well as the block instances are defined in the system type sys, but the properties of the block type is not specified in the diagram. This is because it would imply actual nesting of diagrams which, in general, is not convenient with respect to readability and handling by tools. The *block type symbol* in fig. 21 is instead just a *reference* indicating that the block type conceptually is defined here, but that the block type physically is defined in a separate diagram.

Fig. 21 shows an example of a block type diagram.

In this block type, two process instances are defined, p1 of process type pt1 and p2 of pt2. Process instances are connected by *signal routes* which convey signal instances to/from other process instances in the same block (as the case is for signal route sr1) or convey signals to/from the channels connected to the block (as the case is for the signal routes sr2).

Signal routes resemble channels. The only differences are:

Kind	Specialization	Instance creation	Instance deletion	Virtual
System type	Yes	At system startup	When there are no more process instances	No
Block type	Yes	Implicit	Implicit	Yes
Process type	Yes	Create & Implicit	Stop	Yes
Service type	Yes	Implicit	Implicit	Yes
Procedure	Yes	Call	Return	Yes
Data type	Yes	Implicit	Implicit	No
Signal	Yes	Output	Input	No
Timer	No	Set	Reset & Timer input	No
Syntype	No	Implicit	Implicit	No
Remote Procedure & Remote Variables	No	Modelled using signal interchange	-	No

Figure 18: The types in SDL and their usage

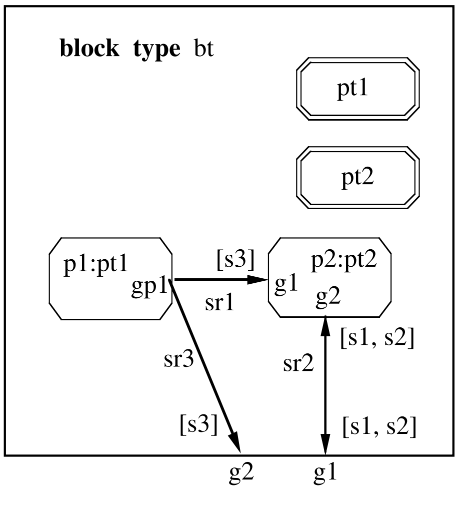


Figure 21: A Block type

- Signal routes are used for connecting process instances and service instances while channels are used for connecting block instances;
- Channels may imply a non-controlable delay when signals are transmitted. This feature is useful for specifying distributed systems where instant signal transmission cannot be assumed;
- Channels may have a *channel substructure* attached to elaborate the communication media in a distributed system (not further described);
- Signal routes are optional and if omitted, they are implicitly derived as being all possible communication paths.

As an alternative to defining process instances, a block type may be a container for a number of block instances, interconnected and connected to the boundary of the block type with channels. In this case, instantiation of the block type results in creation of the contained blocks. If a block type contains a large number of processes, this feature is useful for grouping conceptually related processes.

How instances of a block type are connected to channels cannot be specified for the block type as it depends on where and how the instances of the block are defined. Instead, the block type defines a number of *gates* being identifiable connection points of the block type. The gates are subsequently referred to when channels and block instances are defined.

The block type `bt` defines two gates: `g1` which may be connected to channel(s) conveying signal instances of `s1` and `s2` to and from the block and `g2` which may be connected to channel(s) conveying signal instances of `s3` from the block. In the system type diagram, it is shown in the block symbols that gate `g1` is used as connection point for channel `c1` and gate `g2` is connection point for channel `c2` and `c3`.

The gate concept is also used for specifying connection points of process types and service types. In the example, gate gp1 is thus a gate defined in process type pt1 while g1 and g2 are gates defined in process type pt2.

An example of a process type diagram for pt1 is given in fig. 22

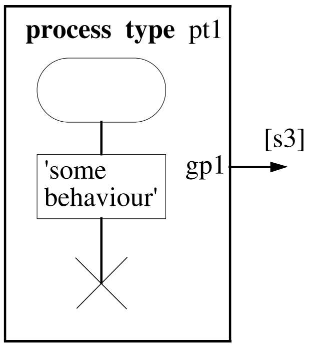


Figure 22: A process type

Instead of defining the behaviour of a process as a state machine, the behaviour may be defined in terms of a number of *service instances*. Each such service instance is a separate state machine, but only one service is executing its graph at a time. When the executing service reaches a state, the service capable of consuming the next signal in the input port of the process instance takes over interpretation, i.e. service instances share the input port of the process instance as well as the variables and the value of the expressions self, parent, offspring and sender.

A service instance is capable of consuming a signal if the signal is not saved in the given state and if the signal can be received by the service instance. Two different service instances must not be capable of receiving the same signal.

Services are useful when the behaviour of a process can be described as a number of independent activities (only sharing data), e.g. the two directions of a protocol, or if some activities are common for process instances of different process types.

In fig. 23, process type pt2 is defined in terms of two service instances s1 and s2 (of the service type st1 and st2 respectively).

The service type st1 is shown in fig. 24

Note the different symbols for block types, block instances, process types, process instances, service types and service instances and that type symbols are reference symbols (as explained above). Type symbols have the same shape as the corresponding instance symbols, but with an extra line surrounding the symbol.

4.3 Instance sets

The constructs for defining block instances and process instances in fact define sets of instances, but the sets con-

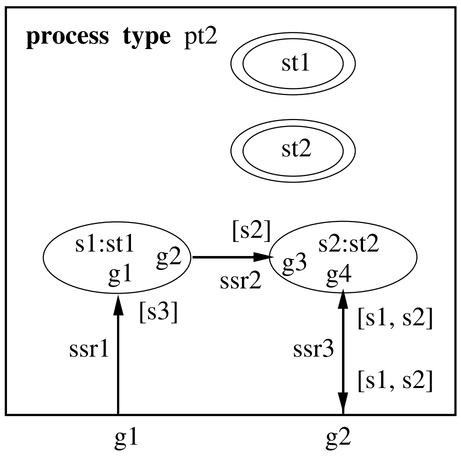


Figure 23: A process type containing services

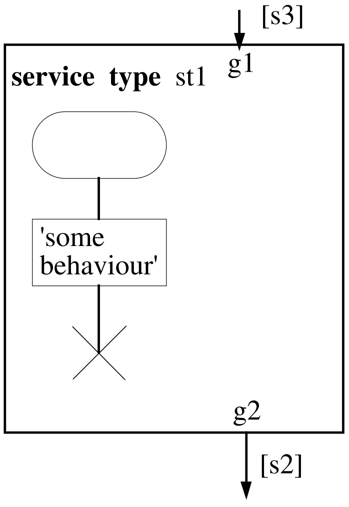


Figure 24: A service type

sist only of one instance unless the *number of instances* construct is used when defining the instance.

It could for example be defined that `b1` in the system type `sys` in fig. 19 should denote identical block instances by defining `b1` as shown in fig. 25.

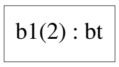


Figure 25: A block instance set containing two instances

The contents of the process instance sets may change during time (a process instance is deleted from the set when it stops and a process instance is inserted when a create action mentioning the process instance set is interpreted). The number of instances specified when defining process instance sets, is thus only the initial number to be created when the system instance is created.

In general, the terms block instance and process instance (rather than block instance set and process instance set) is used where it causes no confusion.

4.4 Instances with Anonymous type

Sometimes only one instance (or one instance set in case of blocks and processes) of a given type is needed in the specification. In such cases the definition of the instance (set) and the definition of its type may be merged into one definition, i.e. into a *system diagram*, a *block diagram* a *process diagram* or a *service diagram*. Such definitions define the instance and the type. The format of the instance diagrams resembles the diagram of their type diagram with the following exceptions:

- The keyword **type** is omitted;
- Channels and signal routes are connected directly to the channels or signal routes of the enclosing type rather than connected via gates;
- As such diagrams do not directly define types, specialization and redefinition of virtuals are not allowed (see 4.5);
- Block instance symbols, process instance symbols and service instance symbols are used as reference symbols for block diagrams, process diagrams and service diagrams respectively.

4.5 Specialization

A type may be a *specialization* (i.e. an extension) of another type. Conceptually, the types in a specification are grouped in a *type hierarchy* (i.e. a tree) where the nearest parent node of a type in the hierarchy denotes the *supertype* from which the type has been specialized. All parent nodes of a *subtype* are *supertypes* of the type. Likewise, all child nodes of a type are *subtypes* of the type. Often, the upper layers in the hierarchy are only introduced for the purpose of classifying types, i.e. some types may be 'abstract' implying that it does not make sense to use them for instance creation.

Consider for example that properties of a keyboard should be described. All keyboards are assumed to have the common properties:

- A number of keys;
- A buffer (possibly of length zero) containing the characters waiting to be delivered to the host;
- A communication interface to the host.

The most simple keyboards have no keypad and no function keys, i.e. each key correspond to one ASCII character only. All keyboards must at least have these 'standard keys'. Fig. 26 outlines a process type giving the properties of such a simple keyboard.

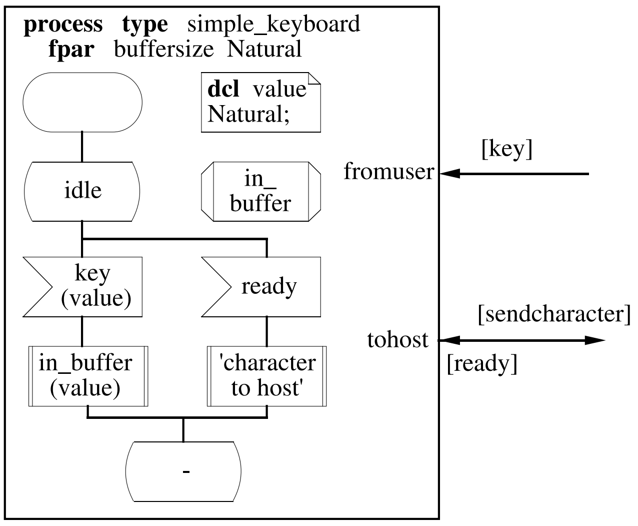


Figure 26: Definition of the simple keyboard

The gate `tohost` is the connection point to the host and the gate `fromuser` is the connection point to the user of the keyboard. The signals in the gates, i.e. `ready`, `sendcharacter` and `key`, are defined outside the process type.

A keyboard which supports function keys can then be defined as a specialization of the simple keyboard.

Specialization is indicated by the keyword **inherits** followed by the identifier of the supertype.

The more advanced keyboard is shown in fig. 27.

This process type has all the properties of the `simple_keyboard` process type, that is the gates, the formal parameters, the variables, the procedures, the start transition and the state defined in the supertype. These entities (e.g. the variable `value`) may also be referred to from the subtype.

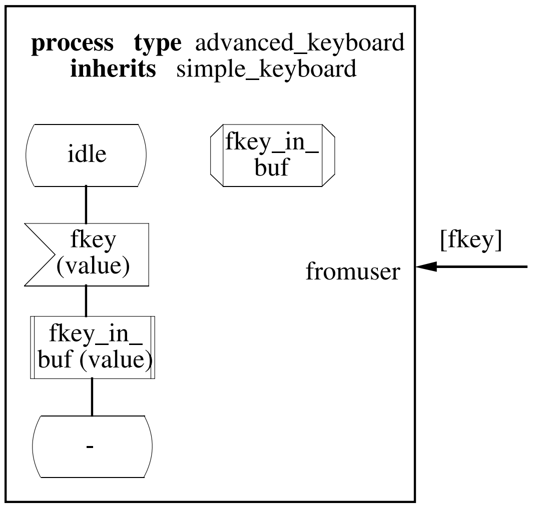


Figure 27: Specialization of the simple keyboard

Beside the properties of the `simple_keyboard` process type, the `advanced_keyboard` process type has:

- An extra input signal `fkey` in the gate `fromuser` and in the `idle` state. The signal models pressing a function key;
- A procedure `fkey_in_buf` which is used for converting a function key into a sequence of characters which are inserted in the buffer.

In general, a specialized type may add any property which can be defined for a type. Specifically, this means that:

- Any definition may be added as long as there is no name clash with the names defined in the supertype. When new channels and signal routes are introduced and their endpoints denote existing instances defined in the supertype, the instances are shown with a dashed symbol frame to indicate that they are not new instances;
- A specialized signal type may add (append) data types to be conveyed by the signal;
- Process types and procedures may add (append) formal parameters;
- Data types may add new literals and operators and behaviour for these;
- Formal context parameters may be appended. (see 4.7);
- Process types, service types and procedures may add a start transition (if there is not one already), add transitions to existing states and add new states;
- Block types, Process types and Service types may add signals to existing gates and add new gates.

4.6 Virtuals

Often a subtype needs to do some modifications to a supertype in order to adapt the supertype to specific needs. Therefore, when a type is specialized, some of its locally defined types may be redefined. The supertype determines which types the subtype is allowed to redefine. Those types are called *virtual types*. For types which define state machines, i.e. process types, service types and procedures, also the start transitions and the transitions attached to states, may be redefined by a subtype. The parts of a state machine which a supertype allows its subtypes to redefine, are called *virtual transitions*.

In order to preserve the properties of a type containing virtual types, there are constraints on how virtual types can be redefined. The 'default' and minimal constraint is that any redefinition of a virtual type must be a specialization of the virtual type itself. But by including the keyword **atleast** followed by the identifier of another type, the constraint is strengthened to denote that other type.

A virtual type or a virtual transition is indicated by the keyword **virtual** in the supertype. When a subtype redefines a virtual type or a virtual transition and the subtype allows further subtypes to redefine the entity, it is indicated by the keyword **redefined**, otherwise it is indicated with the keyword **finalized**.

Consider the process type `simple_keyboard` again. It does not leave much flexibility as it requires a new signal handled by a new transition if a subtype handling function keys is to be defined. It might be the case that the function keys are preferred to be handled by the same `key` signal (e.g. if they can be distinguished by the value carried by the signal).

The process type is made more flexible by defining the contained procedure as virtual and the transition handling the `key` signal as a virtual transition. This is shown in fig. 28.

Now there are two more ways the `simple_keyboard` can be specialized. Either by redefining the virtual procedure as shown in fig. 29 or by redefining the virtual transition as shown in fig. 30

In the first case, the procedure `in_buffer` is redefined taking into account that its parameter may denote a function key. In the second case, a new procedure is defined, and the transition is 'changed' to call the new procedure.

Although the definition of `simple_keyboard` with both a virtual procedure and a virtual transition is the most flexible solution, it involves a certain danger as there is no guarantee that the properties of the supertype are preserved. The specialization redefining the procedure is the most 'safe' alternative as a redefinition in general is always a specialization itself. If no explicit specialization is mentioned in the redefinition, the redefined type implicitly specializes from the original virtual type. Otherwise the specialization must obey the **atleast** constraint of the original virtual type as mentioned above.

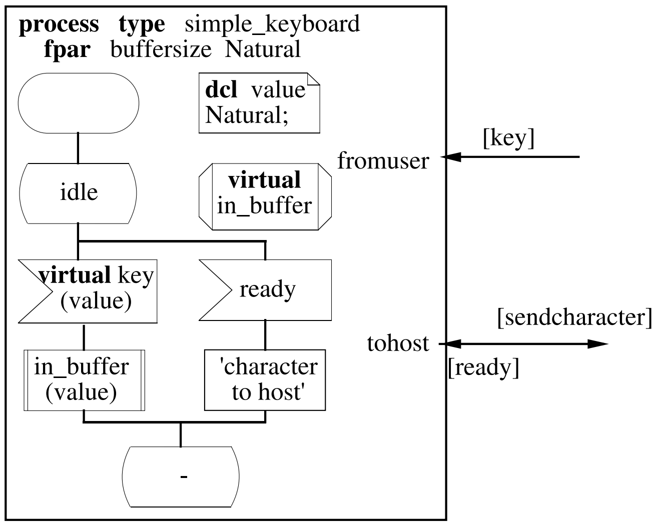


Figure 28: The simple keyboard with virtuals

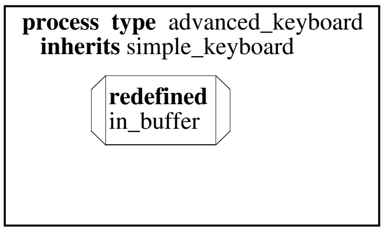


Figure 29: Specialization with redefinition of virtual procedure

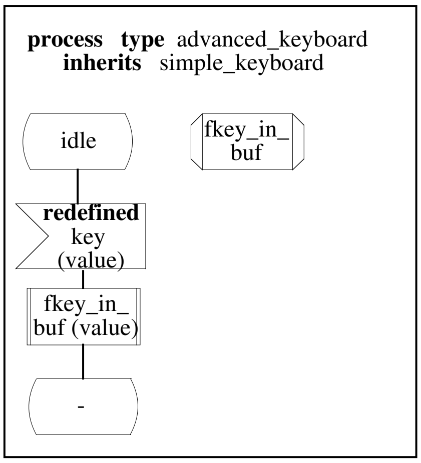


Figure 30: Specialization with redefinition of input

4.7 Generic Types

A generic type is incomplete in the sense that it may refer to entities which are not bound to a definition until the type is specialized. Such entities are known as *formal context parameters* attached to the type. Inside the type, only those properties of the formal context parameters are known (the *formal constraints*) which are of importance for using the context parameters.

Formal context parameters are enclosed by angular brackets (<>) and specified immediately after the name of the type.

When a type with formal context parameters is used, *actual context parameters* are supplied. These are the identifiers replacing the formal context parameters inside the type. The identifiers must obey the formal constraints attached to the corresponding formal context parameters. A generic type can thus be used to build several similar types.

There are two ways formal constraints of a formal context parameter can be specified:

1. As a *signature constraint* which is a specification of some properties an actual context parameter must posses (e.g. A data type must define certain operators).

2. As an *atleast constraint* (the keyword **atleast** followed by a type identifier) which is a requirement on any actual context parameter that it must be a subtype of or identical to the identifier.

Fig. 31 outlines the possible kinds of formal context parameters and how their constraints can be defined.

Some of the actual context parameters may be omitted when a generic type is used. Any formal context parameter which has no corresponding actual context parameter, becomes formal context parameters of the resulting type.

To show the use of context parameters, we may take a third look at the keyboard process (see 4.5):

- The simple keyboard process and the advanced keyboard process use different procedures for insertion of characters in the buffer and different signals for the two kinds of keys. The local procedure and the different signals could then be turned into a procedure context parameter and a signal context parameter respectively. To guarantee that other more advanced keyboard types also use a procedure which is a specialization of the simple one, the identifier of the simple procedure is given as constraints;

- It contains a formal parameter which is the buffer size. Assumably, the size of the buffer is the same for all instances of the same type. Since only the procedure is using the buffer size and the procedure is not locally defined any more, the buffer size is not relevant to the process.

Entity	Signature constraint	Atleast constraint
Synonym	Its data type	-
Variable	Its data type	-
Timer	Data types of its parameters	-
Signal	Data types of its parameters	A supertype
Procedure	Data types of its parameters	A supertype
Process instance	Data types of its parameters	A supertype of the process type
Data type	Required literals and operators	A supertype
Remote variable	Its data type	-
Remote procedure	Data types of its parameters	-

Figure 31: Formal context parameters and their constraints

A generic process type for definition of both the simple keyboard and the advanced keyboard is then defined as shown in fig. 32.

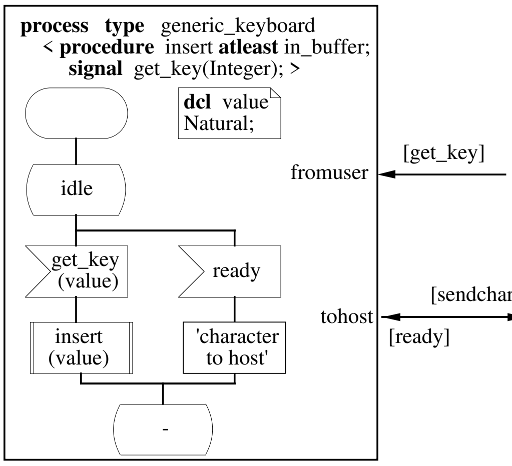


Figure 32: Process type with context parameters

The definition of `simple_keyboard` is shown in fig. 33.

```
process type simple_keyboard
  inherits generic_keyboard
  < in_buffer; key>
```

Figure 33: Use of process type with context parameters

The definition of `advanced_keyboard` is shown in fig. 34.

```
process type advanced_keyboard
  inherits generic_keyboard
  < insert_fkey_in_buffer; fkey>
```

Figure 34: Use of process type with context parameters

Note the clear differences between these process types and the ones given in 4.5:

- The processes are not related through the type hierarchy (i.e. the advanced keyboard process is not a specialization of the simple keyboard process). This has an impact on use of the simple process type in an *atleast constraint*;
- The procedure handling insertion of characters in the buffer is no longer locally defined in the process types.

4.8 Packages

An important characteristic of types is that they allow properties to be defined once, for subsequent use in different places to define objects (i.e. the instances) and other types. If a type is to be used in several specifications, it must be defined in a *package*. A package is defined by a *package diagram*. It contains a collection of types which can be used when defining system instances and when defining other packages.

An example of a package diagram is given in fig. 35:

When packages are used in a package diagram or a system diagram, a *use clause* is given in a text symbol above the diagram. A use clause is the keyword *use* followed by the list of packages to be used. For each package, the identifiers which are used can be listed, separated from the package identifier with a slash (/). If the list is omitted, all identifiers offered by the package can be used.

Fig. 36 shows an example where the package `generic-types` is used.

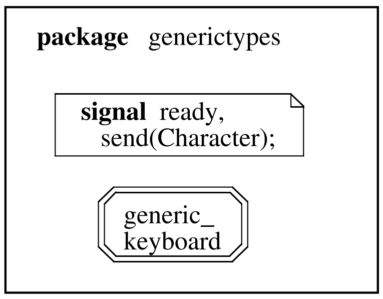


Figure 35: A package diagram

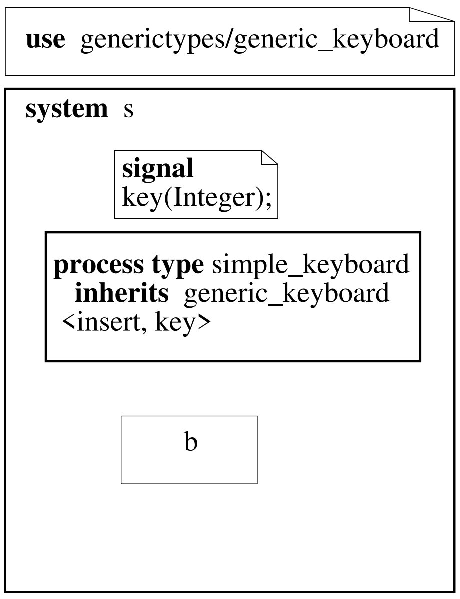

use generictypes/generic_keyboard
system s
signal key(Integer);
process type simple_keyboard inherits generic_keyboard <insert, key>
b

Figure 36: Use of a package

5 Status for SDL Work

5.1 Status in Standardization

The technical work on SDL-92 is completed now, and the result is a new CCITT recommendation ([3]) to formally become a standard in early 1993. It is accompanied by a new appendix: "SDL Methodology Guidelines" ([5]), which presents examples of SDL usage in standards and industry. It provides guidelines to organisations, who want to fit SDL to their own needs and possibly use SDL together with other notations, e.g. a notation for object-oriented analysis (OOA) or a programming language for code-generation from SDL-documents.

CCITT has also developed a new, draft recommendation on "Message Sequence Charts" (MSC, [6]). These diagrams are also well-known under various names like information-flows, interconnection-diagrams or sequence-diagrams. The rationale for producing an MSC-recommendation is to encourage building of MSC-tools and formal analysis of the information in MSCs. Part of [6] states the relations between MSC and SDL.

SDL and MSC are often used together. As an important example, the methodology for describing ISDN-services (recommendation Q.65, [7]), recommends to show typical cases of interactions between entities in a logical network model using MSCs and then to elaborate each of the functional entities as an SDL- process. The use of MSC and SDL in the context of Q.65 is elaborated in [5].

The usefulness of SDL in the context Q.65 gives good hope for utilising SDL to describe IN (Intelligent Network), since the CCITT-methodology for describing IN may likely be an evolution of the Q.65 methodology. SDL is also considered as one of four candidate languages for describing standards for ODP/DAF (Open Distributed Processing/Distributed Applications Framework), a joint ISO/CCITT standardization effort towards open distributed systems.

The SDL diagrams in CCITT recommendations have until now been drawn using the AutoCad-package. This makes automatic checks of the diagrams difficult. CCITT, ETSI, ANSI.T1 and TTC (Japan) have recently decided to use an SDL-tool for future production of SDL-diagrams. All four organisations will use the same tool and thereby achieve maximum, electronic interchangability of diagrams. The decision also implies, that diagrams can be analysed and simulated, before inclusion into standards, and that industry can retrieve the diagrams in a tool-independent format (SDL/PR) for inclusion into industry's own tools. No experience from this new practice is available yet, but it will surely be important for improving the quality of coming telecommunications standards.

5.2 Status in Industry

Industry is in general not interested in a specification technique as seen in isolation, but industry is interested in a development method, which includes use of a specification technique ([8],[9]); if the method can be shown to produce cheaper or better results. For this, CASE-tools are important.

Every second year an international SDL-Forum is held. At the 1991 Forum ([2]), 12 tools were demonstrated, out of these 7 are commercially available. Several of the tool-suppliers offer training courses on SDL. The more advanced SDL-tools today include graphical editor, translator between SDL/PR and SDL/GR, static analyser, simulator, code-generator and some kind of dynamic analysis. The only common form of dynamic analysis is deadlock-detection.

It is difficult to estimate the actual number of SDL-users. A conservative estimate based on a questionnaire in the SDL-Forum, 1989, is app. 8.000 daily users. Based on reports from tool suppliers, we expect the usage to be increasing. However, it must be said, that many SDL-users still use SDL as a drawing tool. Based on the same questionnaire, it can be said that most engineers involved with functional specification, design or documentation of telecommunications system are able to read SDL.

5.3 Status in Research Projects

SDL has been investigated in several research projects. Here, we shall mention work done in two European research projects: the ESPRIT project, Atmosphere ([10]) and the RACE-project, SPECS ([11]). It should also be mentioned, that the object-oriented extensions in SDL-92 originate from a Nordic research project, Mjølner ([12]) with some refinement done in the SPECS-project. Work in these three projects have included prototyping of tools for the additional features of SDL-92 as well as work on other languages in combination with SDL.

The work in the ATMOSPHERE project has led to a proposal of an interchange format of SDL-diagrams between different SDL-tools. SDL/PR is designed as an interchange format for all logical information, but no graphical layout information is maintained by SDL/PR. A proposal for interchange format including layout information is still being debated by CCITT, and quite likely an interchange format preserving layout of SDL-diagrams will be agreed at some time.

In the SPECS project, SDL is being extended to allow a more logic-oriented specification style. For this purpose "property languages" (PL) have been defined. SDL-PL allows to express logic predicates, including temporal logic, on an SDL-specification, e.g. "if signal *event* is received, the next output must always be signal *response*". The resulting mixture of model- and predicate-oriented specification style may possibly be utilised in a dynamic analyser or just considered as a suitable abstraction for behaviour descriptions. The work on PL has not yet been introduced to CCITT, but so far it has shown, that SDL can also be useful in advanced contexts of specification techniques.

Work on utilising SDL for service creation for IN has recently started in a RACE-II project, SCORE.

6 Acknowledgements

The development of a new recommendation is an achievement of the whole CCITT group who has contributed to the work. For this paper, we thank the colleagues from the CCITT group, who reviewed this paper: Ferenc Belina, Birger Møller-Petersen and Rick Reed.

References

[1] CCITT Blue book, Geneva 1989: Vol.X - fasc. X.1, Recommendation Z.100 - Functional Specification and Description Language (SDL)

[2] SDL '91, Evolving Methods, Proceedings from SDL Forum '91, *eds. Færgemand and Reed*, North-Holland 1991

[3] CCITT, COM X-R 17-E, Geneva March 1992: Recommendation Z.100 - CCITT Specification and Description Language (SDL) and Annex A to the Recommendation

[4] Fundamentals of Algebraic Specification 1, *Ehrig and Mahr*, Springer 1985

[5] CCITT, COM X-R 21-E, March 1992: Appendix I to Recommendation Z.100 - SDL Methodology Guidelines

[6] CCITT, COM X-R 22-E, March 1992: Draft Recommendation Z.120 - Message Sequence Charts

[7] CCITT Blue book, Geneva 1989: Vol.VI - fasc. VI.1, Recommendation Q.65 - Stage 2 of the method for characterization of services supported by ISDN

[8] An example of the use of SDL within GPT, *Wiggins*, GPT, in [2], pg. 499 - 508

[9] Experience in the use of SDL/GR in the Software Development Process, *Klick, Patti and Todd*, AT&T, in [2], pg. 449 - 457

[10] Atmosphere Briefings, *ed. Vader*, Philips Research Lab, Eindhoven, the Netherlands

[11] The Project SPECS - available from SPECS prime contractor: GSI-TECSI, 6 cours Michelet, Paris la Defense, France

[12] Mjølner, A Highly Efficient Programming Environment for Industrial Use, *Dahle, Loefgre, Magnusson, Madsen* in Proceedings: Software Tools 87, Online Publications, London 1987