New Features in SDL-92

Ove Færgemand[1] and Anders Olsen[2]

TFL, Telecommunications Research Laboratory

Lyngsø Allé 2, DK-2970 Hørsholm

Denmark

e-mail: ove@tfl.dk, anders@tfl.dk

Table of contents:

---

[1]CCITT-rapporteur for the maintenance of SDL and chairman of CCITT working party X/3 'Formal Description Techniques'
[2]CCITT-rapporteur for the formal definition of SDL

Introduction

This tutorial provides an overview to the version of SDL, which is expected to be recommended by the CCITT plenary, February 1993. The technical work on this version of SDL, which will be called SDL-92 below, is completed [1]. The term SDL-88 will be used for SDL as defined in the 1988 recommendation [2]. The paper is not a complete tutorial on SDL-92, because textbooks on SDL-88 already exist [3, 4], rather it informs about the improvements included in SDL-92. Major new features are in the area of object-orientation but also new concepts have been introduced for the description of behaviour and data in SDL-92. The tutorial first describes the changes to structuring, then the changes to data description, and then the changes to behaviour description. At the end of the tutorial, a number of minor changes are mentioned. The tutorial does not attempt to be a complete coverage of all changes; for this the reader is referred to upcoming textbooks on SDL-92.

SDL has evolved from a small informal drawing technique, first standardized by CCITT in 1976. The intention of CCITT has been to keep SDL stable during the recent study period (1988-1992), but still it is estimated that in terms of concepts, keywords etc. the language has grown by something like 10% in the period. Except for few simplifications, the changes are extensions, implying that a valid SDL-88 specification in almost every case is also a valid SDL-92 specification.

The compatibility with older versions is however one major reason for continued growth of the language. It is therefore expected that the mandate for maintenance in the next study period will allow deletion of old, outdated features (e.g. inheritance of sorts in SDL-88 is incompatible with the general approach to specialization in SDL-92).

The SDL-92 recommendation consists of the main recommendation text and the following annexes:

Annex A:	Index of non-terminals and keywords
Annex B:	Glossary
Annex C:	Initial algebra model (sec. 5.3 of [2])
Annex D:	SDL predefined data (sec. 5.6 of [2])
Annex F:	Formal Definition

and with appendices:

Appendix I:	SDL Methodology Guidelines
Appendix II:	SDL Bibliography

Note, that the SDL Pictorial Elements (Annex E to [2]) will **not** be issued as part of SDL-92, and that the User Guidelines (Annex D to [2]) is **not** maintained, because parts of it are covered by Appendix I.

Structure Representation

Types, Instances and Gates

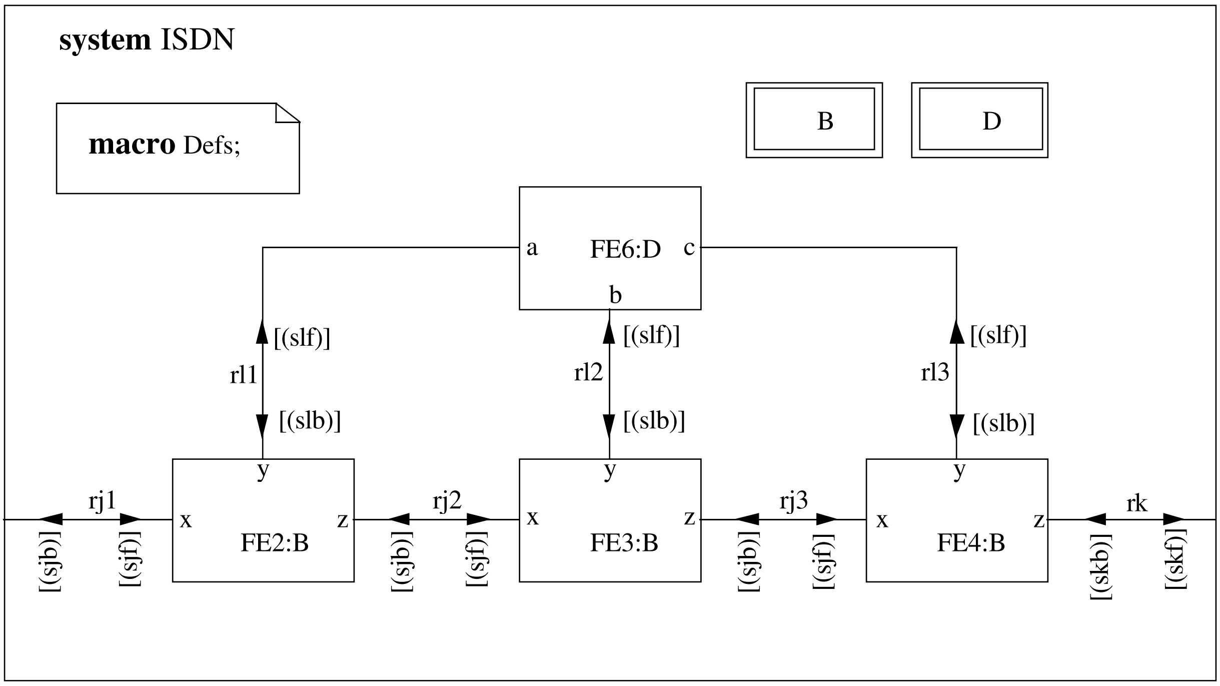
A definition in SDL is either a type definition or an instance definition. Procedure and signal are examples of types, and system, block, process and service are examples of instances. In SDL-92, system type, block type, process type and service type have been added in order to enable reuse of the same definition in different parts of a system. The terminology now used is that a block or process definition denotes a block instance set or process instance set. A block instance set based on a block type contains a fixed number of block instances. A block instance can also be defined directly by a block definition (like in SDL-88). A process instance set based on a process type may contain a varying number of process instances, determined by initial creations, create requests and process stops. A process instance can also be defined directly by a process definition (like in SDL-88). When a construct (e.g. system) can both be a type or an instance, the terminology is that *instance* (e.g. system instance) is implied when neither instance nor type is stated explicitly.

Types can be used to derive new (sub)types through specialization or to generate instances. That is, a block type can be *instantiated* as block instances or *specialized* as new block types, which are subtypes of the original one.

A signal definition defines a signal *type*. A signal *instance* is created when an output is interpreted, and it exists until it is consumed in an input of the receiving process. A procedure definition defines a procedure *type*, A procedure *instance* is created when a process is called and exists until the procedure call returns.

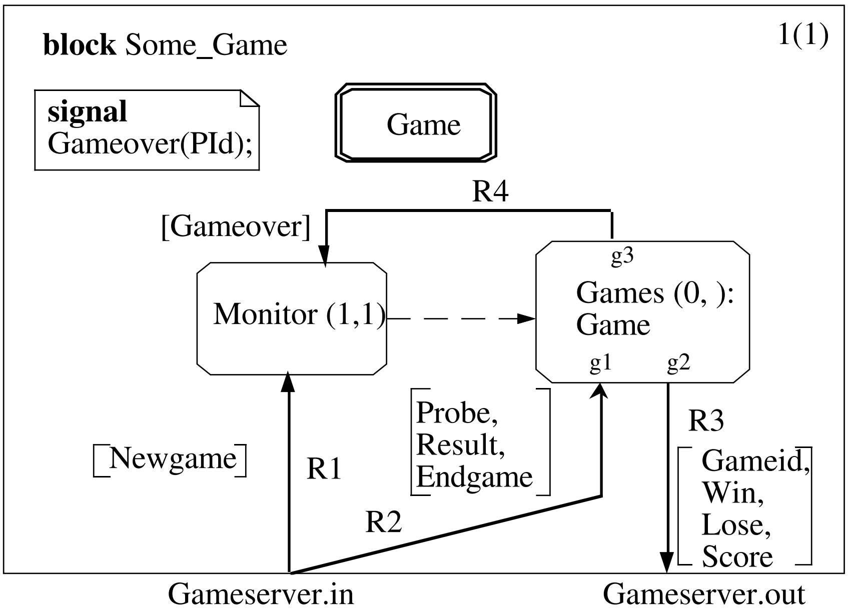
A block, process or service is connected to its surroundings by one or more communications paths, i.e. channels or signal routes. Since a type can be used in different contexts, a block, process or service type is not connected by communication paths. Rather these types have gates, being named connection points, which can be differently connected in different contexts. That is, when a block, process or service type is instantiated, channels or signal routes are connected to its gates. A gate allows to convey certain signals to or from the type, and it can be further constrained by stating that it may only be connected to communication paths leading to an instance of a certain type. This constraint is useful if it is known a priori that instances of a type should only be connected to certain other instances.

The following example shows a system diagram for a logical ISDN-network (the example is taken from [1], Appendix I: Methodology Guidelines) . This structure can only be expressed in SDL-88 by a lot of repeated definitions or by extensive use of macros. This is not a satisfactory solution. The diagram shows a reference ISDN network for describing supplementary services. It can be seen that the network consists of nodes (blocks: *FE2*, *FE3, FE4* and *FE6*) connected by communication paths (channels). But it is also part of Q.65 that the nodes constitute different capabilities depending on their types (*B*, *D*). As the figure shows, this can be conveniently expressed in SDL-92:



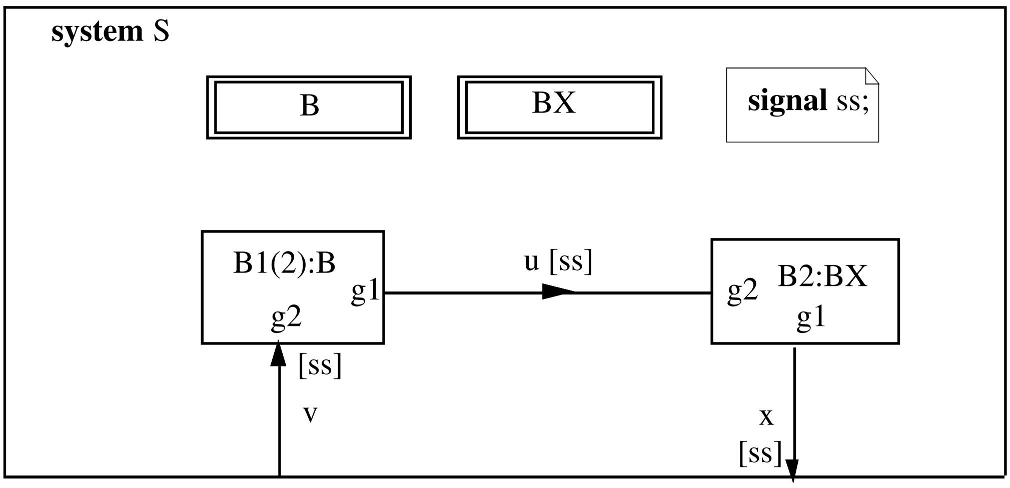
The need to express such reference architecture conveniently in SDL-92 has been one of the motivations for the object-oriented extensions.

The example below shows a block *Some_Game*, which contains two process symbols (octagon with single frame). One of these, *Monitor* is an ordinary process symbol, which contains the name of the process *Monitor*, and states that initial instances = 1 and max. no. of instances = 1. The other referenced process, *Games*, states that initial instances = 0 and that there is no limitation to the max. no. of instances. Then it contains ": Game", which means that it is an instantiation (:) of the type *Game*. The symbol for process type *Game* is also shown in the block. The double-framed symbol indicates a type reference (of a block, process or service) as compared to the instance reference symbols with a single frame already present in SDL-88. The names *g1*, *g2* and *g3* inside the symbol for *Games* are gate names of its type *Game*, describing the connections of the process instance set *Games*.

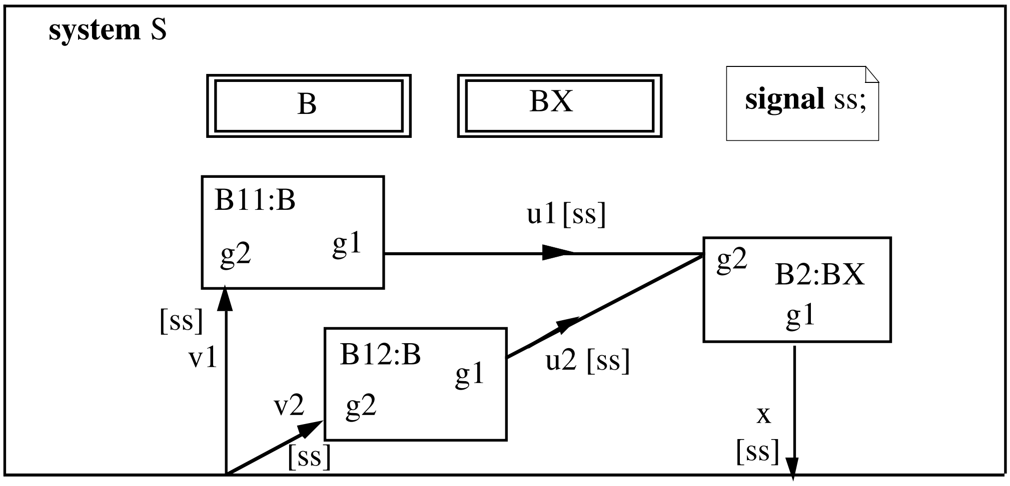


Multiple Blocks

An instantiation of a block type may specify multiple blocks at the same place. The example below shows a system with two block type definitions (block type references *B* and *BX*), which are used to instantiate the block (instance sets) *B1* and *B2*. The "*(2)*" implies that two block instances are created. The communication paths connected to *B1* will also be duplicated.



This diagram is equivalent to:



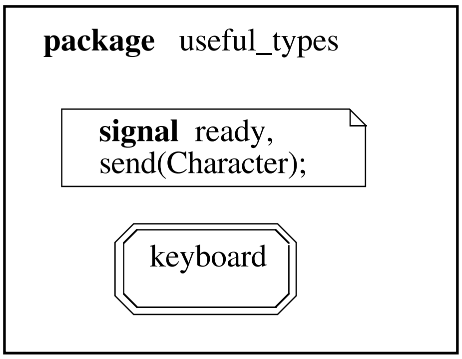
where *B* is instantiated as *B11* and *B12* and communication paths connected to the instantiation have been duplicated: *u1*, *u2* and *v1*,*v2*.

If a block instance set is specified with more than one instance, channels may be specified to lead from the instance set and back. Such channels correspond to channels between the individual blocks in the block set.

Packages

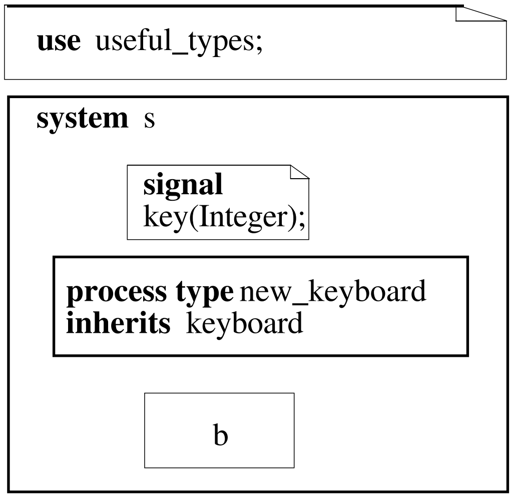
Types allow properties to be defined once, for subsequent use in different places for defining objects (i.e. instances) and other types. If a type is to be used in several specifications, it must be defined in a *package*. A package, which is primarily a collection of types, is defined by a *package diagram*. It can be used when defining system instances and when defining other packages through a **use** clause. In addition to type definitions, a package can contain signal lists, generators and synonyms. Although these definitions do not define types, they are likely candidates for reuse in different specifications.

An example of a package diagram is shown below:



This package contains signal definitions, *ready* and *send*, and a process type definition, *keyboard*, as indicated in the process type symbol.

Packages can be used in a package diagram or a system diagram, by giving a **use** clause in a text symbol above the diagram. A use clause consists of **use** followed by the list of packages to be used. Constructs for including only a subset of the definitions from a package are also available but not described here. When a package is used in a system diagram, the types in the package are considered as if they are defined on the system level. The fig. below shows the use of the package *useful_types* defined above.



The system *s* can be regarded as having three signals at the system level, *ready* and *send* from the package and *key* defined directly. It contains furthermore a process type, *new_keyboard* which is based on the process type *keyboard* from the package, and it contains a reference to a block diagram *b*. Note, that the definition of process type *new_keyboard* is directly enclosed in the system diagram, so no reference symbol is needed. The definition of *new_keyboard* is brief, because *new_keyboard* is based on an existing type (as explained below in specialization).

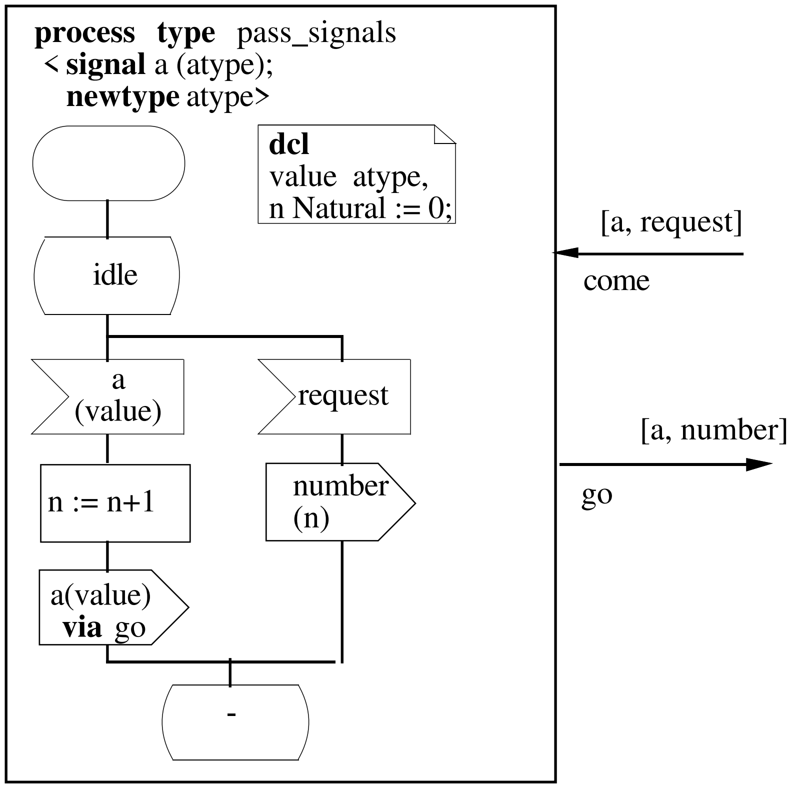
The predefined sorts (data types) and generators (i.e. Boolean, Character, String, Charstring, Integer, Natural, Real, Array, Powerset, PId, Duration and Time), which in SDL-88 are considered to be defined on the system level, are in SDL-92 collected in a predefined package, which is implicitly used by any system definition.

Generic Types

A generic type is incomplete in the sense that it may refer to names which are not bound to complete definitions. Such names are called *formal context parameters* of the type. Inside the type, only those properties of the formal context parameters are defined (the constraints) which are needed for using the context parameters.

Types which can be generic are system types, block types, process types, service types, procedures, signals and sorts. Formal context parameters can be processes, procedures, remote procedures, signals, variables, remote variables, timers, synonyms and sorts.

Formal context parameters are enclosed by angular brackets (< and >) and specified immediately after the name of the type. The fig. below shows a generic process type which has two formal context parameters: a signal context parameter, *a*, and a sort context parameter, *atype*. Note, that *atype* is given as constraint to *a*, because the sort conveyed by *a* is needed inside the generic type, e.g. for declaring a local variable of an appropriate sort.



When a type with formal context parameters is used, actual context parameters are supplied, replacing the formal context parameters inside the type. The actual context parameters must obey the constraints attached to the corresponding formal context parameters. By binding to different actual parameters, a generic type can be used to build several similar types.

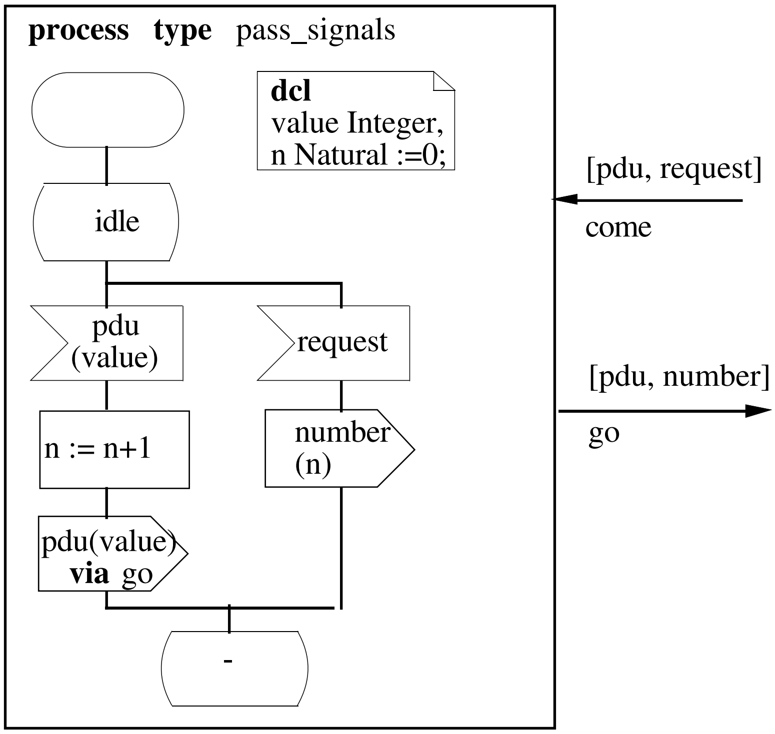
The type *pass_signals* which basically passes an arbitrary signal, conveying a value of some sort, while counting the number of passed signals, can then be used for constructing e.g. a process type which passes this signal:

signal pdu(Integer);

Here, only the process type diagram is given, assuming that the signal *pdu* is defined in some surrounding scope:

process type pass_pdu
inherits pass_signals
<pdu, Integer>

This corresponds to the following process type:



Constraints of a formal context parameter can be specified by a *signature constraint* which is a specification of some properties an actual context parameter must possess (e.g. a sort must define certain operators), or by an *atleast constraint* which is a requirement on any actual context parameter to be a subtype of or identical to the type stated in the constraint. In the example above, only a simple signature constraints have been used: since no operators of *atype* are used within the generic type, no operator needs to be identified in the signature constraint. Assuming that an operator *some* which accepts an Integer value and returns an *atype* value should be used inside the generic type, the formal context parameters in the definition of the process type *pass_signals* above would look like:

```
< signal a(atype);
  newtype atype
    operators some: Integer -> atype;
  endnewtype >
```

That is, again only the necessary information has been stated for the formal context parameter.

Specialization

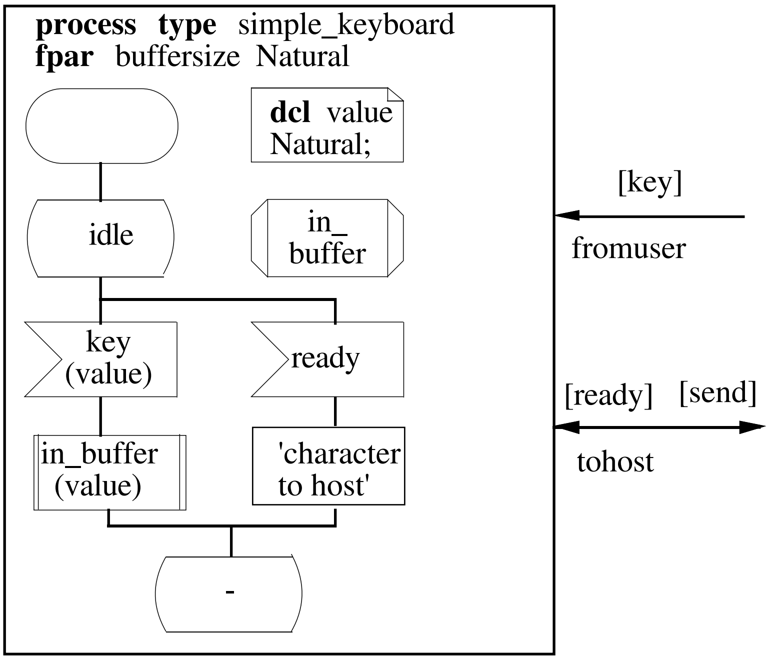
Specialization allows one type to be based on the properties of another type. Specialization can be strict; i.e. addition of properties, or it can allow redefinition of virtual properties (see next section). Properties which can be added can be definitions (of e.g. new sorts, signals or variables) and state-transitions for process types, service types and procedures. Properties which can be redefined can be definitions of types with internal behaviour (i.e. block type, process type, service type and procedure) and transitions for process types, service types and procedures.

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 (sub)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.

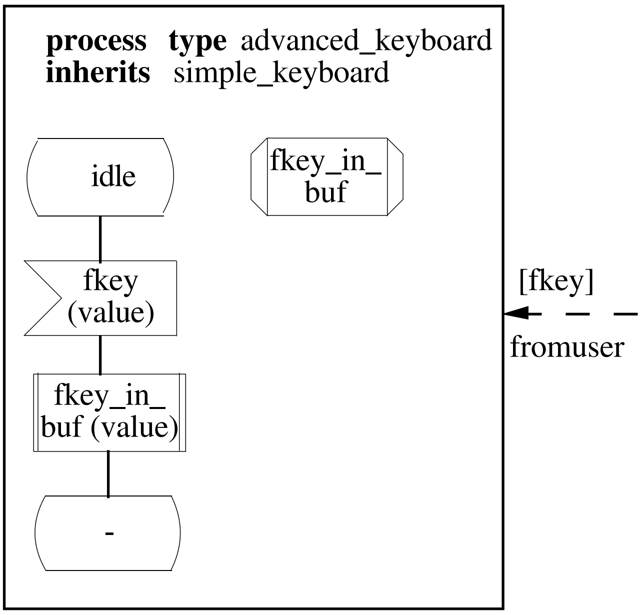
Consider for example that properties of a terminal keyboard is to be described. All keyboards are assumed to have the common properties:

- 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 corresponds to one ASCII character only. All keyboards must at least have these 'standard keys'. The fig. below outlines a process type giving the properties of such a simple keyboard.



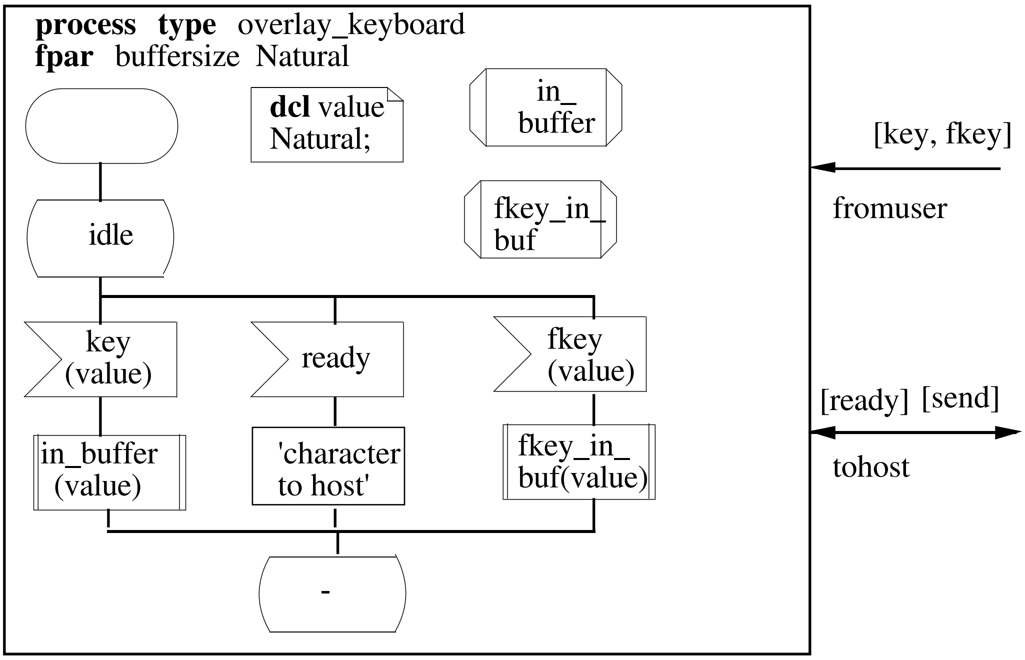
Note, that the procedure *in_buffer* is only given by a reference symbol and that the passing of characters to the host is only stipulated using informal text ('character to host'). A keyboard which supports function keys can now be defined as a specialization of the simple keyboard:



In a specialization, all local definitions from the supertype can be used, i.e. the variable *value* and the state *idle* can be used in the specialization of *simple_keyboard.*

The dashed line for the gate *fromuser* indicates a gate already existing in the supertype (*simple_keyboard*), and used in the subtype (*advanced_keyboard)* for adding signals (*fkey*) to the interface of the type.

The resulting process type is equivalent to the following composition of supertype and subtype:



Virtuals

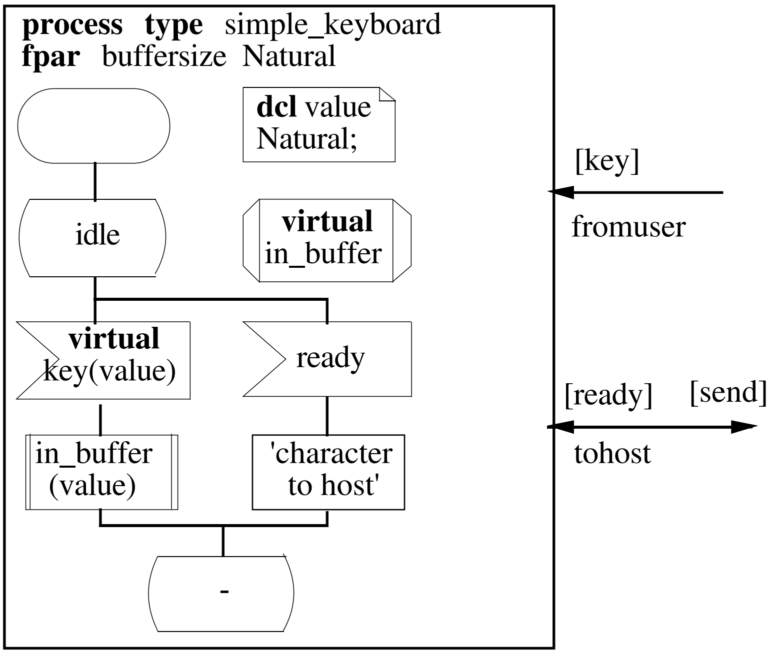
Sometimes a subtype needs to modify a supertype slightly in order to adapt it to specific needs. Therefore, when a type is specialized, some of its locally defined types may be redefined. For types which define state machines, i.e. process types, service types and procedures, also the start transition and the state transitions may be redefined. The supertype determines which types and transitions the subtype may redefine. Those parts of the supertype are called virtual (types and transitions).

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

In order to preserve the properties of a type containing virtual types, there are some 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 stating **atleast** followed by the identifier of another type, the constraint is strengthened to denote the other type.

Consider the process type *simple_keyboard* again. 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 should 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 below:



Identifying the Subtype

The identifier of a process (instance set) is used for two purposes in behaviour description: when a signal is output to a named set of process instances (see extensions of output below) and when the creation of a new process in a named set of process instances is requested. Process instance set is the term used for the construct defined by a process definition or by the instantiation of a process type definition (see sec. Types, Instances and Gates).

Inside a process type, the identifier of the process instance set is not known (a type cannot assume any knowledge about the identity of its instances). Instead **this** can be used to denote the implicit process identifier.

When **this** is used in procedure calls, it denotes a call of the actual subtype.

Global Procedures

SDL-88 only allowed procedures to be defined inside processes. In SDL-92 this restriction has been removed. A procedure defined outside a process assumes the same complete valid input signal set as that of the process from which it is called.

Non-delaying Channels

The delay in channels although intuitively sound when modelling a distributed system, adds to the complexity of the complete system state. This is a problem for tools, whose analysis depends on the number of states in the system state space.

SDL-92 therefore allows a choice whether a channel is delaying or not. For compatibility with SDL-88, the default is delay. A non-delaying channel is in the textual representation denoted by **nodelay**, and in the graphical representation by placing the arrows of the channel at the end(s) of the channel line.

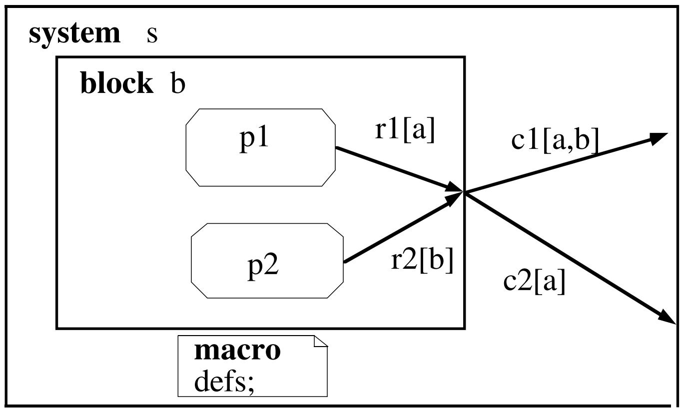
There is no semantic difference between a signal route and a non-delaying channel. Therefore, both constructs have the same graphical representation.

N to M Connections of Channels and Signal Routes

The restriction in SDL-88 that connection points on block and process boundaries can only be 1-to-N is found too restrictive after the introduction of types with gates and specialization. It is quite likely that in a specialization of a supertype, one wants to add new connections to gates inside the supertype, without considering if this gate (say of an enclosed process instance set) has already been connected (in this case to a signal route).

The restriction has been removed.

The example below shows a system, consisting of a block with two processes, *p1* and *p2*. The signal routes *r1* and *r2* are connected to the channels *c1* and *c2*. This is a 2-to-2 connection. The set of signals passed in a certain direction must be the same on both sides of the connection point; this set is here {a,b}. Also observe, that *c1* and *c2* are non-delaying (arrows at their ends) and that a textual macro has been used to keep the textual definitions (here: definitions of signals *a* and *b*) out of the diagram.



One purpose of connecting the two channels on the same point could in this example be to utilise both of the channels in **via** clauses in *p1* and *p2*.

Service as a Primitive Concept

The service construct has been defined as a primitive concept. It has thereby been possible to remove a number of restrictions. The concept service signal route has been merged with signal route.

The restriction of disjoint input sets for services still remains, since it is the basis for a deterministic alternation between services: since the first signal in the input port can only be handled by a single service within a process, the signal determines uniquely which service to activate.

Data Representation

Operator Definitions

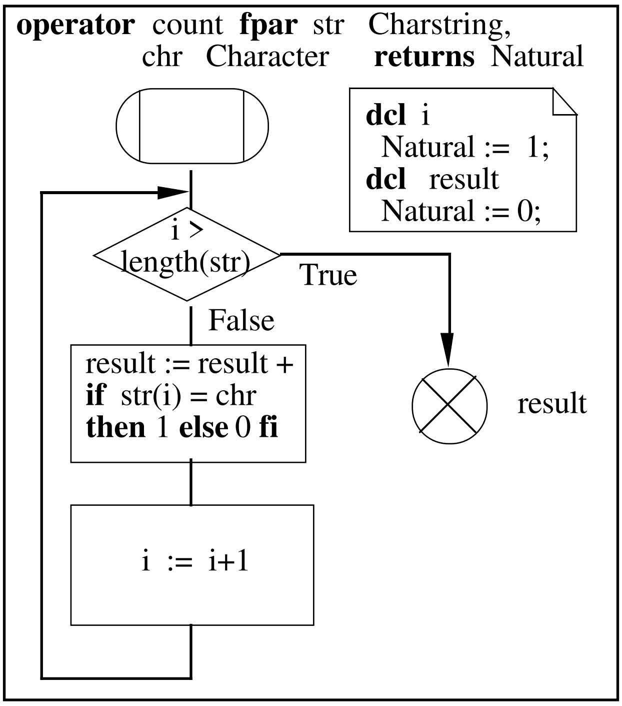
SDL-88 only allowed the properties of operators to be defined axiomatically. Two approaches have been added in SDL-92: algorithmic and alternative definitions. Alternative definitions are described in the next section.

In the algorithmic approach, the operator behaviour is specified in an operator diagram using a transition, like for a procedure. However, in an operator diagram, it is not allowed to manipulate the global state, such as process variables, inputs and outputs. The algorithmic approach is more 'implementation oriented' than the axiomatic approach and it may be 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 sort *Stringutility* with the algorithmically defined operator *count* is given below:

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

The definition contains an operator reference which in the textual representation is a reference to an operator diagram. The operator diagram for the count operator is:



Note, that the operator diagram resembles the procedure diagram (same start and return symbols) and that the result of the operator is stated in an expression next to the return symbol. In this case the expression is the value of the local variable *result*.

Alternative Data

In practice, SDL is often interfaced with another data description language such as the data specification notation ASN.1. Interfacing may also be done when translating SDL to a programming language, thus taking advantage of the features offered by a specific language. The concept of alternative (or external) data has been introduced to meet these requirements.

The alternative notation is enclosed between **alternative** and **endalternative**. It must be noted that, from the SDL point of view, the construct is considered as informal text, i.e. SDL does currently not provide any formal relation to other data formalisms. Future versions of SDL may include bindings between SDL and other techniques using this notation.

This example shows interfacing with C:

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

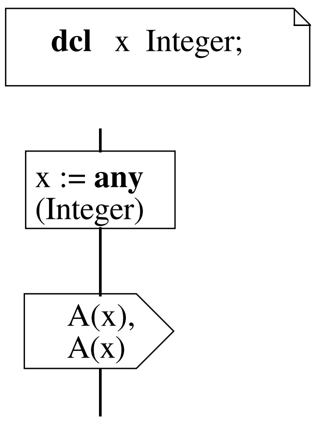
The next example shows interfacing with ASN.1:

```
newtype application_data
        literals empty;
        operators
                anyuseExtract!: application_data              -> Boolean;
                idExtract!      : application_data            -> Integer;
                anyuseModify!: Boolean, application_data      -> application_data;
                idModify!       : Integer, application_data   -> application_data;
                Make!           : Boolean, Integer            -> application_data;
        alternative ASN.1;
                SET {  anyuse BOOLEAN
                       id     INTEGER }
        endalternative;
endnewtype application_data;
```

Complete (SDL) static analysis of usage of these concepts is possible if the alternative data are supplemented by appropriate SDL-literals and operators.

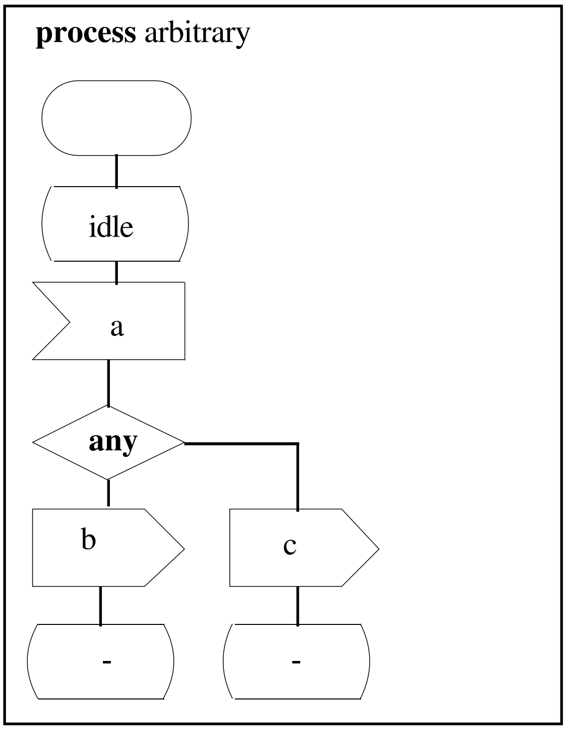
Anyvalue Operator

It is sometimes convenient to specify that a value of a sort can be arbitrary, 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 transmission. This can be modelled using the **any** operator. The **any** operator takes as argument the name of a sort, and it returns an arbitrary value of this sort.



Based on the **any** construct, an arbitrary decision is introduced. An arbitrary decision contains **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 (described below).

The next example shows a process which after reception of the signal *a*, answers with either a signal *b* or *c*. Such behaviour can easily be modelled in SDL-92:



Improved Inheritance

The inheritance for sorts in SDL-88 is not consistently defined. This has been rectified in SDL-92 while retaining the basic scheme. However, this scheme (semantics preserving) is not compatible with the general scheme for specialization in SDL-92 because the scheme from SDL-88 includes copying of non-local axioms, i.e. axioms which mention the operators and literals of the inherited sort. The general scheme of specialization in SDL-92 assumes copying of local properties only. For generic sorts, the general scheme in SDL-92 is applied.

This is not believed to be a problem to the users, however harmonization of the two approaches is one of the outstanding issues in the maintenance of SDL.

An implied operator for every type based on inheritance has been introduced. This operator ("<sort>!") takes as argument a value of the supertype and delivers the corresponding value of the (sub)type.

New Operators

SDL-88 only introduced **mod** and **rem** as keywords. In SDL-92, **mod** (modulus) and **rem** (remainder) have been introduced as operators for the predefined Integer sort.

Num (conversion of Character to Integer) and Chr (conversion of Integer to Character) have been introduced as operators for the predefined Character sort.

Noequality

For some sorts it is convenient that different spelling of literals imply different values, e.g. if one wants to define a simple enumerated sort it could be done like this:

newtype enumeration;
  literals a,b;
endnewtype;

Whereby it is implied that *a* /= *b*, and that a value of this sort can be either *a* or *b*. For other sorts however, like the predefined Integer sort, different spellings of literals may imply the same value. As an example 007 and 7 have different spellings but imply same value.

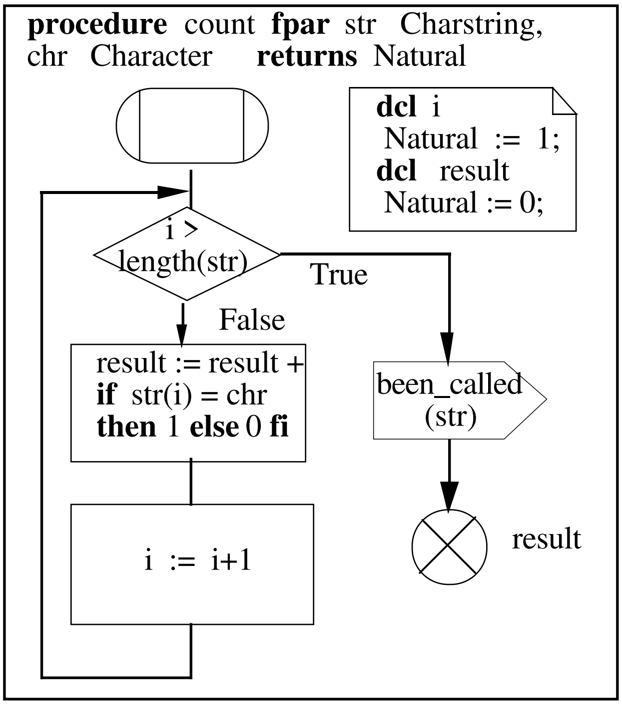
The distinction between these two cases is not done properly in SDL-88, and therefore **noequality** can now be specified in the operator list for sorts, where the biimplication between equal spelling and value is not wanted. In these cases, the user must explicitly define the equality properties, e.g. if **noequality** is specified for the enumerated sort, one would have to state a number of equality axioms explicitly.

It is expected that for user-defined sorts, **noequality** will seldom be needed.

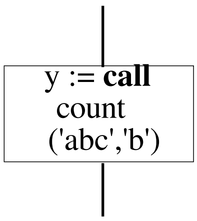
Behaviour Representation

Value Returning Procedure

In SDL-88, a procedure call can only appear as an action. This makes it rather cumbersome to utilise value returning procedures (although the result can somehow be achieved by utilising **in/out** parameters). In SDL-92, procedures can be defined as value returning and then be used in expressions. A value returning procedure is in fact a shorthand for adding an implicit **in/out** parameter. The operator definition shown above has been expanded, by adding an output of signal *been_called* on each activation. This is a side-effect so the operator has been changed to a value returning procedure. Note the similarity in syntax for definition of operators and value returning procedures. The distinction from a user point of view is that an operator is defined in the context of a sort definition and can have no side-effects.



This procedure could e.g. be called from a task:



A value returning procedure may not be called in the boolean expressions of continuous signals and enabling conditions. The reason is, that these boolean expressions may be repeatedly evaluated, and this leads to uncontrollable side-effects of the procedure calls.

Remote Procedure Call

The concept of exported procedure has been introduced as a generalization of the exported variable concept already present in SDL-88.

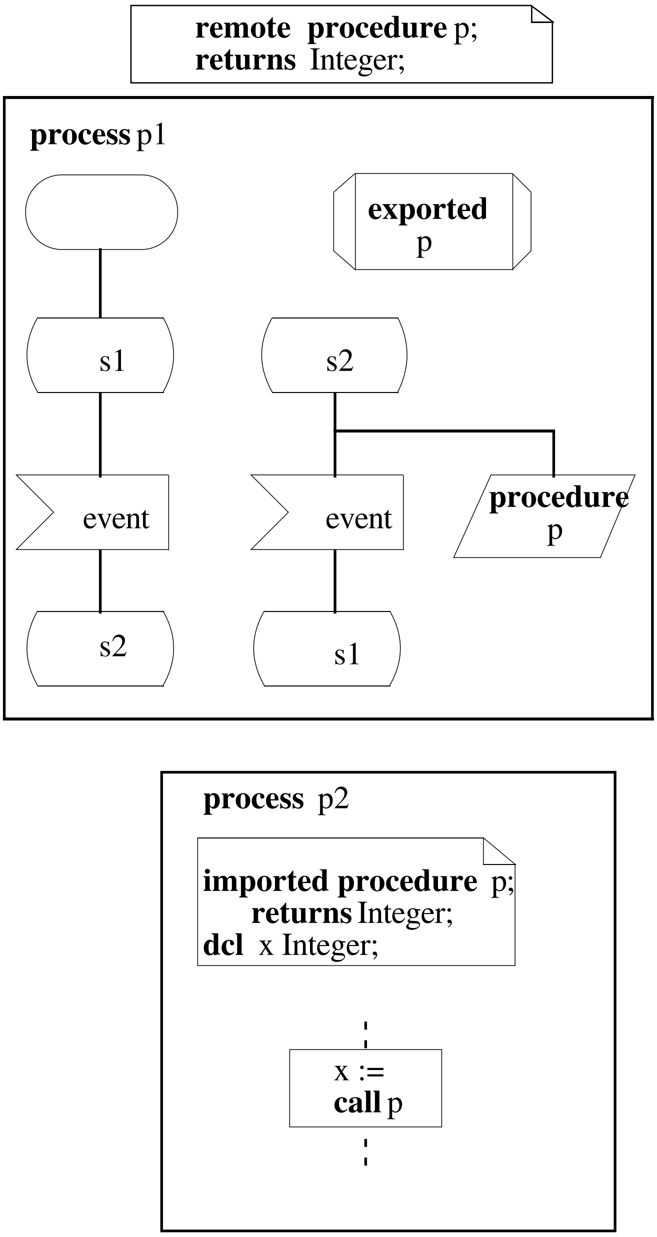
If a procedure has the **exported** attribute, it may be called from other process instances than the one where it is defined. Other processes wanting to call the procedure, must introduce it locally in an import specification. The exporter and the importer specifications are related by referencing the same remote procedure definition. The call of an exported procedure is transformed to an implicit 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 until the called procedure signals that the call is completed.

An acceptance of serving a remote call can be indicated as a state-trigger in the exporting process instance. When the process should remain in the same state taking no further action after the local call of the procedure, nothing needs to be stated in the state (the default). To specify a transition after the local call of the procedure **procedure** followed by the procedure name in an input symbol is used. To specify that the remote procedure is not served in a certain state (serving is postponed), **procedure** followed by the procedure name in a save 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). It also allows modelling of synchronous communication in SDL.

The fig. below shows an example of a process *p2* with a procedure call of a procedure *p*, 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 is used.

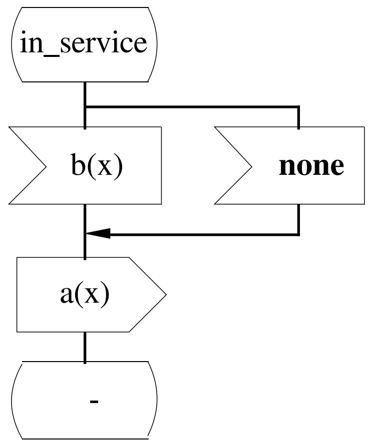
To harmonize the exported variable concept with the exported procedure concept, an optional remote variable specification has been introduced (like the remote procedure specification shown at the top of the previous example). The exported variable specification is optional, for compatibility with SDL-88, but its use is advised for new SDL specifications using exported/imported variables.



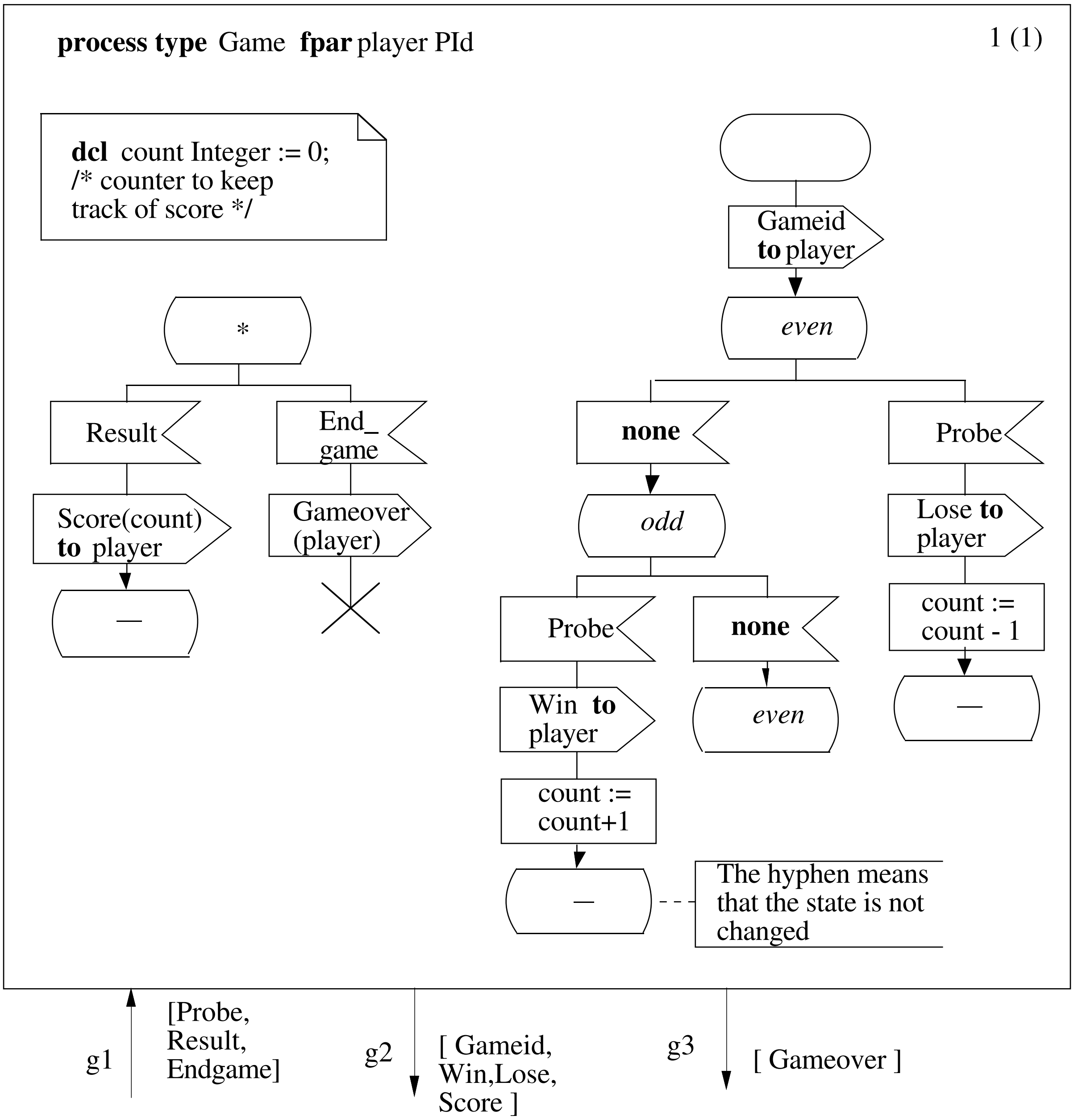
Spontaneous Transitions

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

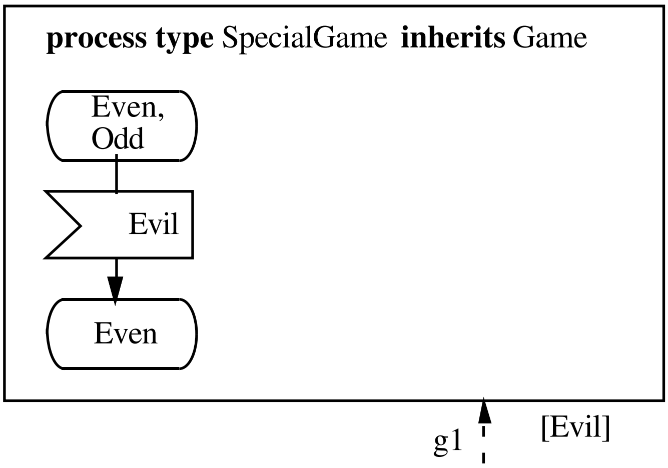
The fig. below shows an example of a state *in_service*, where a process can input a signal *b* and pass it on with same value *x* conveyed. This could be a model of a trivial protocol machine. However, the state has also a spontaneous transition leading to duplicated output of the signal. The process is a model of a communication medium which is not insertion-free.



Another example of use of spontaneous transitions is in the daemon game as it appears in [1]:



The formulation as a type, allows to specialize the game (making it worse for the player):



Addressing of Output

The capabilities of outputs in SDL-92 have been improved in two respects: The delivery scheme has been made easier to handle in a distributed environment and the addressing capabilities have been extended.

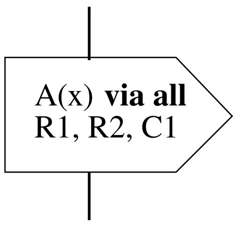
SDL-88 defines some cases of failure of signal delivery as errors, although obeying the error-conditions does not guarantee signal delivery completely. In SDL-92 these cases lead to the signal being lost. Also the requirement on exactly one possible receiver of outputs when no destination is specified has been relaxed, so that delivery is to an arbitrary process instance capable of reception (according to the system structure).

Addressing in SDL-92 has been extended so it is now possible to mention all visible communication paths in a **via** clause. This is important after the change to N:M connections in channel to signal route connections.

The explicit receiver of an output can be a PId-expression (denoting a certain process instance) or a process identifier (denoting an arbitrary instance of that process). The latter case is an extension reflecting actual usage of SDL in cases without multiple instances of the same process (e.g. for specification of protocols).

Multicast of Output

The **via all** construct allows to multicast one output to a number of receivers. This is particularly useful when a block type is instantiated as multiple blocks. The multicast means that an identical signal instance is sent **via** each communication path mentioned in the **via** clause. This output



results in three outputs of *A* with the same value of *x* on the three communication paths (gates, signal routes or channels) *R1*, *R2* and *C1*.

Priority Input and Output

The concept of priority signals and priority output has been removed in SDL-92. Priority output in SDL-88 is just a shorthand for **output** <signal> **to self**.

The priority input shorthand has been decoupled from the service construct and can now be used on a flexible basis like the other shorthands for a certain state (e.g. **input** *).

Default Value for Timers

Setting of timers have been simplified by allowing a default duration to be defined with a timer. When a timer with default duration is **set**, no expiration time needs to be given, and in that case the expiration time is derived from the current time (**now**) + the default duration. This reflects much informal use of timers, e.g. in standards. **now** is the built-in imperative operator which yields the actual time.

Fields of Structured Variables in Input

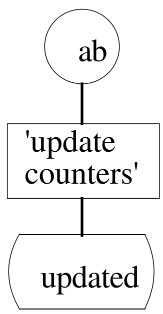
A model has been introduced which allows fields of a **struct** variable or of an indexed variable to be used for receiving values in inputs.

Free Action in Textual Representation

In SDL-88 the syntax for the behaviour description of a process in the textual representation is <start> <state>*, that is description of the start-transition followed by description of state. The graphical representation also allowed for a piece of graph, starting with an in-connector. The syntax in the textual representation has therefore been extended to <start> { <state> | <free action> }* where free action is

> **connection**
>   <transition>
> [**endconnection** [<connector name>] <end>]

where the first action of the transition has a label and where <connector name> must denote the label used in the first action of the transition. This extension makes the translation from graphical to textual representation of the example below, which shows a part of a behaviour graph, straightforward in SDL-92:



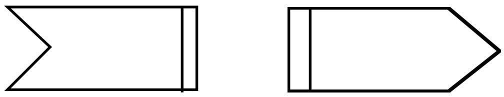
translates to:

connection
ab: **task** 'update counters';
nextstate updated;
endconnection ;

whereas the translation of such a piece was more complicated in SDL-88: before translation one had to find a state description where a join to *ab* appears, and then move the piece to that state.

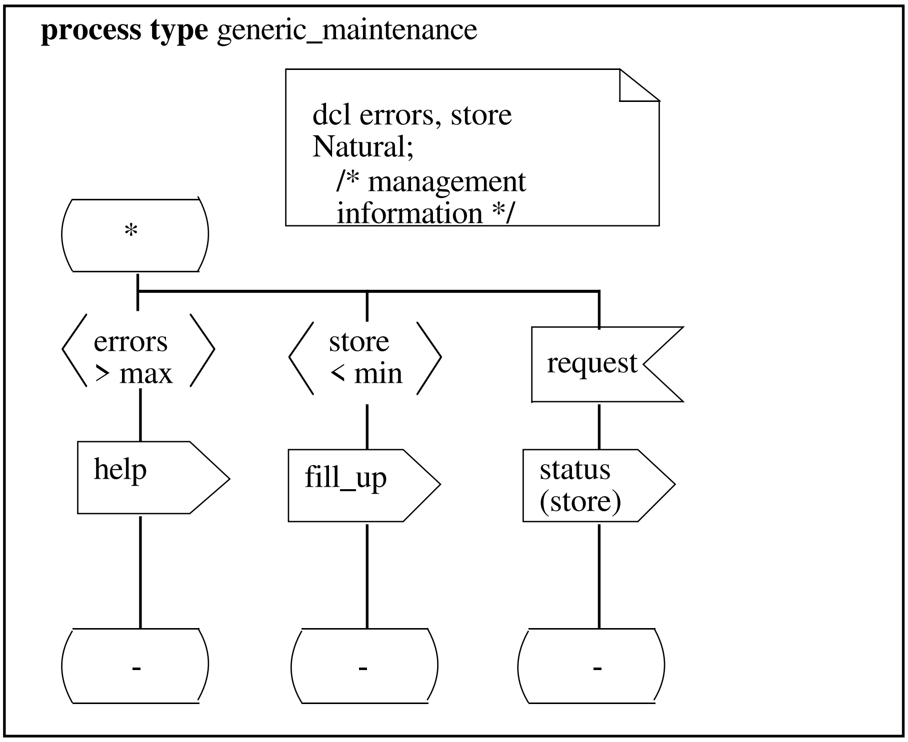
Internal Input and Output Symbols

For compatibility with existing diagrams especially in standards, two symbols have been re-introduced into SDL-92: internal input and output. They are not recommended for new SDL descriptions and they have the same semantics as the input symbol and output symbol respectively. The symbols are:



Multiple Continuous Signals With Same Priority for a State

In order to avoid non-determinism within processes, SDL-88 does neither allow several continuous signals in the same state without priorities, nor with same priorities. With the introduction of spontaneous transition and the arbitrary decision there has been no reason to maintain the restriction. It has therefore been removed. The example shows how this enables a more general use of continuous signals:



Improved View

The use of the revealed variable in view expression is one exception to the visibility rules in SDL-88. Now a special view-name has been introduced in the viewing process.

Compliant with the relaxed addressing rules for output, the rules for addressing in view have been relaxed too.

Due to compatibility constraints in the 1988-92 study period it has not been possible to remove the view/reveal construct in SDL-92. However it is one of the concepts, several organisations active in the SDL-maintenance work would like to remove from the language in the next years. There is no technical reason for using view/reveal for new SDL documents, rather imported variables (whose semantics is based on signal interchange) can be used.

Other Changes in SDL-92

Qualifiers can be enclosed in << and >> to improve readability.

It has been clarified that control characters (e.g. newline) may only appear in character strings if they are not significant. Control characters are not significant in informal text and in comments (<note>s). If significant, the concatenation operator (//) and the literals for control characters must be used.

Trailing commas in parameter list and in number of instances for processes may be omitted.

The rule for identifying referenced definitions has been made less restrictive.

The object-oriented extensions have contributed to larger diagram headings. It is therefore now possible to give the complete heading on the first page of a diagram only and give only a smaller heading (kernel heading) on consecutive pages.

Signal lists used in definitions of other signal lists may contain the same signals. The signal lists are simply considered as sets of signals.

References

[1] CCITT COM X-R 26, part II.3, revised recommendation Z.100, Geneva 1992

[2] CCITT Blue Book, Vol. X - Fasc. X.1, Recommendation Z.100, Geneva 1989

[3] SDL with Applications from Protocol Specification
F. Belina, D. Hogrefe, A. Sarma:, Prentice Hall 1991

[4] Telecommunications Systems Engineering
R. Saracco, R. Reed, J. Smith, North-Holland Elsevier 1989