Description of parallel processes in hardware using SDL (The CCITT specification and description language)

Hulzebos, R.M.

Award date:
1990

Eindhoven University of Technology
Department of Electrical Engineering
Digital Systems Group (EB)

Description of parallel processes in hardware using SDL (The CCITT Specification and Description Language)

by R.M. Hulzebos

report of graduation project by R.M. Hulzebos

Coaches: Prof. Ir. M.P.J. Stevens

Ir. A.C. Verschueren

Eindhoven, The Netherlands, August 1988.

ABSTRACT

SDL, the Specification and Description Language, is developed to describe the concurrent behaviour of processes in telecommunication systems. It has a textual and graphical representation with a one to one translation in both ways.

This report describes the way SDL can be used for description of parallel processes in hardware.

First, reasons are given why a formal description/specification language is needed. Here, we see that the main reason is the system complexity possible in current VLSI design.

Preceding the SDL-syntax description, a comparison is made between SDL and other description techniques Statecharts, ESTELLE, LOTOS and Petri-nets. At comparing the concurrent-, data description-, control-, formal definition- and human orientation- aspects of these techniques, certain needed constructs for description of parallel processes in hardware are defined.

Following the SDL-syntax description including an example description, the implementation of the SDL model and certain SDL constructs are discussed. At this point special attention is given to the asynchronous communication model and its implementation into synchronous communication.

In the last part of this report, a design methodology is given for the use of SDL in the specification phase of VLSI design.

SDL is a human friendly specification technique which can be used to set up a behaviour model of a digital system. This behaviour model must be rewritten into a hardware oriented behaviour description by refining certain parts of the behaviour model using an expert system.

CONTENTS

INTRODUCTION

The increasing complexity of digital systems due to the possibilities of current VLSI design asks for new design techniques. To reduce the time needed for designing a VLSI chip we need a structured design approach. This means that the design process must be divided in small steps and every step must be followed by verification of that step.

To describe the concurrent behaviour of communicating extended finite state machines we need special techniques. SDL, the Specification and Description Language developed by the CCITT, is especially developed to describe the concurrent behaviour of telecommunication processes.

In this report a methodology is given for the use of SDL in the specification phase of VLSI design.

First, some related description techniques are described followed by a comparison between these techniques.
Then, the syntax and structure of SDL is described. This description is clarified by the description of the Dining Philosophers problem in SDL.

In chapter 4, the implementation of certain SDL constructs and the SDL model is discussed. Here, special attention is given to implementation of communication, communication-channels, tasks, timers and process creation.

In the last chapter a specification system is described. This system is built of an expert system which translates the SDL behaviour description in a hardware oriented behaviour description in interaction with the designer.

1. Why do we need a description/specification language?

A design-process of a digital system is a communication-process between several designers. In order to understand each other they have to speak at least the same language. But when a system becomes more complex, the designer will soon need a pen and paper to write down what he/she means. At writing down the designer has to use symbols which are common to all the other designers. In other words: the designers must use some formal language to know what they are talking about.

Digital system design can be split into three levels:

1 Specification level
2 Design level
3 Implementation level

Formal languages used at the design-process can also be split into the same three levels. At the design level (2) the structure of the system is described using a design-language. At level 3 the system is described at port-level. At the specification level the system-behaviour is described using a specification-language.

The description of complex digital systems has to be done in a formal language to avoid lengthy and ambiguous descriptions in natural language.
A **formal** specification language describes a system in terms of functional behaviour no matter what the implementation technique is.

Advantages of a formal specification:

a] No relevant details are forgotten
b] Only one interpretation is possible
c] Computer aided specification (tools)
d] Possibility to simulate
e] Possibility for automated implementation
f] Possibility to prove equivalence of specifications

more precisely:

a] Using a formal-language, the designer has to give a complete specification. He or she is forced to describe all relevant details, otherwise a syntax error is indicated by a syntax-checker.

b] A formal-language does not allow any ambiguities due to its formal character. Any designer knowing the formal-language has the same interpretation of the formal-description.

c] Already pointed at a] is the use of a syntax-checker to check if the specification is complete.

d] The formal-description can be used to simulate the specification. Using simulation, the designer can compare the behaviour of the specification with the wanted behaviour (validation) or with its refinement (verification). The formal-description allows simulation of all possible combination of states and inputs, thus check on deadlock-free design is provided.

e] By defining an interface between the specification-language and a hardware description-language, it is possible to automate the design-process towards an implementation.

f] A formal language based on a solid mathematical theory can be used to prove equivalence of a specification and its refinement That is, prove whether the behaviour of the specification is the same as the behaviour of its refinement.

When designing complex digital systems built out of several parallel-working state machines, the formal specification-language has to cope with concurrent actions. Describing such systems has to be done in a formal-language to get a clear presentation of its behaviour. Using the well known state-diagrams for describing parallel-working state machines, is only useful for very small systems.

Conclusion:

To describe complex digital systems built of parallel-working extended finite state machines we need a formal language to cope with complexity, concurrency, ambiguity, validation, verification and simulation.

2. Statecharts, ESTELLE, LOTOS, Petri-nets and SDL

At first sight SDL looks like a good formal language fit to describe parallel processes in hardware. But to get an objective look at the properties of SDL, we have to compare it with some other formal description-languages.
When comparing, we especially have to keep in mind our application area, parallel processes in hardware. Before we compare some of the properties of Statecharts, ESTELLE, LOTOS, Petri-nets and SDL, we will give a brief description of these languages.

2.0.1. Statecharts

Statecharts is a description technique developed at the Weizmann Institute of Science in Israel [Harel]. This technique gives a presentation of the dynamic behavioral aspects of a system, rather than its physical or functional ones.
This technique is developed to cope with complex digital real-time systems.
Statecharts are based on the well-known state-diagrams extended with possibilities to describe *different* levels of abstraction, parallel working state-machines and a means of communication between those state-machines. In short:
statecharts=state-diagrams + depth + orthogonality + broadcasting

Depth is introduced by the clustering of states to one (super) state and refinement of one state in several sub states (fig 1).

Orthogonality means that several state-machines can simultaneously and independently of each other change state. Synchronization between state machines is possible by means of the implicit broadcast mechanism. Orthogonality in statecharts is described by dashed lines between parallel working state-machines (fig 2).

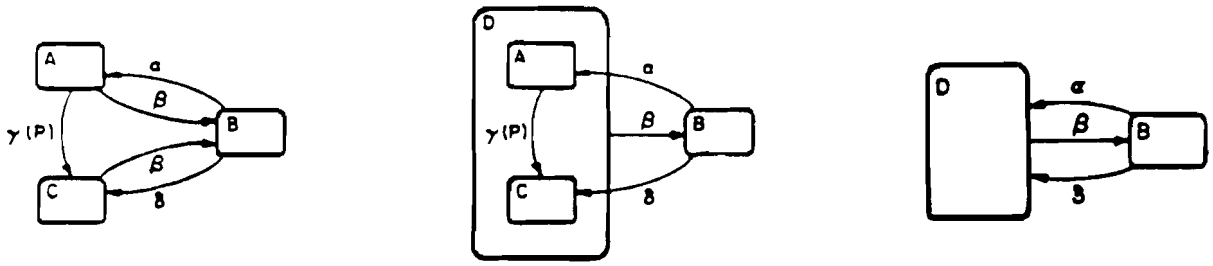


fig 1: example of clustering in statecharts.

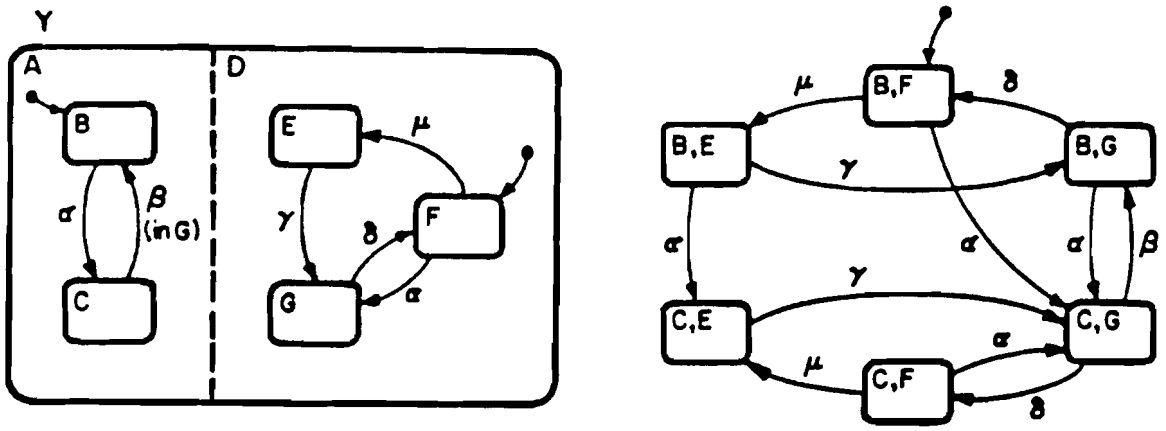


fig 2: example of orthogonality in statecharts.

In addition statecharts have constructs for conditional state transitions, delays and time-outs. Delay and time-out constructs use implicit defined timers. To specify a whole system the behavioral view described by statecharts is not enough. Therefore a structural view and a functional view are provided to make the system description complete. The structural view describes the physical structuring in modules and channels. The functional view describes data-flow and activities of the system-parts.

Above description of statecharts is far from complete but gives a general idea how concurrency is structured. The main advantage of statecharts is that it gives a clear **graphical** representation of concurrency.

2.0.2. ESTELLE

Estelle is developed by the International Standards Organization (ISO) [ISO ESTELLE]. The syntax of Estelle is close to that of PASCAL. Estelle introduces a kind of structuring and a means of describing the dynamic behaviour of a system. It only supports a textual representation of a system.

The Estelle system is called a **module**. This module can have a substructure of modules which communicate with each other using **channels** by receiving and sending **interactions**. A module can also be defined by an extended finite state-machine or the combination of state-machines and substructures.

Estelle has a tree structure where the root is the system and each leaf is an extended finite state-machine called a **Process**. The branches in the tree represent the substructuring of the system in modules. The Estelle-Process is modelled by states, input interactions, output interactions, transitions and other topics related to the description of an extended finite state machine.

Declaration of variables is done the same way as in PASCAL except that they may be defined at several (visibility) levels. Interactions between processes take place by means of interaction points. Each interaction point can be assigned to a common queue or can use an own queue containing the interactions received.

```
MODULE Recl_Type PROCESS (List2 .... List3 .... List4 ....;
                          List6: List6_Id(Receiver) INDIVIDUAL QUEUE;
                          PARAM Port_Nr.: Port_Type);
BODY  Recl_Body  FOR  Recl_Type;
  ....
  VAR  V_S,V_R: Seg_Type;
  ....
  TRANS
    FROM  Transfer  TO  SAME
      WHEN List6.Dat
        PROVIDED R_Nr ≠ V_R + 1
          BEGIN  OUTPUT List3.Rej(V_R);  END
        PROVIDED R_Nr = V_R + 1
          BEGIN  V_R:= V_R + 1;
                 OUTPUT List3.Rr(V_R + 1, S_Nr);
                 OUTPUT List2.U_Dat(I);
          END

    FROM  Transfer  TO  Discon
      WHEN List6.Dis
           BEGIN  END
  ....
```

fig 3: example of a part of a Estelle-module without substructure

2.0.3. LOTOS

LOTOS stands for Language Of Temporal Ordering Specification [ISO LOTOS]. It is strongly based on process algebra like CCS, a Calculus for Communicating Systems [Milner]. This means that the system is specified by defining the temporal relationship between the interactions that constitute the externally observable behaviour of a system. LOTOS is a functional description technique generally applicable to distributed, concurrent information processing systems. LOTOS has been especially developed for OSI standards.

Design principles of LOTOS:
Processes are described by ordering the units of their observable behaviour called **events**. Events are atomic instances of synchronized interaction between two or more processes. A LOTOS expression that defines an ordering of events is called a **behaviour expression**. Behaviour expressions may be combined in a new behaviour expression by the use of **operators** defined in LOTOS. For example, B1[]B2, means the combination of the processes B1 and B2 to one process containing a choice between the two processes.
The definition of abstract data types is done in the same way as in SDL. This means that a type is defined by its operators and the domain/range the operators work on.
Until now LOTOS has only a textual form, but the CCITT X/3 group has the intention to develop a graphical syntax for LOTOS and a proper tool for it. They will use the experience with the SDL graphical representation as basis and possibly bridge the gap between SDL and LOTOS.

2.0.4. Petri nets

There are all kind of description languages based on Petri nets. In these languages the basic principle of Petri nets has some extensions to handle specific problems in describing systems.
Some of these extensions are:

- introduction of timing constructs
- queues for asynchronous message handling
- use of predicates
- combination with state-diagrams.

However it seems that Petri nets and related forms are only useful for description of small concurrent systems. When the system to be described is big or is built out of more than two parallel working state-machines, it is very hard to give a clear representation using Petri nets.

2.0.5. SDL

SDL stands for Specification and Description Language. SDL is based on the model of Communicating Extended Finite State Machines (CEFSM). SDL has evolved from an informal language to a formalized language defined in Rec.Z100 of the CCITT [CCITT]. SDL supports a graphical and a textual presentation with a one to one translation in both ways.

An SDL **SYSTEM** has a set of **BLOCKS**. Blocks are connected to each other and to the **ENVIRONMENT** by **CHANNELS**. Within each block there are one or more **PROCESSES**. These processes communicate with one another by **SIGNALS** and are assumed to execute concurrently. The Processes are defined by extended finite state machines.

In SDL there are three levels of abstraction:
system level
block level
process level

Detailing in the block level is possible by BLOCK SUBSTRUCTURES and CHANNEL SUBSTRUCTURES. At process level detailing is possible using SIGNAL REFINEMENT.

Every SDL Process has one input-queue for all signals defined in a signal definition. Signals received by a process not defined in the signal definition are discarded. Consumption of input messages can only take place starting from a state. When the first message to be consumed is not defined in a particular state as an INPUT or a SAVE, the message is lost.

SDL has the same abstract data typing as mentioned in the LOTOS description. A more exact description of SDL constructs will be stated in the next chapter.

2.1. Comparison of SDL, Statecharts, ESTELLE, LOTOS and Petri nets.

Every description technique has its major advantages and disadvantages. When comparing different techniques we can define some idealized language which has all the advantages. The problem is that formalization of a description competes with a good human interface. A good specification/description language will be a compromise between certain demands.

In the following an evaluation is given of some features we need and how those features are implemented by some of the specification/description languages. This evaluation is partially based on [SDL '87] where SDL, LOTOS and ESTELLE are compared.

2.1.1. Concurrent aspects

Communication between processes can be described asynchronously or synchronously. From an implementation point of view the easiest way to let two state-machines communicate is synchronous. The description of communication between processes is easier when done asynchronous. Asynchronous communication between processes does not bother about (unspecified) delays, a state-transition occurs when a message arrives. When we want to synchronize certain actions to a clock-signal, it is easier to describe communication synchronously.

Another aspect of concurrency is the way different processes are synchronized to each other. Again, from an implementation point of view it is easier to synchronize two (or more) state-machines to a common system-clock. In a system description however it is often not important whether two processes (running in parallel) are synchronized or not.

The easiest way to cope with concurrent aspects is to base the specification language on asynchronous communication/parallelism with special constructs for synchronous communication/parallelism.
None of the languages stated cover all concurrent aspects. SDL and ESTELLE only implement the asynchronous model. Petri nets and Statecharts are based on the synchronous model. LOTOS is based on asynchronous parallelism with synchronous communication.

2.1.2. Data description aspects

To describe a digital system at a high level the language must contain a well structured data typing. The visibility of data has to be easy to define either explicit or implicit. This means that the visibility of data is defined by structuring of the system or by means of exporting/importing data between processes.

A good mathematical based data typing is necessary for simulation purposes and prove of deadlock-free design. On the other hand a strong mathematical base must not lead to difficult to make descriptions. For instance, a shift operation on a register must be a simple data operation rather than operations on all bits of a register.

The standard form of Petri nets has no data typing. Statecharts (and related views) are not formalized yet towards data typing. ESTELLE uses the same (weak) data typing as in Pascal. LOTOS and SDL are based on almost the same (strong) abstract data typing. The difference between LOTOS and SDL is that LOTOS is developed on this abstract datatyping, and SDL is formalized by introducing abstract data-typing later on.

The visibility of data in ESTELLE is based on free access of data inside the 'parent-relationship'. This means that child-processes of the same parent can access each others data. Inside a SDL-BLOCK data owned by one process can be made visible by a REVEALED

mechanism to other processes using a VIEW mechanism. A process can access data from a process outside a SDL-BLOCK using an IMPORT action containing the identifier of the owning process (data has to be EXPORTed first before it can be imported). LOTOS does not know visibility constructs. All interactions in LOTOS are based on explicit actions. This means that the only way to make data visible elsewhere, is to use explicit parameter passing.

2.1.3. Control aspects

To describe state-transitions at process level in an easy way, the language needs conditional and case statements. These statements should base their decisions upon input conditions as well as the contents of variables. Also conditional loops based on variable conditions are very useful to describe a process on at high level.

Especially in hardware design the language needs a natural way to describe system reset mechanisms and reset-state definitions.
This means that the language needs constructs for general control of all processes in the system. Priority constructs are very useful in this context, for instance to describe interrupt handling.

ESTELLE, LOTOS, SDL and Statecharts all implement a kind of conditional statements/loops. However general control of all processes is not possible using a simple construct. The only way to implement general control, is to describe in every state of every process how to react on general control messages.

2.1.4. Formal definition aspects

It is only possible to prove the correctness of a specification (description) if the specification (description) language has a formalized semantics based on a solid mathematical theory. During the specification stage it must be possible to prove whether the specification is equivalent with its refinement or not. To prove equivalence between two specifications means proving whether the observable behaviour is the same for the two specifications.

To automate implementation of the description it is also very useful to have a strong formal base. That is: it is easier to translate a strong formal based language into a high level hardware implementation language like [HHDL].

ESTELLE is based on Pascal but is not formalized yet. Until now the formalization of ESTELLE is not towards a strong mathematical base, so prove of equivalence of refinement is not possible. However ESTELLE stands very near to an implementation language because in ESTELLE many details have to be specified.
SDL is not formalized completely towards a strong mathematical base. However there are possibilities to translate SDL descriptions in CCS [Koomen].
LOTOS is based on CCS, so it has a good formal base. For automation of implementation SDL and LOTOS will need a library of certain objects.
Petri nets are common to use in searching for deadlocks in communication protocols between two devices. However Petri nets are not very useful (at least the standard form is not) in describing more than two state machines.

2.1.5. Human orientation aspects

In chapter one we already pointed out that we need a specification (description) language to handle things like system complexity, concurrency, ambiguity, validation and simulation. In other words we need a way to describe these things in a human friendly way.
The language therefore not only has to be easy to use but also easy to read. For readability purposes the language therefore needs a graphical representation. However this graphical representation must have a one to one representation in text for simulation and translation purposes.
An aspect which is closely related to the ease of using of the language is that the implementation has to be as independent of the specification as possible.

LOTOS and ESTELLE only have a textual form. Therefore it is not easy to get an overview of the system in a short time. Because of the closeness of ESTELLE towards implementation, ESTELLE also is not easy to use (to many details have to be specified).
Statecharts, SDL-GR and Petri nets are easy to read because their graphical representation. SDL-GR has an (almost) one to one representation in textual form (SDL-PR).

2.2. Conclusions

Before the strong and weak aspects of the different languages are given let us summarize the fields the languages are developed for.
Statecharts is the only language which is especially developed for describing complex systems built of parallel working state machines. ESTELLE, SDL and LOTOS all are based on a (state) machine model. LOTOS is developed especially for formal description of protocols in the OSI environment. ESTELLE is developed for specification of data communication protocols. SDL is developed for specification and description of telecommunication-standards,

processes and systems especially those related to switching and signalling.

Statecharts is strong because of its graphical representation. But the technique is weak because of its closeness to implementation in state machines.

ESTELLE is strong because it is based on the Pascal language which is used by many design languages. However ESTELLE (Pascal) does not have abstract data typing and is weak because of its closeness to implementation.

SDL has grown to a formal description language from a human oriented description technique. The main advantage of SDL is the formal use of graphical description with one to one translation to a textual form.

LOTOS is a strong language because of its mathematical base (CCS). But LOTOS is to close to CCS to be a user friendly language. When a graphical form of LOTOS is provided, the language will be far more popular.

Petri nets are easy to read but only useful for very small systems. Extended forms of Petri nets are more user friendly towards bigger systems.

3. SDL syntax and structuring (Dining Philosophers problem)

In this chapter the syntax of SDL will be described with an example description of the Dining Philosophers problem [Dijkstra]. The Dining Philosophers problem is used to describe semaphore operation and avoidance of starvation. The exact problem definition is as follows:

There are n philosophers whose lives consist of alternately thinking and eating. The philosophers eat at a large circular table with a preassigned plate for each. Between two plates is a fork, which may be used by either adjacent philosopher. In order to eat, a philosopher must have two forks (one on his left and the other one on his right).

avoid deadlock and starvation:

* deadlock will occur when all philosophers have one fork waiting to get the other one.

* starvation will occur when the neighbours of a philosopher get their common fork first.

```
model:
        processes:philosophers
        resources:forks

    Philosopher(i):     begin
                            Think;
                            claim_forks;
                            Eat;
                            release_forks;
                        end
    LEFT.FORK(i)=RIGHT.FORK(i+1)
    initial:    all forks on table -> all semaphores=1
endmodel
```

In trying to show many constructs in SDL the decomposition of the Dining Philosophers problem into different processes may be exaggerated (or inefficient). In the following paragraphs the SDL

description of the Dining Philosophers problem will be given. After the SDL description some other SDL-constructs (not used in the Dining Philosophers problem) will be presented.
Before we look at the SDL-syntax, let us see how SDL is structured. SDL has three abstraction levels: system level, block level and process level. System level is the top level of the SDL description. The system contains one or more block definition(s). Different blocks can communicate through channels by signals. Communication with the environment of the system also take place using channels which are connected between the system boundary and a block.

The block definition can be split in one or more block definition(s) by means of substructuring or in one or more process definition(s). Processes can communicate using signalroutes.
Processes may be split in one or more service definition(s). The bottom level definition of a process (service) is called the process-body (service-body). This process-body (service-body) defines the actual behaviour depending on input messages and conditions using a kind of flow-chart.
A schematic overview of SDL-structuring is given in the following figure.

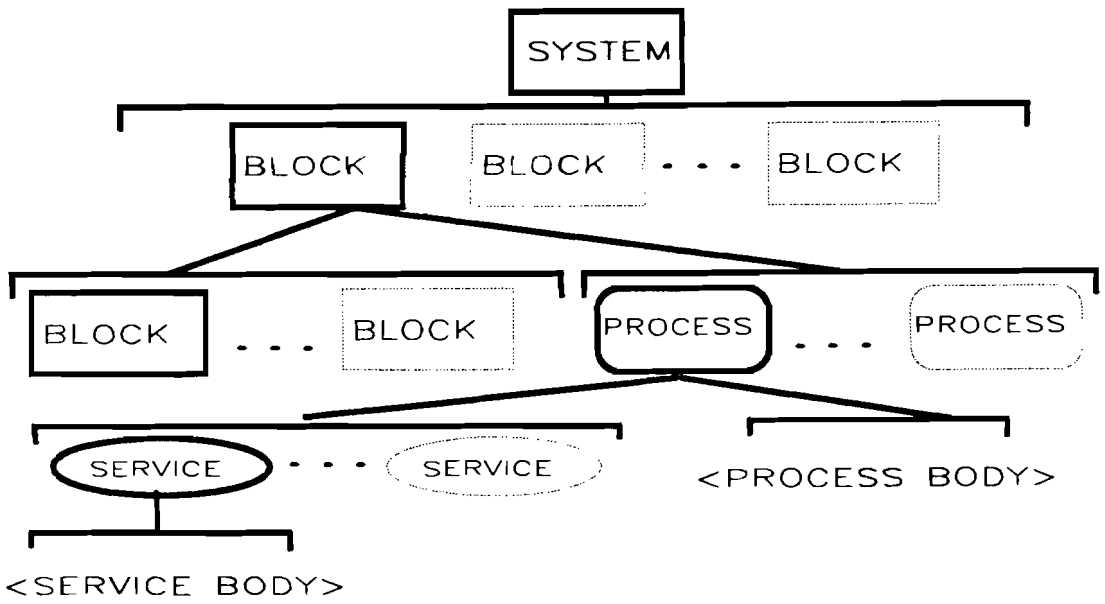


fig 4: structuring in SDL

3.1. System definition

A system-definition is the SDL representation of a specification or description of a system. A system is separated from its environment by a system boundary and contains a set of blocks. Communication between the system and the environment or between blocks within the system can only take place using signals. Within a system, these signals are conveyed on channels. The channels connect blocks to one another or to the system boundary.

The use of block references in a system-definition gives a better overview when named properly. A block reference is similar to a macro call except that it can only be used once using a reference to a remote block definition (a block definition given separately).

Besides channel-, block- and signal-definitions, the system definition may contain select-, macro- and data-definitions. The select-definition is used to insert optional system parts depending on conditional values outside the system. Macro-definitions contain system parts which are grouped together to form a logical subsystem which can be inserted using a macro call. Data-definitions contain type definitions used for signal- and select-definitions. Note: definitions of variables owned by the system are not allowed at system level so it is not possible to define global variables visible for all system parts using a variable declaration.

3.1.1. SYSTEM DINING_PHILOSOPHERS

Graphical representation (fig 5A)

The system-definition is contained by a frame symbol which forms the system boundary. The system name (DINING_PHILOSOPHERS) is placed in the upper left corner of the frame symbol. The system-definition needs one page (with number 1) which is stated in the upper right corner of the frame symbol. The text symbol (optional) contains a textual description of the system between the note-

symbol (/*<note>*/) and a signal-definition. The messages claim_forks and release_forks are of type integer (they carry an integer value with them). This integer value is used for identification of the philosopher. The message forks_free contains no explicit value, this message only has a synchronization (control) function. All messages contain implicitly the identifier of the process (PID) they come from.

Note: In general the text-symbol is used for all kinds of definitions which can not be described graphically.

The system is build of two blocks, one BLOCK FORKS and one BLOCK PHILOSOPHERS. Channel C1 can carry the messages claim_forks and release_forks from PHILOSOPHERS to FORKS. Channel C2 can carry the message forks_free from FORKS to PHILOSOPHERS. The only thing the philosophers 'see' are the forks, they do not communicate with the environment through channels.

Textual representation (fig 5B)

The textual representation is a one to one representation of the graphical representation.

SYSTEM DINING_PHILOSOPHERS 1(1)

```
/* There are 10 philosophers whose lives consist of alternately
   thinking and eating. The philosophers eat at a large circular
   table with a preassigned plate for each. Between two plates is a fork,
   which may be used by either adjacent philosopher. In order to eat, a
   philosopher must have two forks (one on his left and the other one on his
   right) */
SIGNAL claim_forks(integer),release_forks(integer),forks_free;
```

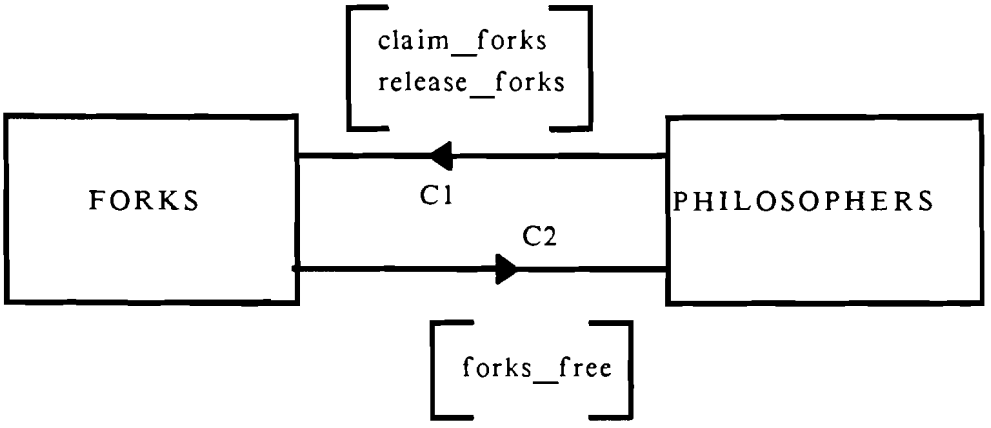


fig 5A: graphical representation SYSTEM DINING_PHILOSOPHERS

```
SYSTEM Dining_Philosophers;

/* There are 10 philosophers whose lives consist of alternately
thinking and eating. The philosophers eat at a large circular table
with a preassigned plate for each. Between two plates is a fork,
which may be used by either adjacent philosopher. In order to eat,
a philosopher must have two forks (one on his left and the other
one on his right) */

SIGNAL claim_forks(integer),release_forks(integer),forks_free;

CHANNEL C1
    FROM PHILOSOPHERS TO FORKS
    WITH claim_forks,release_forks;
ENDCHANNEL C1;

CHANNEL C2
    FROM FORKS TO PHILOSOPHERS
    WITH forks_free;
ENDCHANNEL C2;

BLOCK PHILOSOPHERS REFERENCED;
BLOCK FORKS REFERENCED;

ENDSYSTEM Dining_Philosophers;
```

fig 5B: textual representation SYSTEM DINING_PHILOSOPHERS

3.2. Block definition

A block definition is a container for one or more process definitions of a system and/or a block substructure. Purpose of the block definition is the grouping of processes that as a whole perform a certain function, either directly or by a block substructure. A block substructure is a partitioning of a block definition into a set of subblock definitions, channel definitions and subchannel definitions. A channel substructure is a partitioning of a channel definition into a set of block definitions, channel definitions and subchannel definitions. A subblock definition is a block definition, and a subchannel definition is a channel definition. This means that a subblock (subchannel) may be partitioned in subblocks (subchannels) any number of times.

The partitioning of the system in subblocks can be clarified by an auxiliary diagram containing a block tree. In this tree the subblock definitions are one level lower than the originating block definition. That is, the branches of the tree visualize the way a block is partitioned into subblocks. Only leaf blocks in the tree contain process definitions.
Note: This diagram is not part of the SDL syntax.

3.2.1. BLOCK PHILOSOPHERS and BLOCK FORKS

Graphical representation PHILOSOPHERS (fig 6A)
The block definition is contained by a frame symbol which forms the block boundary. The block name (PHILOSOPHERS) is placed in the upper left corner of the frame symbol. The text symbol contains a textual description of the block.
The block PHILOSOPHERS is build of two process definitions represented by special process symbols. Process Philosopher has at system creation zero instances and a maximum of ten instances stated by (0,10) in the process symbol. Process BIRTH has at system creation one instance and a maximum of one instance stated by (1,1) in the process symbol. Process BIRTH can create a process philosopher visualized by a create line symbol (dashed line plus arrowhead). Process Philosopher is connected with channel C1 and C2 by signalroute R1 and R2 respectively. The arrowhead of a signalroute is connected to a frame symbol to visualize the difference with a channel where the arrowhead is not connected to a frame symbol.

Textual representation PHILOSOPHERS (fig 6B)
The textual representation is a one to one representation of the graphical representation except that the create relation between BIRTH and Philosopher has no textual counterpart.

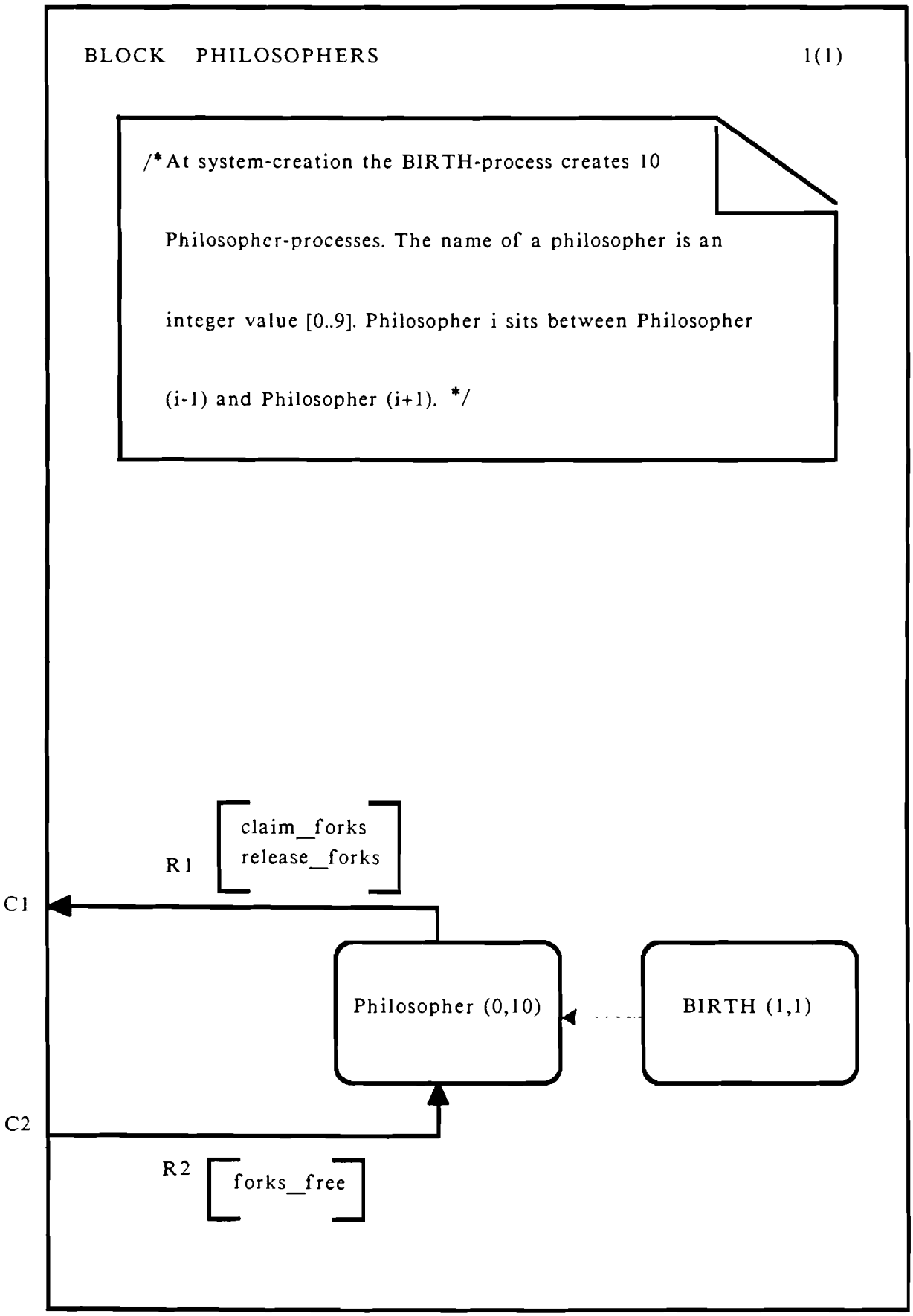


fig 6A: graphical representation BLOCK PHILOSOPHERS

```
BLOCK PHILOSOPHERS;

/* At system-creation the BIRTH-process creates 10 Philosopher-
processes. The name of a philosopher is an integer value [0..9].
Philosopher i sits between Philosopher (i-1) and Philosopher (i+1).
*/

CONNECT C1 AND R1;
CONNECT C2 AND R2;

SIGNALROUTE R1 FROM Philosopher TO ENV
    WITH release_forks,claim_forks;

SIGNALROUTE R2 FROM ENV TO Philosopher
    WITH forks_free;

PROCESS Philosopher (0,10) REFERENCED;
PROCESS BIRTH (1,1) REFERENCED;

ENDBLOCK PHILOSOPHERS;
```

fig 6B: textual representation BLOCK PHILOSOPHERS

Graphical representation FORKS (fig 7A)

The text symbol contains apart from a textual description of the block a signal definition. The signal definition contains the signals carried by the internal signalroutes. The signal test_and_set has a second type PID. Type PID is used here to identify the source process (one of the instances of process P). The input process can create P and V processes with a maximum of ten instances each. To keep track of which philosopher has claimed or released the forks the input process creates a P or V process with a formal parameter corresponding to the value of claim_forks or release_forks respectively.

Note: The use of these particular formal parameters is not stated in the block definition but is described at process level. The create action has an implicit value passing causing the updating of the values OFFSPRING in the creating process and PARENT in the created process. After successful process creation OFFSPRING has the PID value of the last created process and PARENT has the PID value of the creating process (the "parent" process). When the

BLOCK FORKS 1(1)

```
/* Claim_forks(name) causes the input-process to
   create a P-process. Release_forks(name) causes
   to create a V-process. The P-process looks to the TABLE
   to look if the claimed forks are free, and claimes them when
   they are free. The V-process frees the forks at the TABLE. */
   SIGNAL test_and_set(integer,PID),reset(integer),
          fork_free;
```

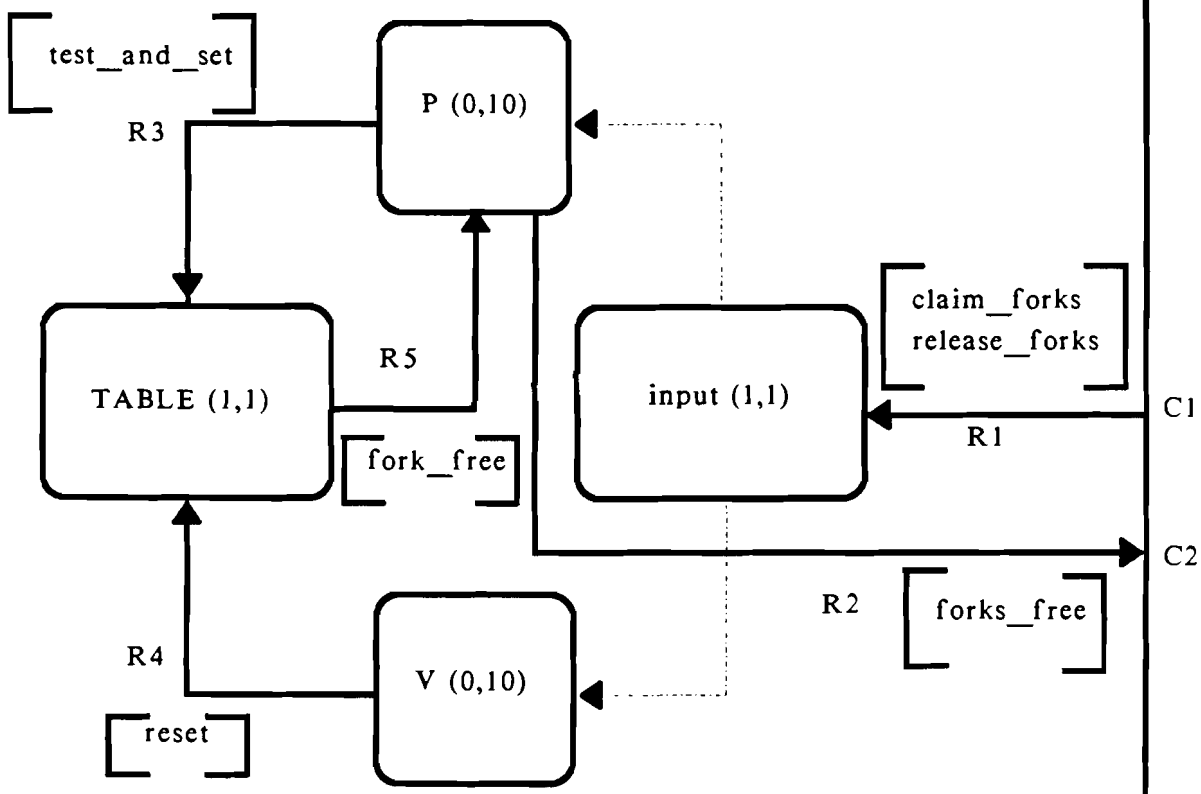


fig 7A: graphical representation BLOCK FORKS

process creation has not been successful (the maximum allowed number of processes was already created) the value of OFFSPRING will be null.

Textual representation FORKS (fig 7B)

The textual representation is (as usual) a one to one representation except for the create line symbols.

```
BLOCK FORKS;

/* Claim_forks(name) causes the input-process to create a P-
process. Release_forks(name) causes to create a V-process. The P-
process looks to the TABLE to look if the claimed forks are free,
and claims them when they are free. The V-process frees the forks
at the TABLE. */

SIGNAL test_and_set(integer,PID),reset(integer),fork_free;

CONNECT C1 AND R1;
CONNECT C2 AND R2;

SIGNALROUTE R1 FROM ENV TO input
     WITH claim_forks,release_forks;

SIGNALROUTE R2 FROM P TO ENV
     WITH forks_free;

SIGNALROUTE R3 FROM P TO TABLE
     WITH test_and_set;

SIGNALROUTE R4 FROM V TO TABLE
     WITH reset;

SIGNALROUTE R5 FROM TABLE TO P
     WITH fork_free;

PROCESS input (1,1) REFERENCED;
PROCESS P (0,10) REFERENCED;
PROCESS V (0,10) REFERENCED;
PROCESS TABLE (1,1) REFERENCED;

ENDBLOCK FORKS;
```

fig 7B: textual representation BLOCK FORKS

3.3. Process definition

As stated in [CCITT] the semantics of a process are:
A process definition introduces the type of a process which is intended to represent a dynamic behaviour.

In the number of instances the first value represents the number of instances of the process which exist when the system is created, the second value represents the maximum number of simultaneous instances of the process type.

A process instance is a communicating extended finite state machine (CEFSM) performing a certain set of actions, denoted as transitions, accordingly to the reception of a given signal, whenever it is in a state. The completion of the transition results in the process waiting in another state, which is not necessarily different from the first one.

The concept of a finite state machine has been extended so that the state resulting after a transition, besides the signal originating the transition, may be affected by decisions taken upon variables known to the process.

Several instances of the same process type may exist at the same time and execute asynchronously and in parallel with each other, and with other instances of different process type in the system.
When a system is created, the initial processes are created in a random order. The initial processes are those actions described between the start state of a process and the first (waiting) state. The signal communication between processes commences only when all the initial processes have been created. The formal parameters of these initial processes are initialized to an undefined value so they have no meaning at system creation.

Process instances exist from the time that a system is created or can be created by create request actions which start the proces-

ses being interpreted. Their interpretation starts when the start action is interpreted. They may cease to exist by performing stop actions on themselves.

Signals received by process instances are denoted as input signals, and signals sent to process instances are denoted as output signals.
Signals may be consumed by a process instance only when it is in a state. The complete valid input signal set consists of those signals in all signal routes leading to the process together with all signals defined local to the process, the implicit signals and timer signals (timer signals are signals which are placed in the input queue by a timer which has expired).

One and only one input port is associated with each process instance. When an input signal arrives at the process, it is put into the input port of the process instance.

The process is either waiting in a state or active performing a transition. For each state, there can be a save signal set. When waiting in a state, the first input signal whose identifier is not in the save signal set is taken from the queue and consumed by the process. When the signal consumed in a state is listed as input action, the corresponding transition takes place, otherwise the signal is deleted and the next signal in the queue not defined in the save signal set will be consumed.

The input port may retain any number of input signals, so that several input signals are queued for the process instance. The set of retained signals are ordered in the queue according to their arrival time. If two or more signals arrive on different paths "simultaneously", they are arbitrarily ordered.

When the process is created, it is given an empty input port, and local variables are created with undefined values assigned to them.
The formal parameters are variables which are created either when the system is created (but no actual parameters are passed to them

and therefore they are not initialized) or when the process instance is dynamically created.

To all process instances four expressions yielding a PID (Process Identifier) value may be used: SELF, PARENT, OFFSPRING and SENDER. They give result for:

a) the process instance (SELF);
b) the creating process instance (PARENT);
c) the most recent process instance created by the process (OFFSPRING);
d) the process instance from which the last input signal has been consumed (SENDER);

For all process instances present at system initialization, the PARENT expression always has the value NULL. For all newly created process instances the predefined SENDER and OFFSPRING expressions have the value NULL.

The Process description has two means of abstraction. The first one is the use of procedure definitions. The procedure is invoked in the process by means of a procedure call referencing the procedure definition. Parameters are associated with a procedure call: these are used both to pass values, and also to control the scope of variables for the procedure execution. Which variables are affected by the interpretation of a procedure is controlled by the parameter passing mechanism.

Another use of abstraction in a process definition is the use of decomposition of the process definition in services as stated in the beginning of this chapter. Service definitions have almost the same syntax as process definitions. The major differences are:
A service instance can not run in parallel with other service instances of the process instance. Within a process instance only one service instance can perform a transition at any one time.
Only one service definition is allowed to have a start containing a transition string (an initializing process).

Procedure definitions called by a service must not have states.

Special priority input and output signals can be used between services and to a service itself (using TO SELF).

Note: Service definitions in a process can be merged to one processbody with all possible combinations of service states represented as a state-vector (with the same number of elements as the number of services). All possible combinations of service states means all combinations within the restriction that only one service instance can perform a transition at any one time. At a transition only one element in the state-vector may change. If in a service a transition leads to a STOP, then the resulting process will also lead to a STOP: the process ceases to exist.

3.3.1. PROCESS Philosopher, BIRTH, INPUT, P, V and TABLE

Graphical representation Philosopher (fig 8A,8B)
In the upper left corner behind the process name (Philosopher) the use of a formal parameter: name (type integer) is stated. This parameter receives at process creation the value identifying the philosopher's place at the table. In the upper right corner is stated that the process definition takes two pages.

The text symbol contains a timer definition and a variable declaration stated by TIMER an DCL respectively. There are two timers named HUNGRY and FULL respectively. When timer HUNGRY (FULL) is expired, a message HUNGRY (FULL) is placed in the input queue of the process. The variable declaration defines a variable random_duration of type duration. Type duration is a special type especially for setting timers and related time expressions.

Although not formally stated the access of variable random_duration will return a random value of type duration.
The process begins at the start symbol followed by two initializing tasks before it enters the wait state THINK. The first task gives an informal action description GO THINK. The second task sets timer HUNGRY with the value NOW+random_duration (NOW returns the value of the actual system time).

When the timer HUNGRY expires (causing a message HUNGRY) the following transition takes place. The message claim_forks containing the value of name is put out by the process and the process enters in the next state WAIT.

When the message forks_free is received, the philosopher starts to eat, the timer FULL is set and the process enters state EAT.
When the timer FULL expires, the message release_forks(name) is
put out, the timer HUNGRY is set, the Philosopher goes thinking again and goes back to state THINK.

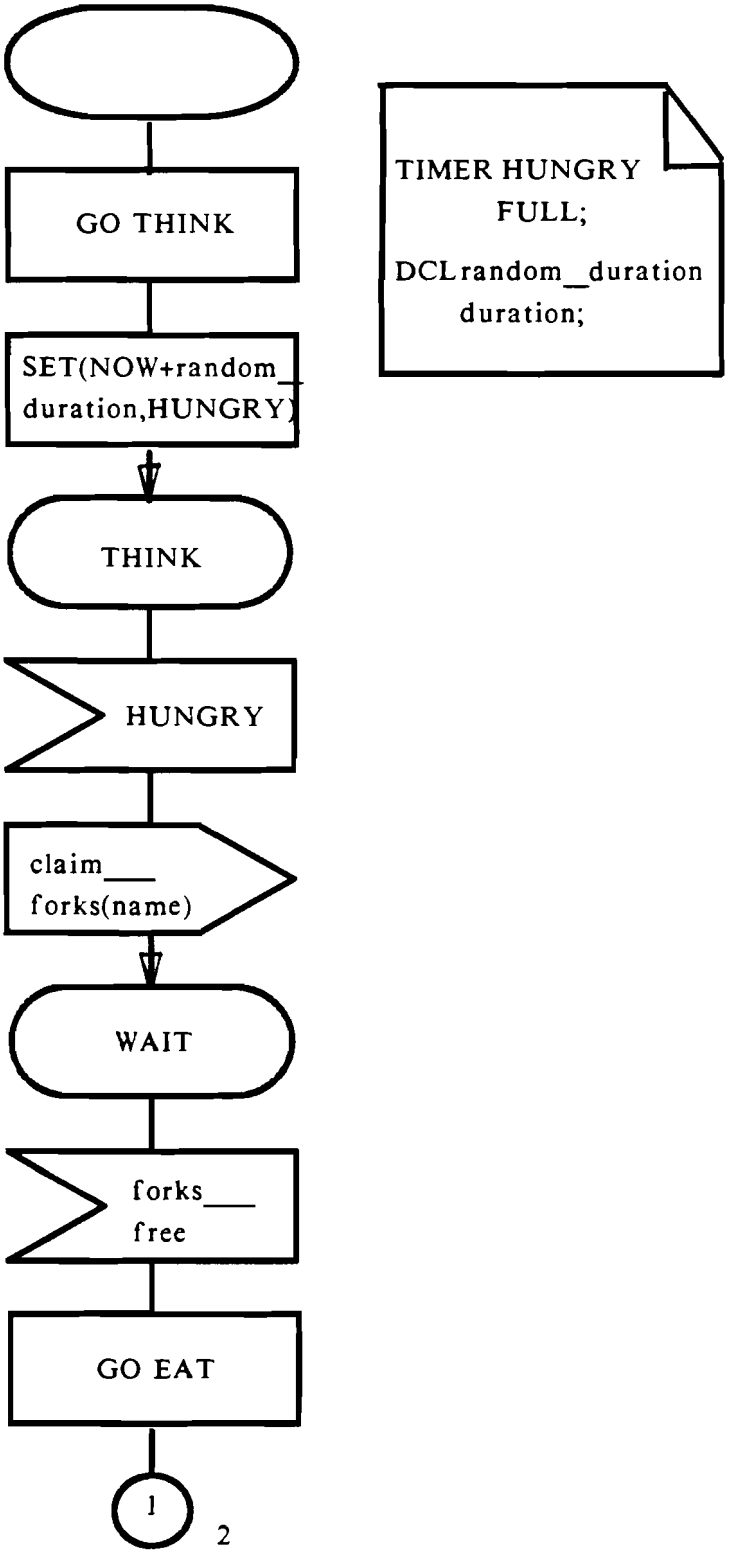


fig 8A: graphical representation PROCESS Philosopher (p.1)

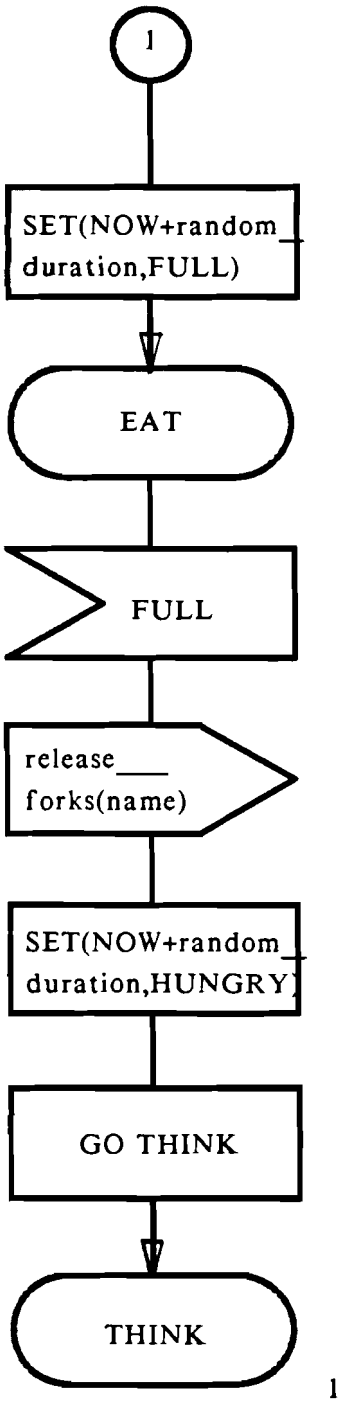


fig 8B: Graphical representation PROCESS Philosopher (p.2)

```
PROCESS Philosopher (0,10);

FPAR name integer;
DCL random_duration duration;
/* random_duration is assumed to give a random figure of type
duration when called */
TIMER HUNGRY,FULL;

START;
    TASK GO THINK;
    TASK SET(NOW+random_duration,HUNGRY);
NEXTSTATE THINK;

STATE THINK;
    INPUT HUNGRY;
    OUTPUT claim_forks(name);
NEXTSTATE WAIT;

STATE WAIT;
    INPUT forks_free;
    TASK GO EAT;
    TASK SET(NOW+random_duration,FULL);
NEXTSTATE EAT;

STATE EAT;
    INPUT FULL;
    OUTPUT release_forks(name);
    TASK SET(NOW+random_duration,HUNGRY)
    TASK GO THINK;
NEXTSTATE THINK;

ENDPROCESS Philosopher;
```

fig 8C: textual representation PROCESS Philosopher

Graphical representation BIRTH (fig 9A)

Process BIRTH creates ten Philosopher processes with different formal parameter values (0..9). This process has no waiting states, all actions take place at system creation ending with a STOP causing the process to cease to exist.

The start symbol is followed by a task which initializes i. After that process Philosopher is created with formal parameter set to the value of i. The next task increments the value of i followed by a decision whether i equals 10 or not. Upon false the process creates the next Philosopher process. When i equals 10, process

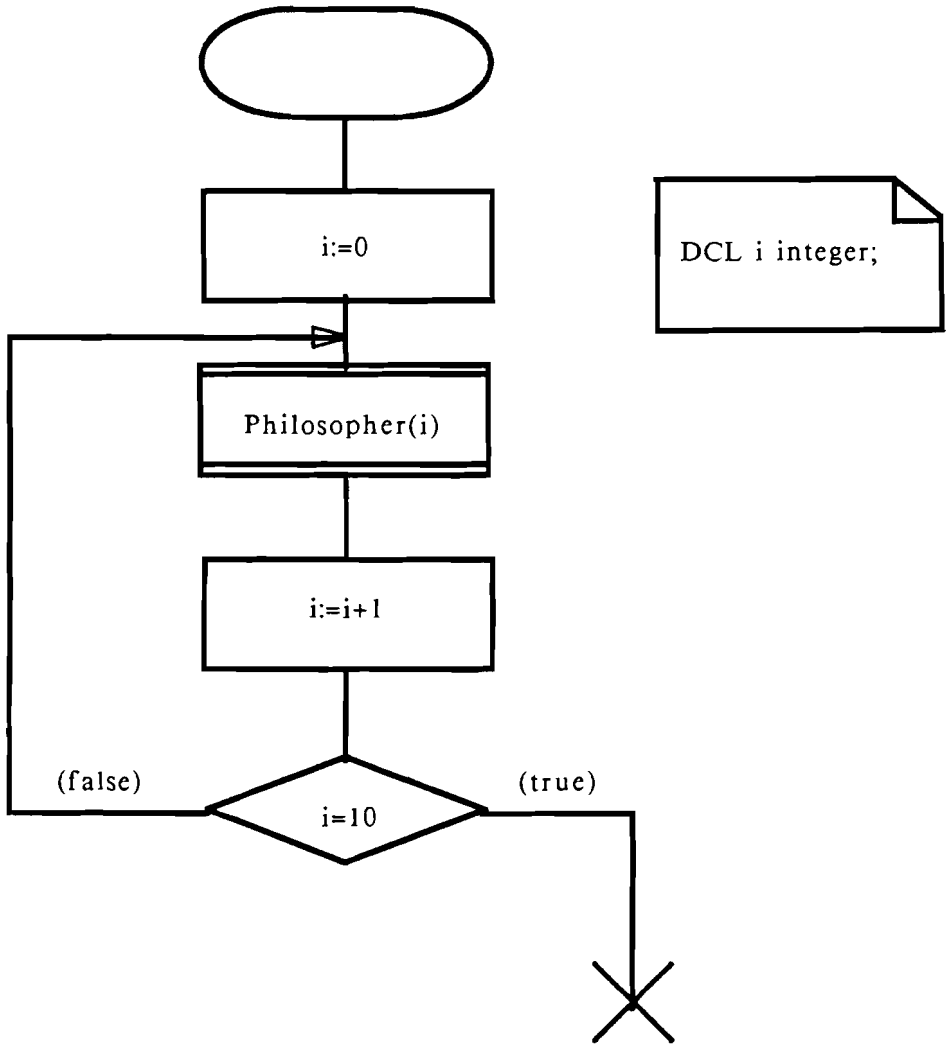


fig 9A: graphical representation PROCESS BIRTH

BIRTH ceases to exist visualized by the stop symbol.
N.B. STOP causes the immediate halting of the process instance issuing it. The signals in the input port are discarded and the variables, timers, input port and process all cease to exist.

Textual representation BIRTH (fig 9B)

The conditional loop is described by the use of a label (#A) and a JOIN statement describing the jump back (to #A).
The DECISION description has a special syntax for describing complex decisions upon any number of values or informal text. The decision-answers (stated in parentheses) must be mutually exclusive. Other ways to describe the decision in the process would be:

```
DECISION i;                    DECISION i;
(i=10):    STOP;               (i=10):    STOP;
(i<10):    JOIN #A;            ELSE:      JOIN #A;
```

```
PROCESS BIRTH (1,1);

DCL i integer;

START;
     TASK i:=0;
#A:  CREATE Philosopher(i);
     TASK i:=i+1;
     DECISION i=10;
     (true):     STOP;
     (false):    JOIN #A;

ENDPROCESS BIRTH;
```

fig 9B: textual representation PROCESS BIRTH

Graphical representation INPUT (fig 10A)

Process INPUT uses the implicit updating of the PID value SENDER at the consumption of a signal. To identify the source process (one of the Philosophers) at the creation of P and V processes, the value of SENDER is given to the first formal parameter. The value of name is given to the second formal parameter of P or V when created.

The state symbols including the bar (-) are an alternative representation for the preceding state. After the transition the process enters the same state where it started from (here the input state).

```
PROCESS INPUT (1,1);

DCL name integer;

START;
NEXTSTATE input;

STATE input;
    INPUT claim_forks(name);
    CREATE P(SENDER,name);
NEXTSTATE -;
    INPUT release_forks(name);
    CREATE V(SENDER,name);
NEXTSTATE -;

ENDPROCESS INPUT;
```

fig 10B: textual representation PROCESS INPUT

Graphical representation P (fig 11A)

Process P is created by the input process, it puts out test_and_ set messages to process TABLE. These messages contain an identifier of the wanted fork (place) and the PID of the process P itself (SELF) to be used by process TABLE when sending back the message fork_free.

Testing and setting of the forks is done one fork at a time to avoid deadlock and starvation. To avoid deadlock the philosopher

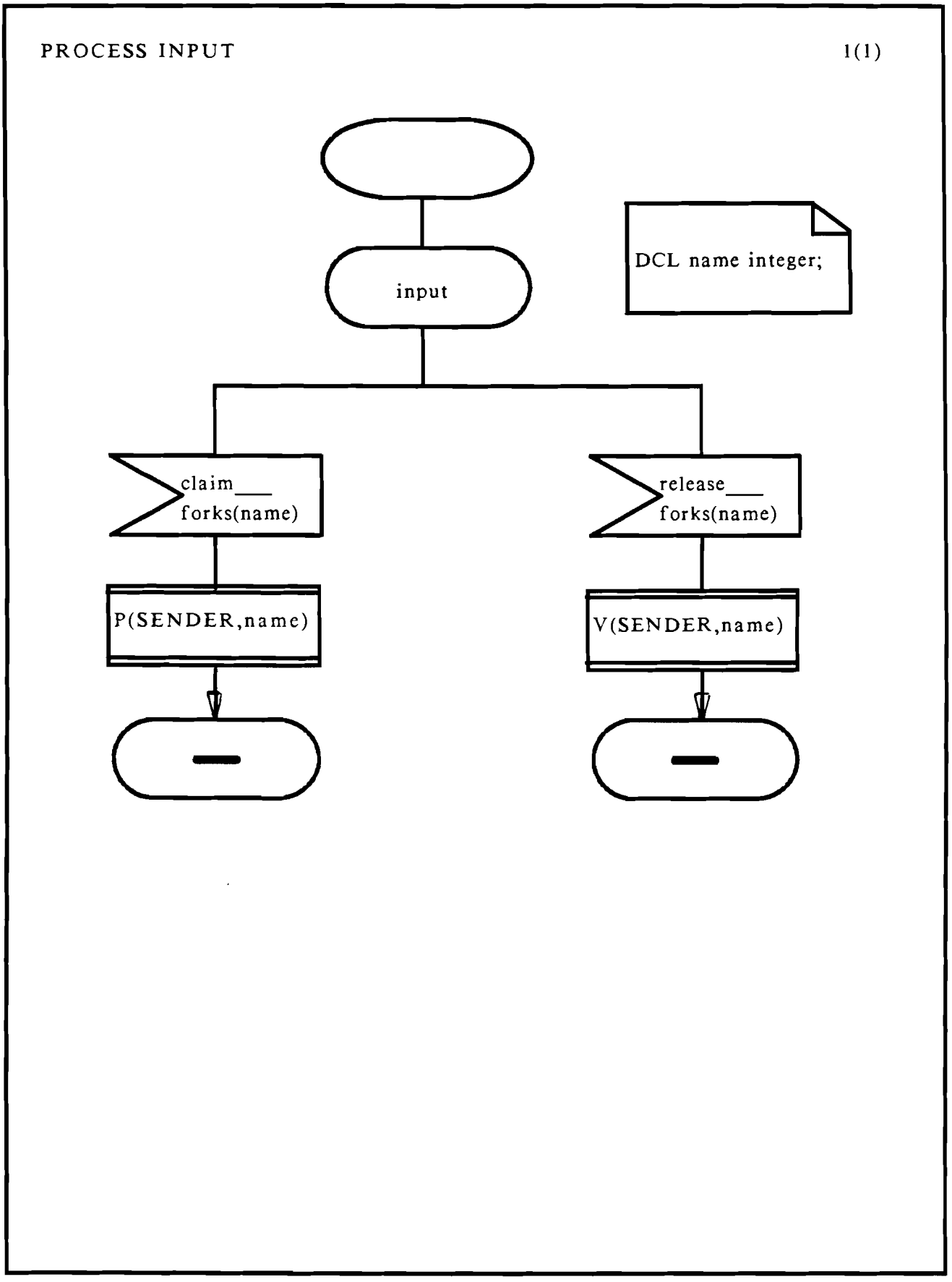


fig 10A: graphical representation PROCESS INPUT

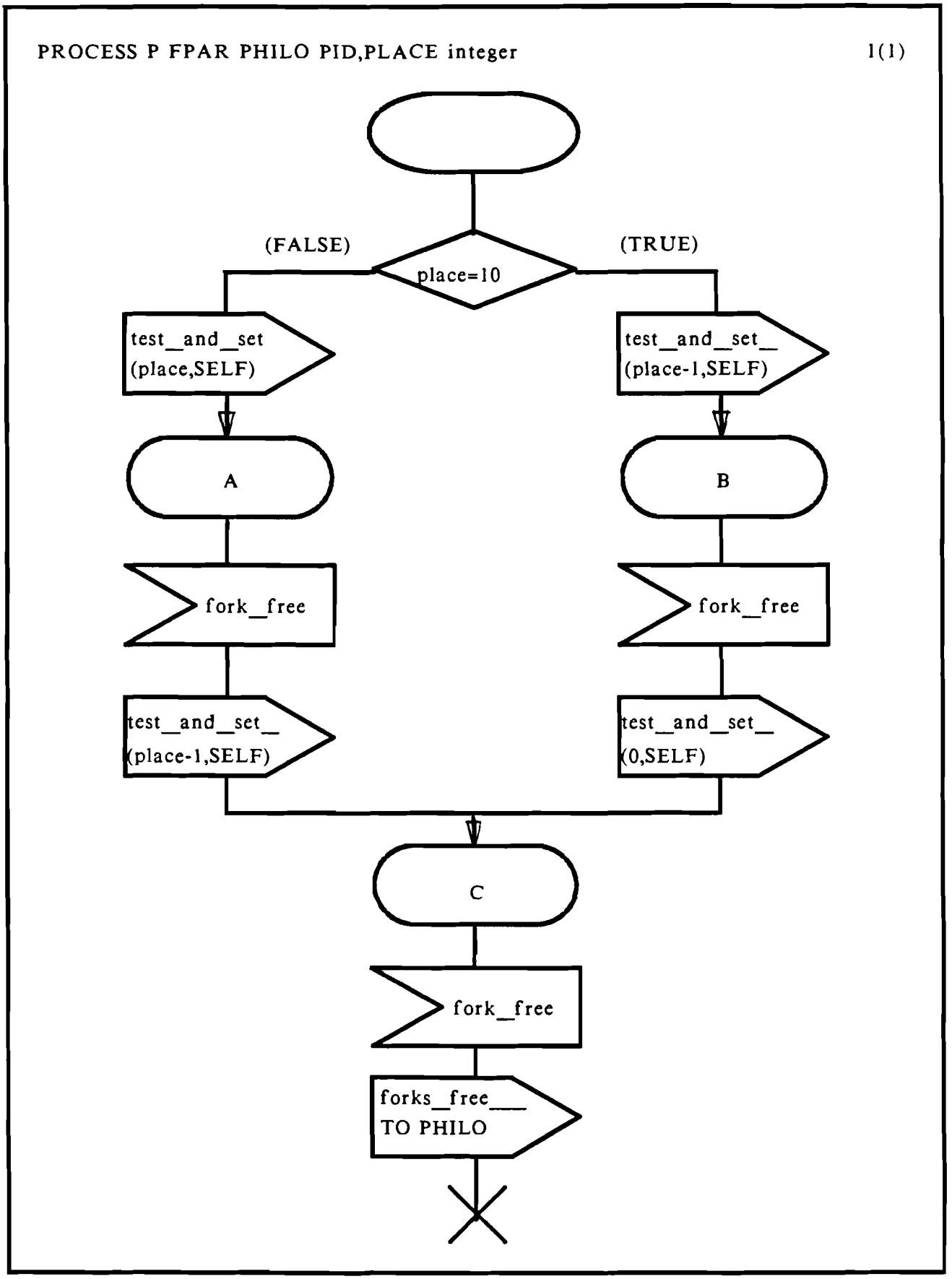


fig 11A: graphical representation PROCESS P

sitting at place 10 gets his forks in reverse order (so the situation of all philosophers having one fork waiting for the other one is not possible).
When both forks are available the process P sends forks_free to the waiting philosopher identified by the value of PHILO and the process P ceases to exist.

```
PROCESS P (0,10);

FPAR PHILO PID,PLACE integer;

START;
    DECISION PLACE=10;
    (true):     OUTPUT test_and_set(PLACE-1,SELF);
NEXTSTATE B;
    (false):    OUTPUT test_and_set(PLACE,SELF);
NEXTSTATE A;

STATE A;
    INPUT fork_free;
    OUTPUT test_and_set(PLACE-1,SELF);
NEXTSTATE C;

STATE B:
    INPUT fork_free;
    OUTPUT test_and_set(0,SELF);
NEXTSTATE C;

STATE C;
    INPUT fork_free;
    OUTPUT forks_free TO PHILO;
STOP;

ENDPROCESS P;
```

fig 11B: textual representation PROCESS P

PROCESS V FPAR PLACE integer 1(1)

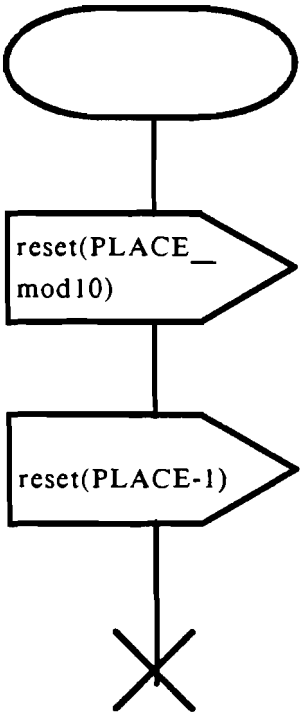


fig 12A: graphical representation PROCESS V

```
PROCESS V;

FPAR PLACE integer;

START;
    OUTPUT reset(PLACE mod 10);
    OUTPUT reset(PLACE-1);
STOP;

ENDPROCESS V;
```

fig 12B: textual representation Process V

Graphical representation TABLE (fig 13A,13B)

The first page of the process definition of TABLE shows the initializing of the flags corresponding with the availability of the forks (setting all flags because no fork is in use).

The second page shows the test_and_set mechanism and the reset mechanism. When a test_and_set message is consumed, the decision upon the availability of the fork is taken. When the fork is already taken (when fork[id]=0) the test_and_set message is put back in the input queue of process TABLE. When the fork is free, the corresponding flag is set to zero and a message fork_free is put out by the process to src (that is to the source P process of the message test_and_set).

N.B. the explicit value passing of the PID value of the P process would not be necessary when the message test_and_set is only put once in the input queue. Because then it would be possible for process TABLE to use the value of SENDER after reception of test_and_set.

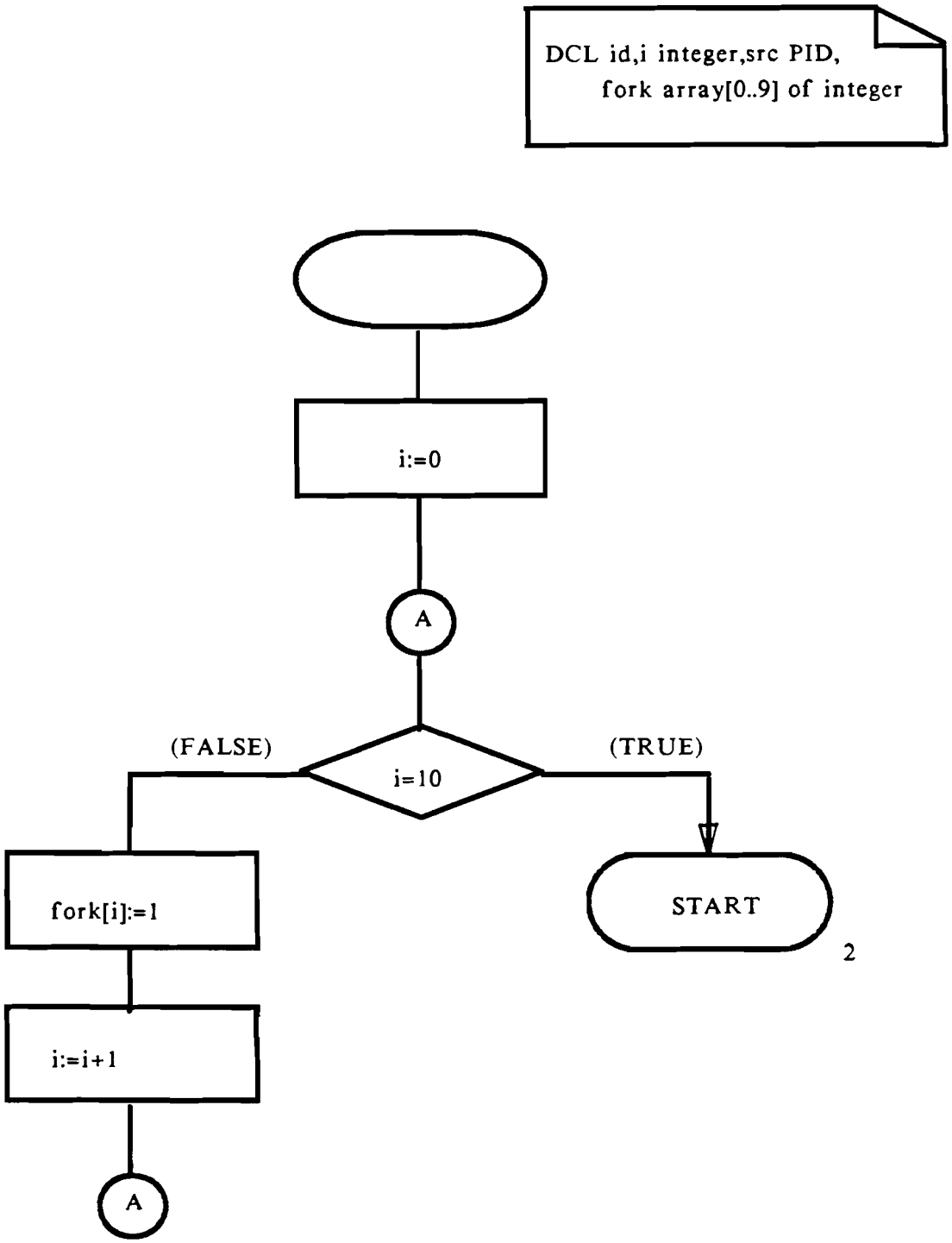


fig 13A: graphical representation PROCESS TABLE (p.1)

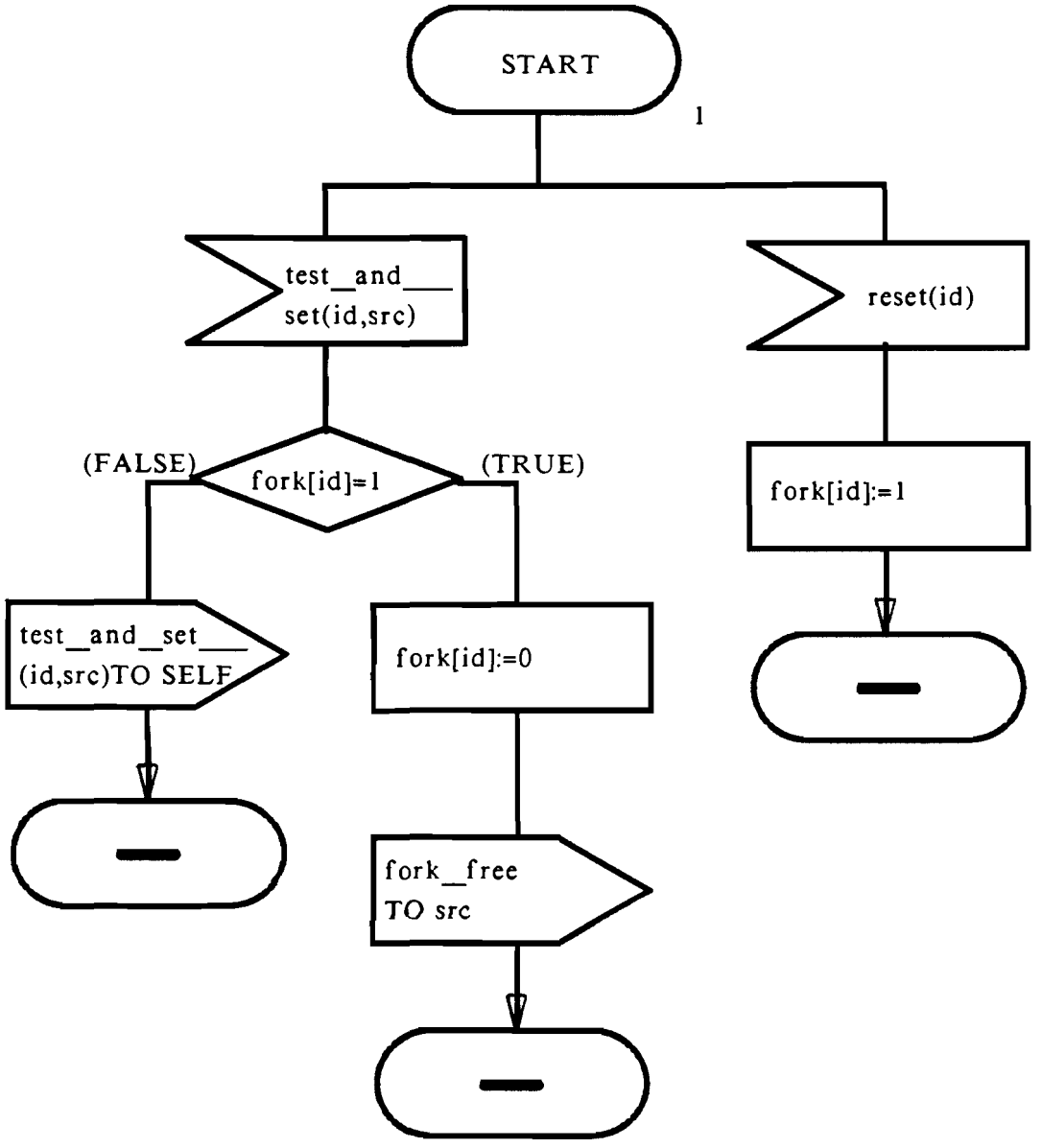


fig 13B: graphical representation PROCESS TABLE (p.2)

```
PROCESS TABLE;

DCL id,i integer,src PID,fork array[0..9] of integer;

START;
    TASK i:=0;
#A:     DECISION i=10;
    (true):
NEXTSTATE START;
    (false):    TASK fork[i]:=1;
                TASK i:=i+1;
                JOIN #A;

STATE START;
    INPUT test_and_set(id,src);
    DECISION fork[id]=1;
    (true):     TASK fork[id]:=0;
                OUTPUT fork_free TO src;
NEXTSTATE -;
    (false):    OUTPUT test_and_set(id,src) TO SELF;
NEXTSTATE -;

    INPUT reset(id);
    TASK fork[id]:=1;
NEXTSTATE -;

ENDPROCESS TABLE;
```

fig 13C: textual representation PROCESS TABLE

3.4. Other SDL constructs

SDL constructs not mentioned in the preceding paragraphs are:

- save
- asterisk- state, save and input
- enabling condition and continuous signal
- procedure call, procedure start and procedure return
- exported and imported value
- revealed and viewed value
- NEWTYPE

Except for the last construct (NEWTYPE) all other constructs are used in the SDL process definition. Let us see how they can be used.

save

A save specifies a set of signal identifiers (a save list) whose instances are not relevant to the process in the state to which the save is attached, and which need to be saved for future processing.

Graphical representation (fig 14)

A parallelogram containing a save list.

Textual representation:

SAVE <save list>;

fig 14: the save symbol

asterisk- state, save and input

The asterisk notation is used to combine states or signals to specify their common behaviour.

Asterisk state is used to describe a transition which is the same for all states contained in the asterisk state list (e.g. a reset action). The asterisk state list is transformed to a state list containing all state names of the process definition in question, except for those state names mentioned in the asterisk state list in parentheses.

Graphical representation:

A state symbol containing the asterisk state list

Textual representation:

The asterisk state list is an asterisk (*), possibly followed by a list in parentheses of one or more states (separated by commas) which are excepted.

In BNF format:

<asterisk state list>::=

<asterisk> [(<state name> {,<state name>}*)]

<asterisk>::= *

Note: All state descriptions in a process definition having the same name are joined into one state having the same name, so the actual state description is the composition of all separate state descriptions with the same name.

Asterisk input is used to describe a transition which is the same for all inputs contained in the asterisk input list.

E.g. a number of sources sending signals to one process. The signals have different names for source identification but will start the same transition. Then use of an asterisk input list is useful to visualize the common functionality of the different signals.

An asterisk input list is transformed to a signal list containing the complete valid input signal set of the process definition, procedure definition, or service definition in question, except for signal identifiers contained in other input lists and save lists of the state.

Graphical representation:

An input symbol containing an asterisk.

Textual representation:

An asterisk instead of an input signal name.

Asterisk save will cause the saving of all signals not mentioned as input signals of the attached state. The transformation of an asterisk save list to a signal list is done in the same way as the transformation of the asterisk input.

Note: It is only allowed to use either one asterisk input list or one asterisk save list in a state.

enabling condition and continuous signal

Until now a transition in a SDL process was initiated only by an input signal. The continuous signal construct allows a transition to be initiated directly when a certain condition is fulfilled. A related construct, the enabling condition, allows signal reception only when the enabling condition is fulfilled (when it's value equals True).

Both constructs use the same symbol containing a boolean expression. The difference between the two is visualized by the position of the symbol. A continuous signal is connected to a state symbol, an enabling signal is connected to an input symbol.

enabling condition

The enabling condition is evaluated before entering the state in question, and any time the state is re-entered through the arrival of a stimulus through the input port (this stimulus is generated at the arrival of input or generated by the model itself). A signal denoted in the input list which precedes the enabling condition can start a transition only if the value of the boolean expression is True. If this value is False, the signal is saved instead.

Graphical representation (fig 15)

The enabling condition symbol contains the boolean expression.

Textual representation:

PROVIDED <boolean expression>;

continuous signal

The boolean expression in the continuous signal is evaluated upon entering the state to which it is associated, and while waiting in the state, any time no stimulus of an attached input list is found in the input port. If the value of the boolean expression is True, the transition is initiated when the input queue is empty. When the value of the boolean expression is True in more than one of the continuous signals, then the transition to be initiated is determined

by the continuous signal having the highest priority (the lowest value for <integer literal name>, see below).

Graphical representation (fig 15)
The enabling condition symbol contains the boolean expression and an optional PRIORITY <integer literal name> part.

Textual representation:
PROVIDED <boolean expression>;
[PRIORITY <integer literal name>;]
<transition>

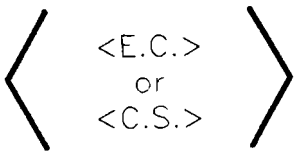


fig 15: the enabling condition symbol

procedure reference and procedure call (fig 16A)
SDL uses special symbols to call a procedure and to reference to a procedure definition.

The procedure reference visualizes that a procedure is used in a process definition. It refers to a procedure definition elsewhere (on another page). The name of the referenced procedure is stated inside the reference symbol.

Interpretation of a procedure call is replacing the call by the actual procedure graph defined in the corresponding procedure definition. The procedure call symbol contains the procedure name optionally followed by an actual parameters in parentheses. The actual parameters definition is used to pass values to the procedure's formal parameters.

There are two kinds of formal parameter attributes in the procedure definition. The IN formal variable parameter is a local variable to which a value can be assigned to when the procedure is called. The IN/OUT formal variable parameter is a synonym name for a variable defined outside the procedure. That is, an IN/OUT variable takes over the value of a global variable at procedure start and the (altered) value is assigned back to the same global variable at procedure return.
When no formal parameter attribute is given, the IN parameter is taken as default.

procedure variables are local variables within the procedure. This variables can not be revealed, viewed, exported or imported. They are created at procedure start, and cease to exist at procedure return.

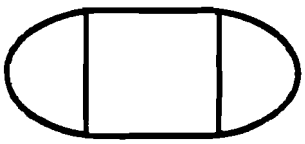
procedure reference procedure call

fig 16A: the procedure- reference and call symbols

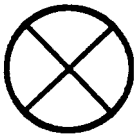
procedure start and procedure return (fig 16B)
As already stated, the procedure definition is almost the same as a process definition. To emphasize the difference, the procedure definition has its own start symbol and stop symbol (the procedure return symbol).

Other (already mentioned) differences with a process definition are:
- (trivial:) the identifier PROCEDURE instead of PROCESS,
- the use of IN an IN/OUT attributes in the formal parameter definition,

- and related to the attributes, the special properties of the local variables.



procedure start



procedure return

fig 16B: the procedure- call, start and return symbols

exported and imported value

If a process instance needs to access the value of a variable owned by another process in a different block, a signal interchange with the process instance owning the variable is needed. In this case the export/import construct can be used as a shorthand notation for this particular signal interchange without the need to specify a communication path between the processes. However the importing process needs to know the PID value of the exporting process, updating of this PID value has to be done by an ordinary signal interchange before importing is possible!

export operation

exported variables have the keyword EXPORTED in their variable definition, and have an implicit copy to be used in import operations. An export operation is the execution of an EXPORT by which the implicit copy is updated according to the current value of the exported variable.

Graphical representation:

A task symbol containing an EXPORT expression.

Textual representation:

EXPORT(<variable identifier> {,<variable identifier>}*);

import operation

For each import definition in an importer there is a set of implicit variables, all having the name and sort given in the import definition. These implicit variables are used for the storage of imported values. An import operation is the execution of an IMPORT by which the implicit variable is updated according to the implicit copy of the exported variable owned by the process with PID specified in the import expression. The association between the exported- and implicit- variable is specified by having the same identifier in the EXPORT and IMPORT expression. In addition, the exported variable and the implicit variable must have the same sort.

Graphical representation:

A task symbol containing an IMPORT expression.

A text symbol containing an IMPORTED definition.

Textual representation:

<import definition>::=

IMPORTED <import name> {,<import name>}* <sort>

{,<import name> {,<import name> }* <sort>}* ;

<import expression>::=

IMPORT (<import identifier>,<PID expression>);

revealed and viewed value

Normally the value of variables are only known by the owning process. However using the revealed construct it is possible to allow processes in the same block to view a variable using a viewed construct. Viewed variables are continuously updated as if the variable was locally defined. However a viewed variable can not be modified by the viewing process. The variable is still owned by one process.

The REVEALED attribute is put in front of variable names which are to be revealed in the variable definition. In BNF format:

DCL **REVEALED** <variable name> {,<variable name>}* <sort>

Note:

preceding variable definition emphasizes the use of REVEALED i.e. it is not a complete variable definition.

The viewed definition is as follows:

VIEWED <variable identifier> {,<variable identifier>}* <sort>;

Note:

The viewed definition may contain more variable identifiers of different sorts. The variable identifier is different from a variable name in case of more variables of different processes carrying the same name, the identifier then has to carry a qualifier identifying the owning process.

NEWTYPE

NEWTYPE is part of the SDL abstract data definition. The keyword NEWTYPE introduces a partial type definition which defines a distinct new sort. A sort can be created with properties inherited from another sort, but with different identifiers for the sort and operators. The definition contains the name of the sort, the operators defined on the sort and (optional) properties of the sort. For exact definition see [CCITT].

Note:

Definition of data is allowed at any level of a SDL description.

4. SDL and hardware implementation

SDL is a specification/description language based on communicating extended finite state machines (CEFSM). A specification in SDL is a behaviour description of a system based on the CEFSM model.
In this report it was already stated that a formal specification language describes a system in terms of functional behaviour no matter what the implementation technique is. At first glance this would mean that a SDL description does not impose restrictions on the implementation form. If we want to use SDL as a description language towards hardware implementation however, we need to impose some restrictions upon the use of SDL. In other words: the implementation form puts restrictions upon the use of SDL. These restrictions are mainly based on the fact that large input buffers are unwanted and the need for synchronization of state machines to avoid races, spikes etcetera.

Before we give a methodology to use SDL towards hardware implementation (in the next chapter), let us take a closer look at the SDL model and its implementation in hardware. The SDL model is presented in two parts. In the first part the communication model including message handling and channel structuring will be given.

In the second part we will look at the implementation of the SDL processes, the state machine model.
Next the description of two implementation forms of the SDL timer model is given followed by an implementation model of the SDL create process model.
In the last part of this chapter an example description of a universal asynchronous receiver transmitter (UART) is given. In this example some specific problems arise on the use of SDL towards hardware.
In the conclusions of this chapter a summary is given of wanted constructs and methods we need to use SDL for hardware specification.

4.1. The communication model

Communication between processes is asynchronous because the processes are not synchronized to eachother and because of the use of input queues. In related literature this communication model sometimes is called half-synchronous because the receiving process is synchronized on message reception. The name asynchronous communication is in this case reserved for interrupt driven input handling.

The model does not provide some inherent communication protocol, putting out a message by a process means putting it in the queue of the destination process. There is no control on buffer-overflow, the input queue is assumed to have infinite length. Of course a physical implementation of such a buffer is not possible.
The question arises if this model itself can be simulated using finite queues. Taking a closer look to a set of communicating processes, we see that in most cases processes have duplex signalroutes. That is, there are states of both processes where a process waits for input of the other process. The effect will be that those processes must have some maximum buffer length given by the possible output actions to the process between the wait states for input of that particular process. In this view processes without output actions are dangerous regarding to their needed input buffer length.

Another way to look at the question of infinite queues is:
If a particular process in a system specification would need an infinite queue, then this would mean that this process could never finish its job causing a deadlock because other processes can not continue at a certain moment.

In general :
A system specification does not need infinite input buffers unless the specification is wrong (deadlock).

Still this does not mean that there is no need for buffers, but only that the needed buffer-length in the model is finite and in most cases large.

The advantage of the asynchronous communication model is that it imposes almost no restrictions on the description of processes. So all attention is concentrated on the actual process tasks not bothering about signal communication. Asynchronous communication implies that processes only have wait states for input, not for output as synchronous communication has. Therefore asynchronous communication provides an easier to understand description model because transitions are triggered by input actions the same way as in state-diagrams.

Asynchronous communication needs much more hardware to implement than synchronous communication does. Therefore synchronous communication is chosen as implementation form.

Synchronous communication means that a receiving process waits to receive a message, and a sending process waits for the message to be received. Thus, the sender and receiver synchronize to exchange a message. The major drawback of this kind of communication is that the actual processing per time interval is lower than with asynchronous communication. This could be a problem for real-time processes. One way to solve this problem is the use of special protocols which use deadline information of source and destination process [Copper]. Deadlines specify how long processes are willing to wait for successful communication. When the deadline of a communicating process is reached, the process must decide whether the communication was successful or not. This decision must be equal for both processes in communication (which is taken care of by the protocol). When communication was unsuccessful the process will continue with a (specified) exception handling task.
Note:
Another solution to counter this loss in performance is the use of a separate buffer process to be described in par. 4.1.1.

The advantage of synchronous communication is that there is no need for extra buffers for communication. But this is not the only reason why synchronous communication is chosen. Later on we will see that the processes are implemented as state machines synchronized to a common system clock. It is therefore logical to choose for synchronous communication not only to avoid use of buffers but also to avoid problems with propagation delay and input clocking.
The description of the implementation of synchronous communication will be treated in the next paragraph together with the description of message handling.

4.1.1. Message handling

SDL messages can be split in two different kind of messages, control messages and data messages. Both kinds have a control function but only data messages contain data to be processed by the destination process. Identification of the source process in SDL is arranged by the implicit value passing, causing the updating of the SENDER value at message reception. Implementation of this particular value passing can be done either by vectored inputs or by space division (using separate signallines for different sources).

It is almost natural to split every SDL process at implementation in a control-unit and an operational-unit. This splitting is done the same way as proposed in [VLSI.1], where the processes are split in an information processing part (TASKs) and a control part (INPUTs, DECISIONs, ENABLING CONDITIONs, etcetera).

The same way we split processes, messages will be split in a control and data part visualized in figure 17 on the next page.
The control part of the message will start a transition as described in the SDL process definition. The data (if any) will be clocked in a register of the operational unit and will be available for further processing.

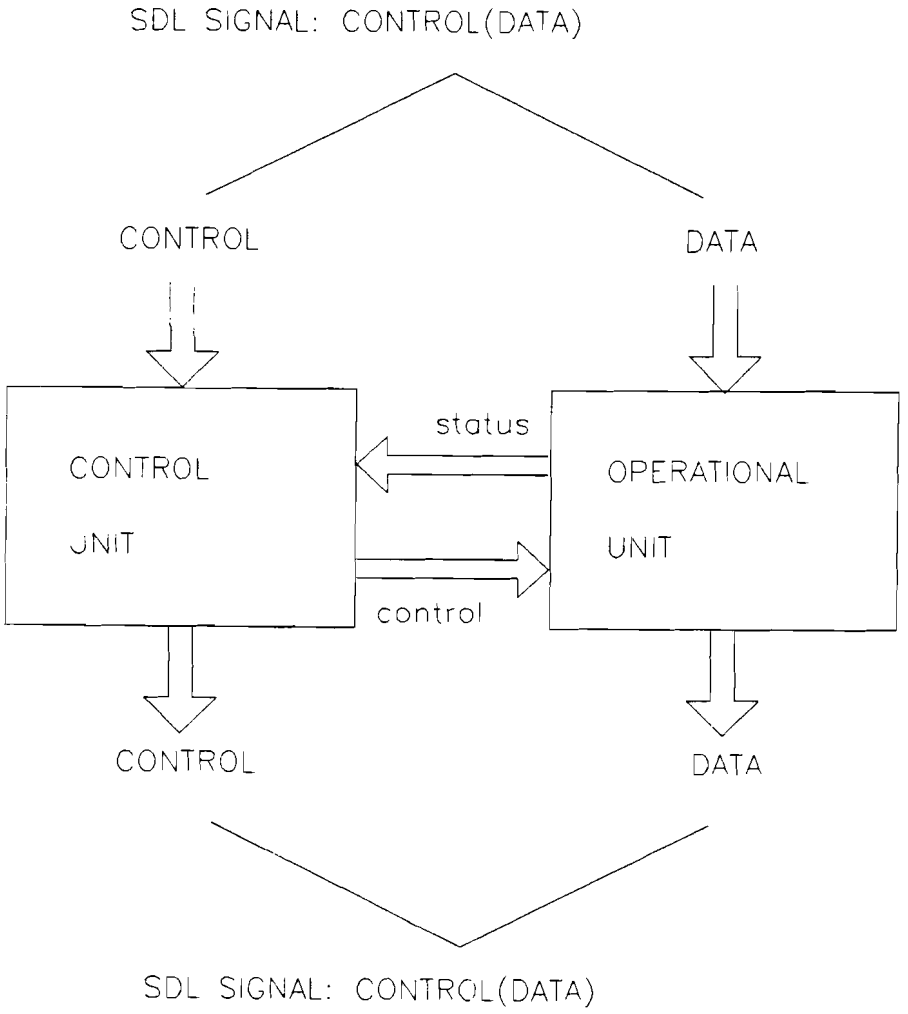


fig 17: splitting of SDL SIGNAL in data and control

The output process is very much the same as the input process. An output message contains a control part indicating that the data part (if any) is set ready to be received by the destination process.
The choice for synchronous communication implies output signals to be Moore type, so the signals are only a function of the process' state. In general Mealy output can cause unwanted oscillations or large settling times when several state machines are connected together.

Note:

The acknowledge signal back to the source process is Mealy type. This type is chosen because the acknowledge has to be an instantaneous signal not connected to a state but to one single signal reception (to avoid mis-interpretation by the source process). In this case Mealy output causes no problems related to propagation delays because the signal is expected by the source process.

Furthermore we state that produced outputs cause inputs plus specified actions (TASKs) to occur within a single clock period. So at specification level propagation delay is neglected.

The implementation of synchronous communication in its basic form is visualized in figure 18 on the next page.

We see that synchronous communication introduces an extra (wait) state in the sending process, causing loss in performance. When this loss is not wanted, a solution could be introduction of an extra output process. This process receives the output message anytime, putting it in a buffer. Now the contents of the buffer is put out by the output process using synchronous communication (in parallel with the original process). The original process only needs an extra test on buffer overflow.
Another advantage of an extra output process is that the behaviour of the resulting process is very much the same as the original process using the asynchronous communication model.

The implementation of the extra output process is illustrated by figure 19 on the next page.

In many cases the extra output process only needs a queue of length one for buffering a control message. Implementation of such a 'queue' is done by a flip-flop which can be tested and set by the sending process and reset by the receiving process (i.e. the flip-flop is the hardware implementation of a semaphore).

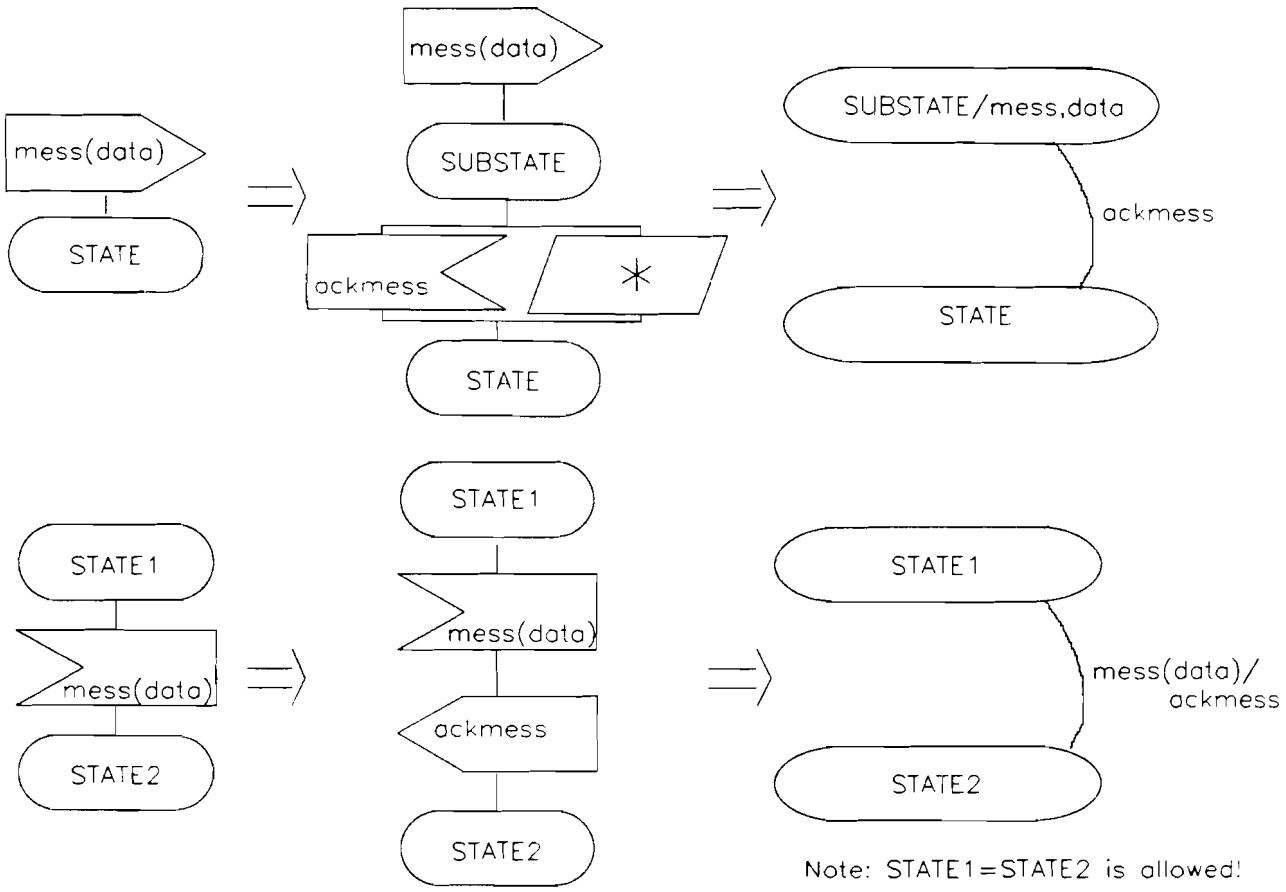


fig 18: implementation of synchronous communication

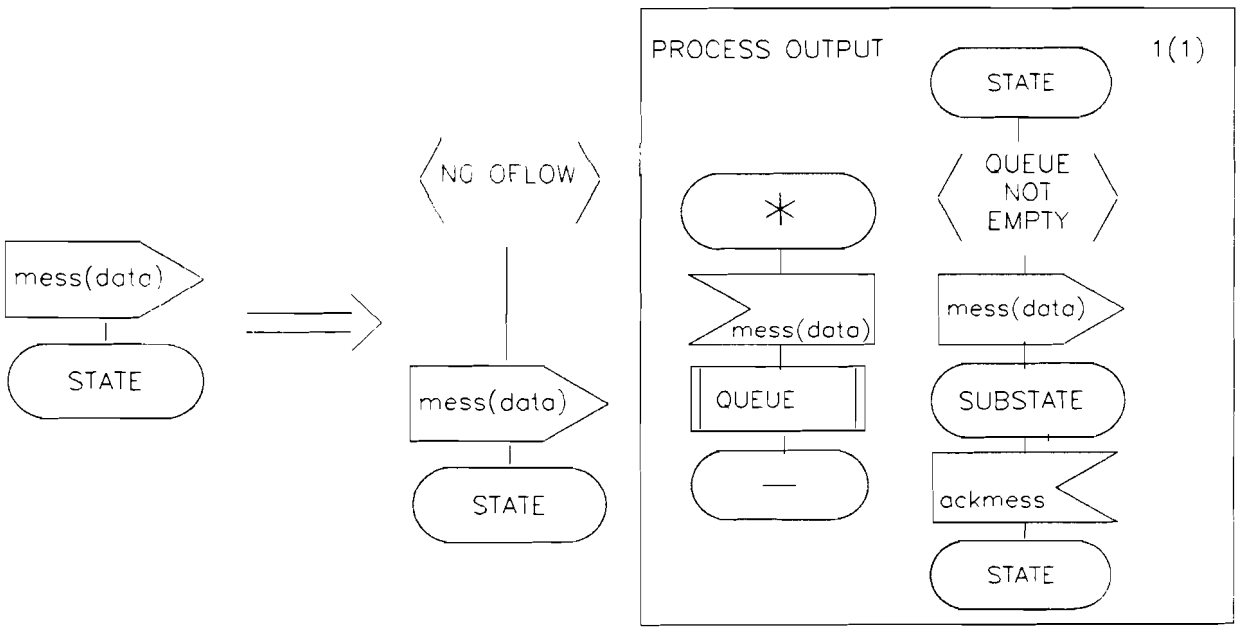


fig 19: the model of an extra output process

Note: In this case a counting semaphore is implemented by replacing the flip-flop by a up/down counter.

The situation where several processes are sending the same signal to the single destination, can be modelled in SDL in a rather elegant way. The only thing that has to be added in the synchronous communication model, is the destination of the signal ackmess, so ackmess is replaced by ackmess TO SENDER. The order the processes are serviced, is modelled by the implicit queue of the process, which means a first come first served order. When this kind of ordering is not wanted, for instance when some processes have higher priority, an extra input process is needed. The two models of this particular communication form are given in the figures 20A and 20B.

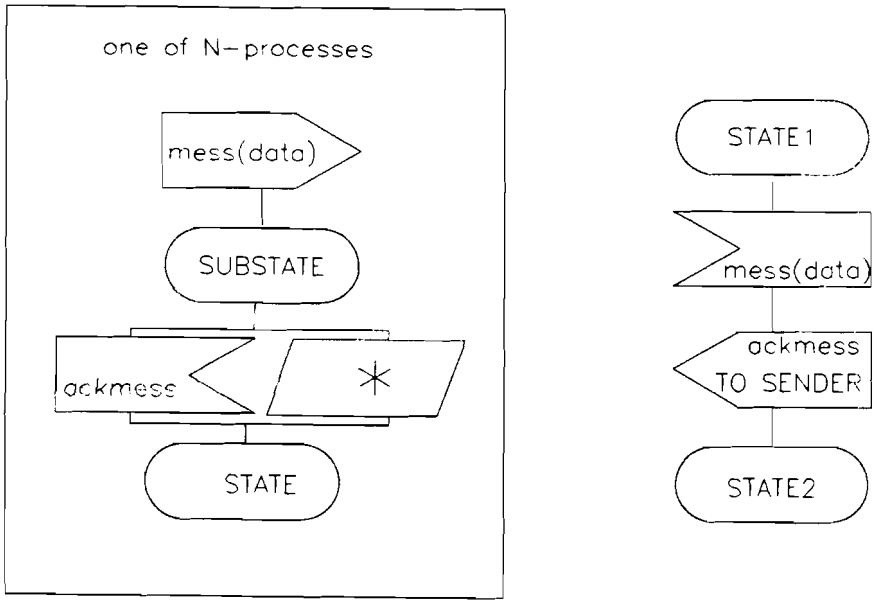


fig 20A: communication based on first come first served order

Note: The restriction that all processes send the same signal can be deleted by replacing the input action, mess(data), by an asterisk input action.

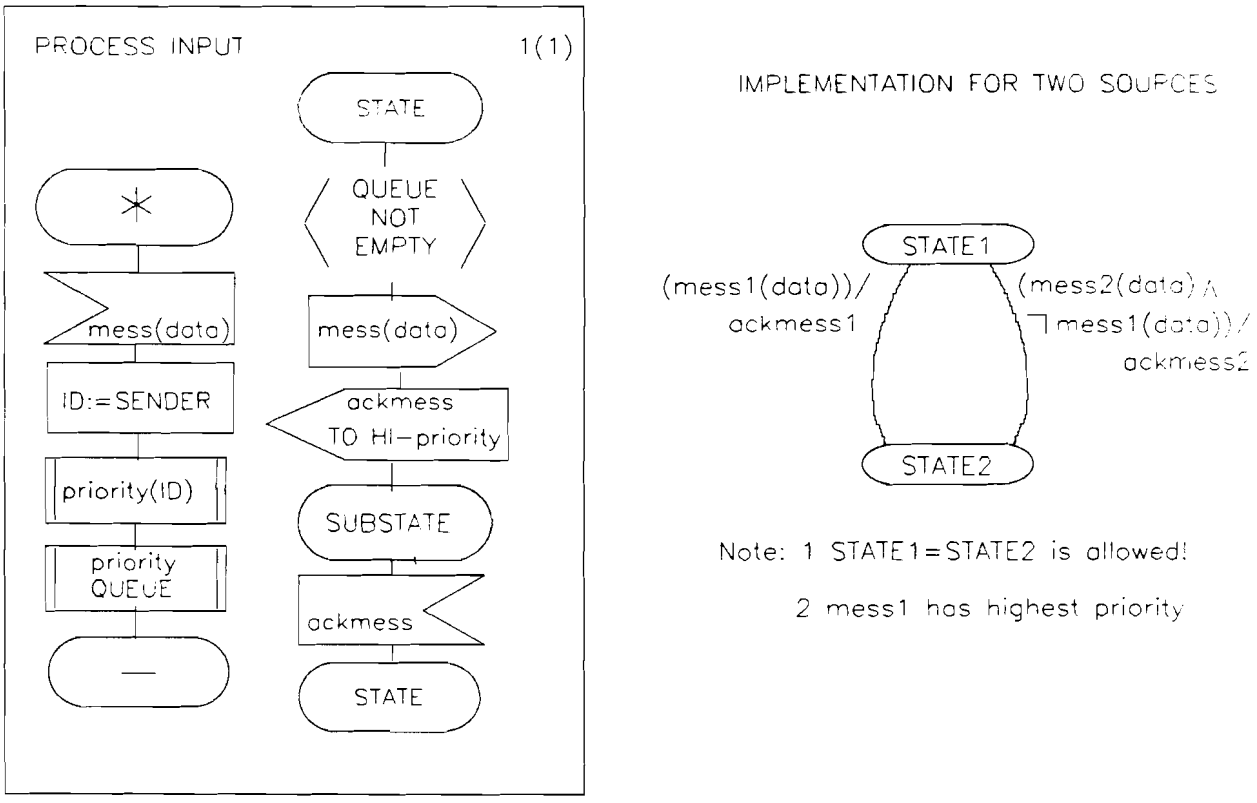


fig 20B: communication based on priority order

In the preceding example we used the implicit process queue for modelling the order processes are serviced. This kind of explicit use of the queue model not only must be allowed, when SDL is used towards hardware implementation, but must be promoted to get an easy to read (and write) SDL description. Other examples of explicit use of the queue model are to be found in the Dining-Philosophers description of the preceding chapter (in the TABLE and input process descriptions).

The save construct in SDL is used when in a certain state a particular input has to be handled first, causing the other signals to be saved. At the hardware implementation of a process using the save construct, this explicit saving is not necessary when the

processes communicate synchronously and when the saved signals have another source than the specified input signals attached to the same state. The synchronous communication takes care of the saving itself by waiting for an acknowledge.

Note:

Saving of signals from the same source as specified inputs in a particular state must be implemented in hardware using separate buffers to avoid "communication" deadlock.

The SDL model allows receiving of one input signal per transition, taking one signal at a time from the input queue. It is allowed to specify different input signals attached to a single state, giving interesting possibilities at hardware implementation. Under certain circumstances it is possible to react on more than one signal when implemented in hardware.

Those circumstances are:

1. The different input actions must start a transition leading to the same next state and
2. The concurrent behaviour of the following tasks must lead to the same result as the (sequential) combination of separate tasks. That is, the tasks must not have operations on same variables or results that can affect their common behaviour.

Another possibility is to specify the common behaviour at reception of a combination of input signals. That is we must specify some sort of exception handling. This specification is easier to give in a later stage because in SDL this kind of exception handling is difficult to specify.

Note:

In CCS [Milner] this kind of exception handling is modelled by axioms stating that certain combinations of state and queued inputs can be replaced by other states with some or all queued inputs deleted.

4.1.2. Channel structuring

SDL signals are to be carried between blocks by channels and between processes by signalroutes. In this paragraph we will speak of channels meaning both channels and signalroutes.

Channels have a message structure, they carry only the signals defined in the corresponding signal list. Channels may split in several channels, meaning that signals travel a certain direction depending on the signal lists of the divergent channels or depending on the TO/VIA constructs when a signal is defined on more than one divergent channel. Because channels can carry signals in both directions, channel splitting can also mean that at a certain point signals from different directions travel ahead together in the same direction.

Broadcasting is not possible using only the channel structure, a signal can not be split in two (or more) identical signals. Broadcasting is only possible by explicitly defining output actions with their destination for every single destination. But broadcasting is usually used to update values owned by the destination processes, so it is better to use the EXPORT or REVEALED construct. Now updating of the common values is done in one central point which can be accessed by the destination processes. The latter way of broadcasting is in some cases more realistic towards the implementation in hardware. In hardware the value is contained by one register controlled by one process with output lines to the destination processes for testing.
Note:

The broadcasting mechanism using signals is still useful when the latter solution takes to much outputlines for testing.

Channels act like FIFO's, a signal can not be passed by another on the same channel. Simultaneously putting two signals on one channel will result in a random ordering of those two signals. Channel conflicts are not possible. Implementing channels one to one in

hardware, means connecting state machines by control- and data-busses including an arbitration algorithm. This algorithm only needs the control signals to decide upon which process may use the data bus.

A one to one implementation of every channel is not realistic, because channels are rarely used. Therefore following evaluation of the system, **signals** will be assigned to central busses (including a bussharing algorithm) or to signallines only carrying that particular signal.

When deciding what implementation form the communication path between processes must have, we need to know facts about signal-rate, allowed bus integration area, relative placing of communicating state machines, needed performance and other related parameters. Therefore this implementation choice will be postponed to a later design stage. At specification the only function channels have, is visualizing the set of input and output signals a process or block has. At rewriting the behaviour model in a model with specified queue use (synchronous communication, for instance), we will state that every signal has its own channel avoiding collisions.

4.2. The state machine model

We had already decided that all processes must be synchronized to one system clock to avoid races. With the introduction of synchronous communication, the transitions upon input signals already imposed a kind of synchronization. But now all transitions will be synchronized to eachother by using the same clock.

Note:

To avoid other clock related problems of synchronous state machines, the implementation will be based on master/slave operation combined with a two phase clock as described in [VLSI.2].

Because certain tasks between states take more time than one clock period, the state will not always change within a clock period. This means that the SDL process description is not the same as an ASM chart [VLSI.1] where every active clock transition causes the change from present state to next state. Another difference between a SDL process description and an ASM chart is that a transition in SDL can only take place upon reception of certain (defined) input signals or upon True condition of certain variable expressions (continuous signal).

So states in SDL must be seen as super states containing a sequence of substates (without input signal consumption) leading to a next super state. This kind of abstraction is very useful in describing/specifying complex digital systems because in describing the behaviour only the results of tasks are important.
Note: The same kind of abstraction is used in many hardware description languages e.g. Statecharts mentioned in chapter two.

When tasks are described in a program language (C, Pascal etcetera) it is possible to automate the implementation using compilation of algorithms into hardware. This kind of compilation sometimes causes an overhead in hardware but on the other hand it saves a lot of time in designing. Regarding this, it might be useful to examine all tasks within a process to see if some hardware can be shared. The choice of an implementation form of a task then might depend on other tasks to be implemented.

In general the implementation of separate processes (without the communication part) into state machines is relatively easy when all tasks are formally described. Therefore how separate tasks are exactly implemented using ALU's, registers, counters, logic etcetera, is not described in this report, but can be found in [VLSI.1, VLSI.2, Klemann] and numerous other texts.

We will now examine which parts of the SDL process will be implemented in the operational unit and which are implemented in the

control unit.

Communication between processes is controlled by the control unit. The data part of a message is clocked in/out by the operational unit under control of the control unit via the control lines between control unit and operational unit. Control messages do not use the operational unit.

Enabling conditions, continuous signals and decisions all use the status lines between control unit and operational unit. At an enabling condition or continuous signal the status lines will be read when the control unit is in a super state. When the status line values give a True result for the enabling condition or continuous signal, the control unit will enter a new state. Otherwise the control unit will stay in its superstate. A decision is also taken using the status line values, but can take place in any state, and causes to enter a state depending on the decision value.

Tasks operating on data will take place in the operational unit and are controlled/started by the control unit. When a task takes more than one clock period, the task is split in subtasks each taking one clock period. Subtasks are controlled by substates of the control unit.

Service and procedure definitions are rewritten to/filled in process definitions before rewriting of the SDL behaviour description begins.

Start is implemented by a Master Reset state possibly followed by a start up routine controlled by the control unit (using substates) and done by the operational unit (initializing of register values, flags etcetera). The final substate of the start up routine leads to the first superstate (the first state in the SDL process description). When there is not a start up routine, the Master Reset state equals the first super state.

4.3. The SDL timer model

The SDL TIMER model makes use of the implicit process queue. The signal indicating that the timer is expired is placed in the input queue together with all other signals to be processed. Resetting the timer will cause the removal of all associated signals in the queue.

In hardware time is related to the amount of clock ticks between two events. Therefore use of timing in certain processes will be implemented as counters with preset and reset possibilities.

The SDL TIMER model will be rewritten as follows:

1. The TIMER definition will be replaced by a process definition containing a counter model.
2. The SET statement will be replaced by an output signal PRESET(time). {time integer}
3. The RESET statement will be replaced by an output signal RESET.

Note:

It is possible to let the counter process be created by the process which uses it. In this case, use of process creation visualizes the fact that a counter can be shared by more than one process.

The process definition of the created-counter model differs from the original definition in the fact that creating the process means starting the counter. In this case the preset value is given by a formal parameter at process creation. Resetting the counter will cause the process to cease to exist. When the time is expired, the process also ceases to exist after putting out the expired signal.

The control of multiple access of the counter is provided by the SDL model. In this case the number of counter process instances has an initial number of zero and a the maximum number equals the number of hardware implemented counters.

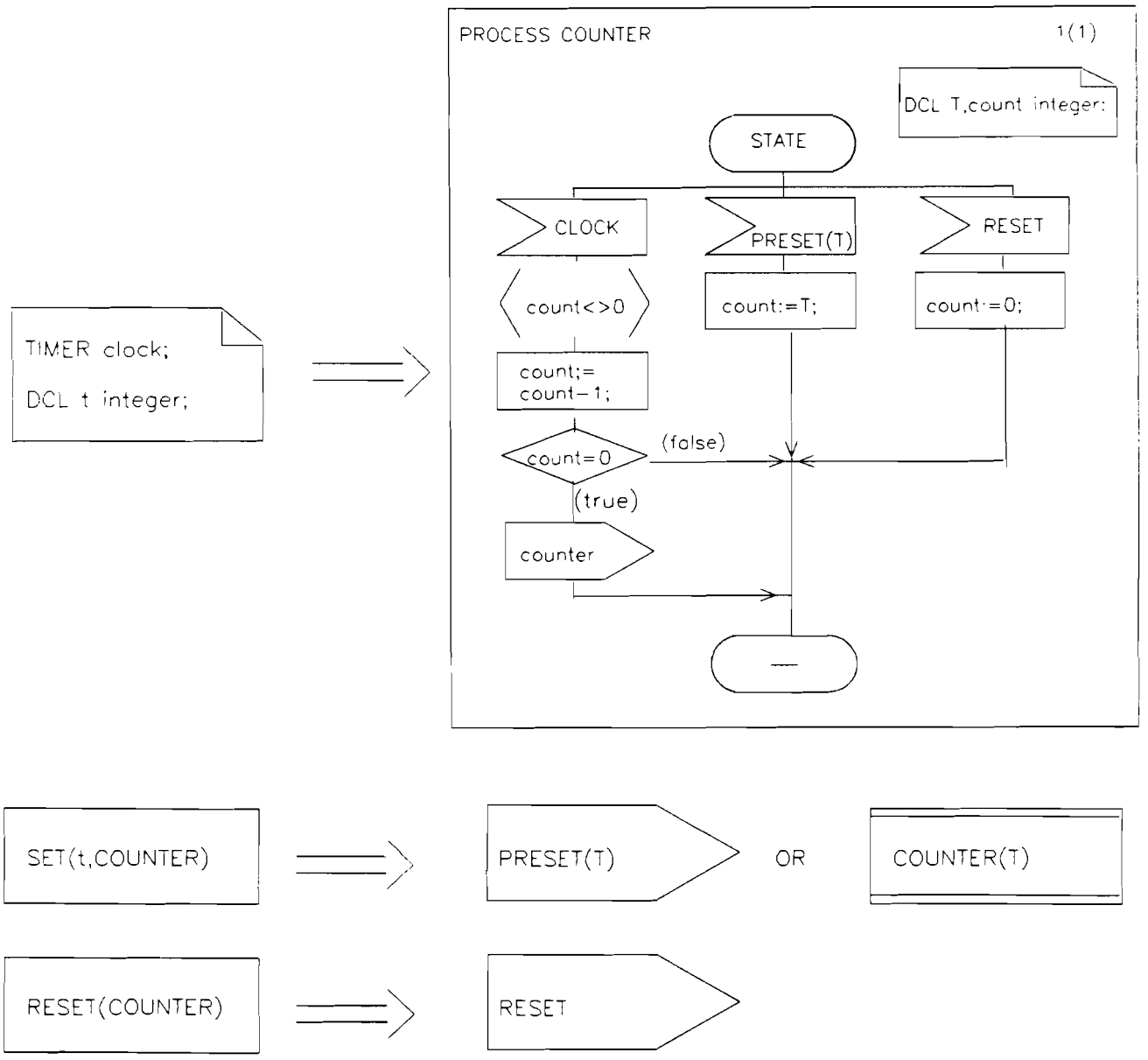


fig 21: rewriting the SDL TIMER model in the counter model

Note:

1. The identification of the clock (when more than one is used) is done by adding TO <PID counter> to the PRESET and RESET signal.
2. The counter model is an asynchronous behaviour model using an asynchronous clock signal, so it has to be rewritten towards implementation in synchronous state machines.

4.4. The SDL create process model

In the preceding paragraph we used the create construct to visualize the fact that a counter can be shared by more than one process. In general the create process construct is very useful to describe the sharing of resources by several processes.

The create model can be implemented in different ways depending on the number of parent processes, the creation rate and the maximum allowed number of created processes.

When the number of parent processes does not exceed the maximum allowed number of created processes (provided that the parent processes can create only one process per parent process at a time), the simplest implementation form is one process per parent process. That is, in this case we do not have to bother about the implementation of the updating of PARENT and OFFSPRING values.

In many cases the maximum allowed number of created processes will be kept small when these processes are implemented in hardware. In this hardware implementation we need a process to control the process creation. This control process takes care of identification of the parent process, the number of created processes and acknowledgement of successful process creation.

Special care must be taken that process creation in the SDL model does not guarantee successful process creation. The creation is not successful when the maximum allowed number of created processes is reached, causing the OFFSPRING value set to NULL and continuation of interpretation of the process graph. To be sure that the process will be created, the create symbol must be followed by a decision upon OFFSPRING=NULL. When this decision gives a True value, process creation will be tried again, otherwhise interpretation of the process graph continues.

Note: an alternative for waiting until process creation is successful is specifying an exception handling upon

OFFSPRING=NULL giving the same result as successful process creation.

When the create process model is implemented using a separate control process, the process graph will be rewritten as follows:

The create symbol with the decision upon OFFSPRING=NULL attached will be replaced by an output create_process followed by a substate with input successful_creation(pc) (pc is the PID value of the created process). (see fig 22 on next page)

Note:

The contents of the signal successful_creation (the PID value of the created process) can be used for specifying direct control (for instance resetting) of the created process.

Communication between created process and parent can be implemented using a data switch controlled by the separate create_control process.

The description of the created process will be rewritten in a process description with initial number and maximum number of instances set to the maximum allowed number of created processes. The first input action of this process will be start_process(PARENT) from the control process. The PARENT value can be used to communicate directly with the parent process.

The process stop symbol will be replaced by an output signal process_stop (to the control process) with the stop symbol attached to it.

Note:

The create line symbol in the block level description will be replaced by a signalroute carrying the signals:

create_process and successful_creation.

The process symbol of the created process will be split in a control process and the actual (rewritten) process. A signalroute between these processes carries the signals:

start_process and process_stop.

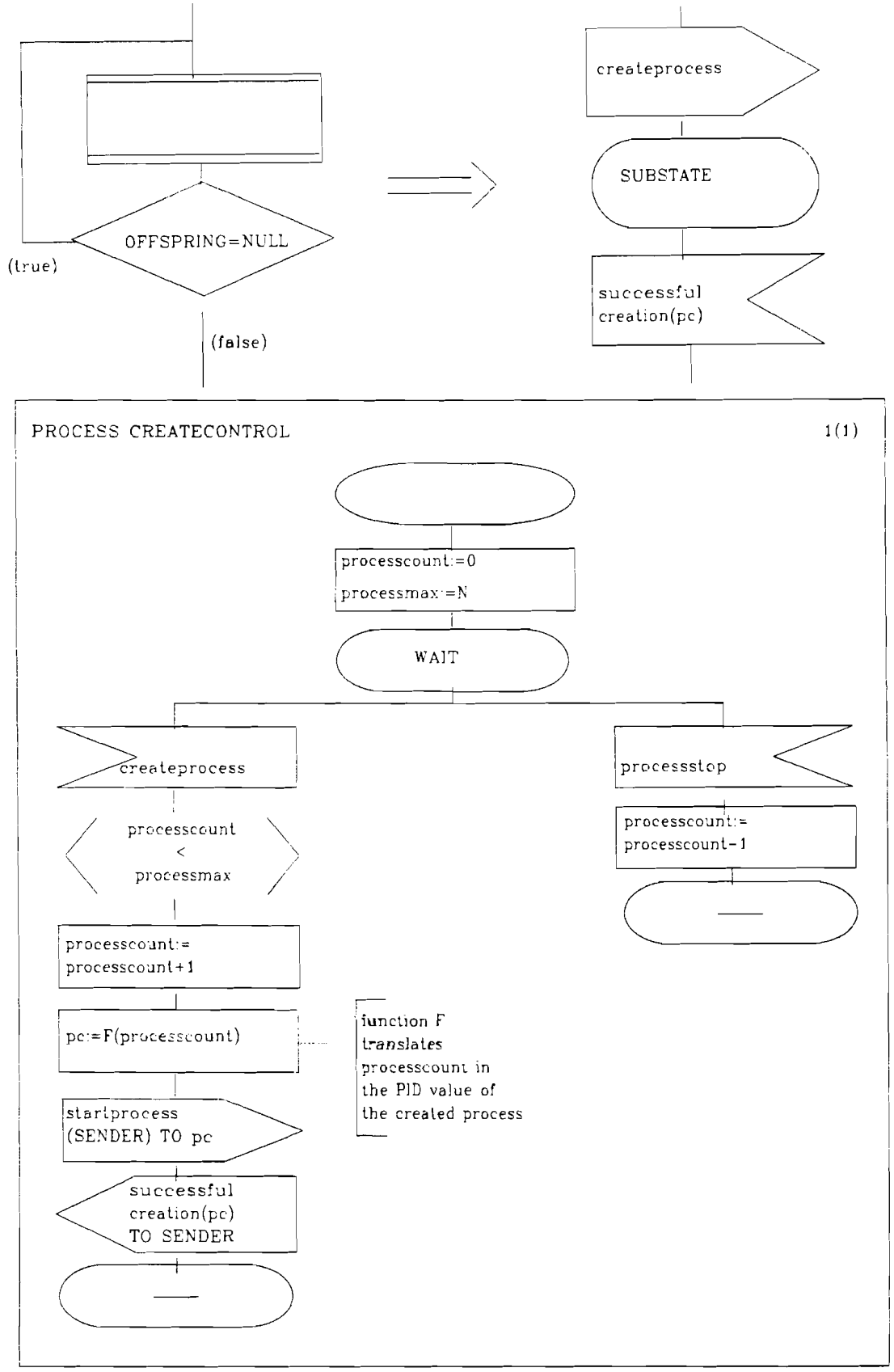


fig 22: the implementation model of the SDL create process model

4.5. SDL example description of a universal asynchronous receiver transmitter (UART)

In this paragraph a SDL description is given of the UART design described in [Verschie]. This description is made following the functional decomposition and gate level description of the UART. That is, this SDL description is not a behaviour description (without implementation knowledge) as first step of the design. But this description must be seen as an example of using SDL for description and functional decomposition of a digital system. Therefore, some parts (the testfacilities and internal loopback) of the UART design are not given in the SDL description and some parts are not described at process level.

A UART is an interface between parallel data and serial data transmission. It is used for communication between CPU's via a serial link. The UART takes care of the message format, buffering of messages, error detection, modem control, interrupt handling and other interface tasks.

Note: The UART design described in [Verschie] is a re-design of the ic 8250.

SYSTEM UART (fig 23)

Let us see what *is* in the text symbol.
First it is stated that the input mr (master reset) is defined as a variable visible for all processes in the UART system. This is an informal description of a variable which is owned by some process (possibly a special input process), and exported by that process to all other processes. This description is done informally because the formal description using export mechanisms, signals, an extra process and extra channels/signalroutes takes to much room and makes the description difficult to read.

The second informal description about mr says that when mr is true all processes will be initialized. Meaning that mr is a boolean variable causing to reset all processes (starting them from process start) when its value is true. This is an informal way to define that every process, in every state, has a transition upon mr=true to its start state.

SDL is not bit oriented, the datatypes bit and byte are not standard. Therefore bit and byte are defined as (referenced) NEWTYPEs. The actual type definition is not given because it is difficult to do this properly within the abstract datatyping and because at this point the exact definition is of less importance.

The SIGNALLIST definitions are used to combine different (related) signals to a set of signals identified by the signallist name. Use of signallists is very useful to save room in the signal definitions connected to channels/signalroutes.

The abbreviated signal names in the signallists and signal definitions are explained in Appendix A.

The last informal definition saying that a change of a signal value automatically causes a signal to be placed in a queue is given to cope with the level driven nature of the environment in the message driven SDL model. Regarding this, a clock signal must be seen as a continuous stream of messages, one message per clock change. The latter may seem unnatural but emphasizes the fact that transitions are driven by the event 'clock change'.

The UART_BLOCK has three channels. Channel C1 is the channel connected to the CPU, with the data bus, control bus and interrupt line. Channel C2 is the channel connected to the environment of the CPU system, with the duplex serial line and the control bus connected to the modem. Channel C3 carries all clock signals important for synchronization.

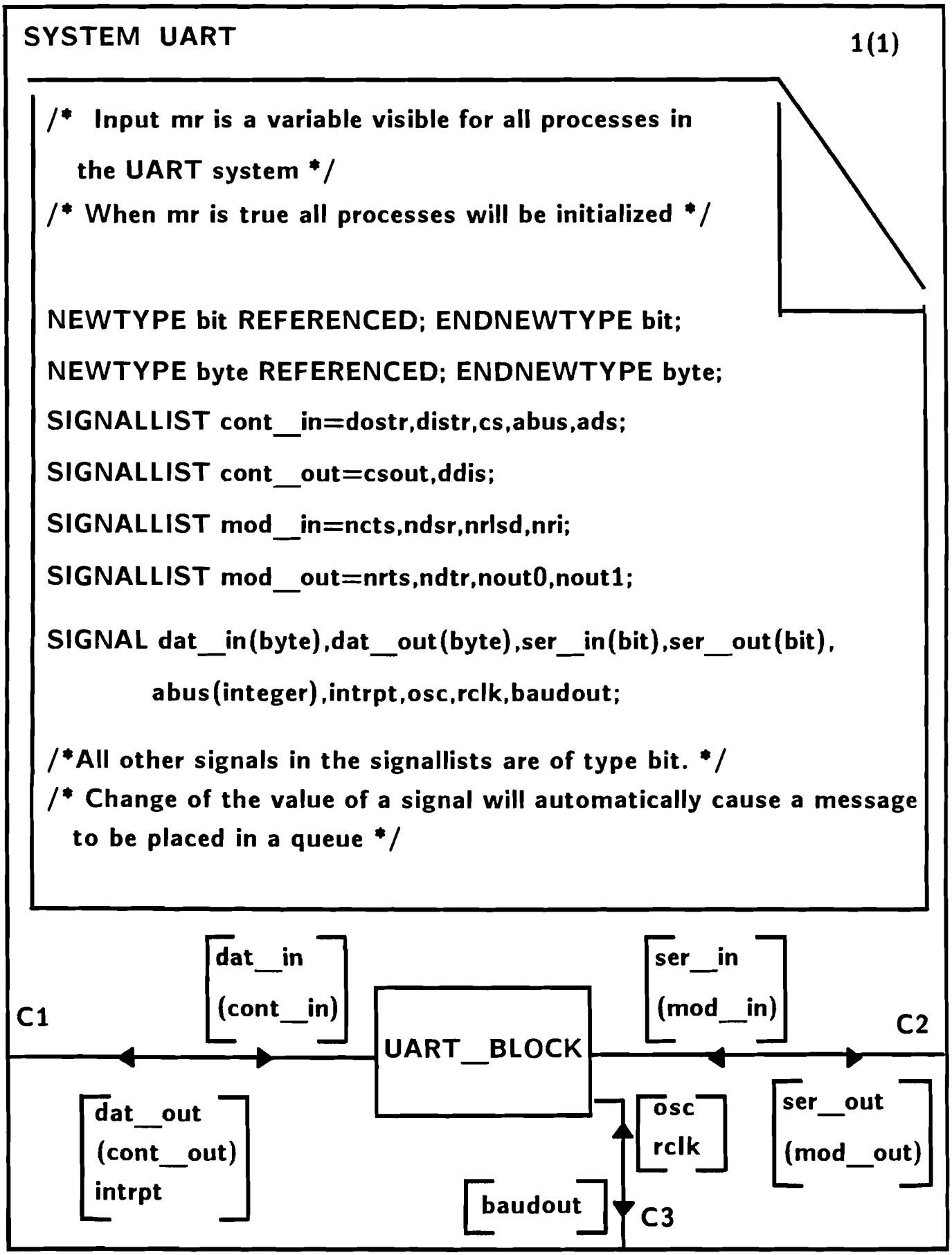


fig 23: SDL description: SYSTEM UART

BLOCK UART_BLOCK (fig 24)

The UART_BLOCK is divided in 4 blocks (substructuring).

The bus_interface distributes data from/to the CPU to/from the other blocks depending on the control messages from the CPU.
The interrupt_control handles all interrupts from other blocks depending on the interrupt mask which can be read/written via the bus_interface.
The transceiver is the main part of the UART_BLOCK which takes care of the message format, buffering of messages and error detection. Only this part's decomposition will be described.

Note:
Write actions take only one signal (the data and control is combined) and read actions take two signals, namely a request to read (control) and a data signal back.

The abbreviated signalnames in the signallists are explained in Appendix A.

BLOCK TRANSCEIVER (fig 25)

The first comment in the text symbol says that the variables contained by the line_control and line_status register are EXPORTED variables.
The second comment speaks for itself.

BLOCK TRANSCEIVER is divided in 4 blocks:
TRANSMITTER, TRANSCEIVER_CONTROL, RECEIVER and BAUDRATE_GENERATOR
The functions of the different blocks are at this point sufficiently described by their names.

Only the decomposition of the transmitter block and transceiver _control block will be described.

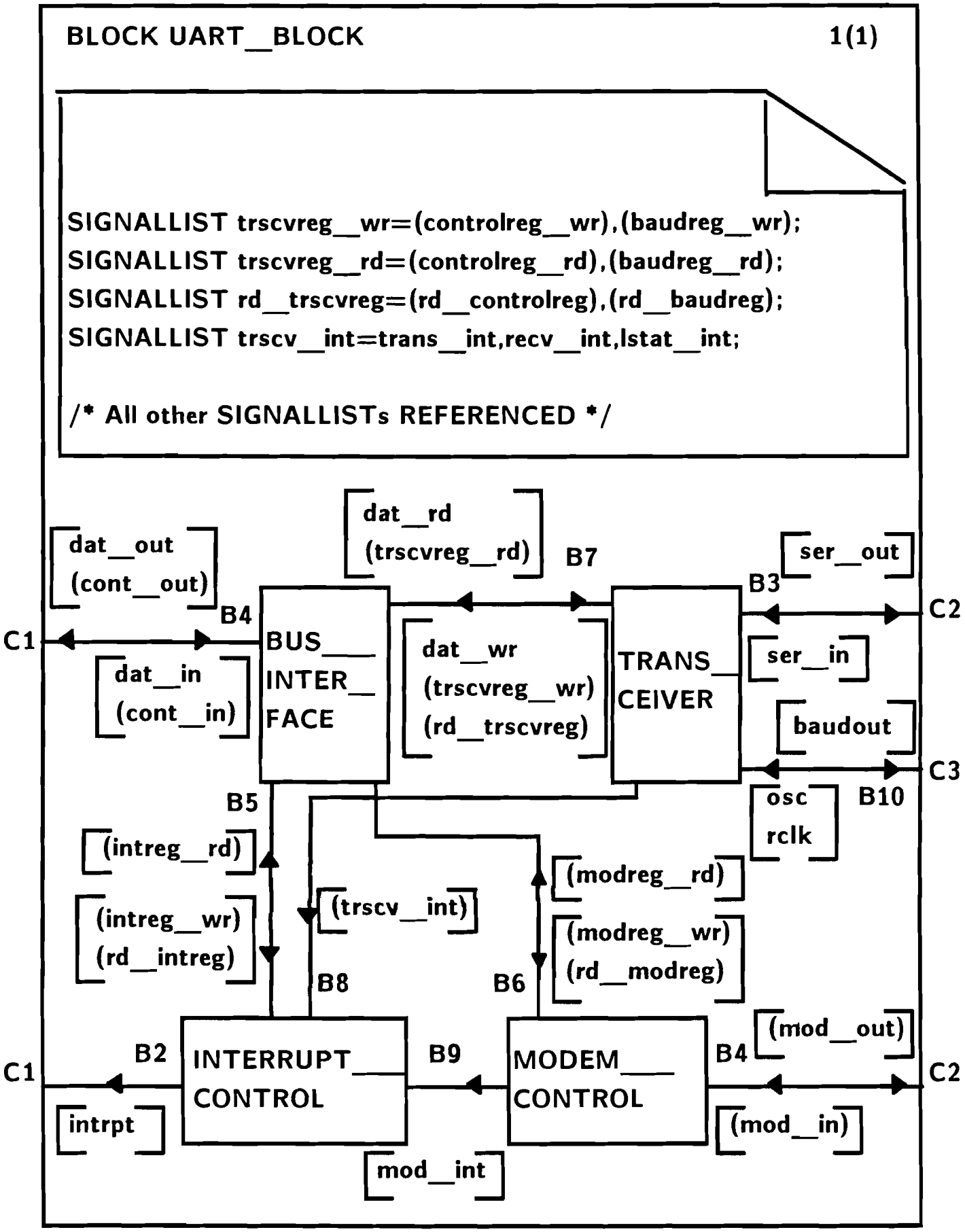


fig 24: SDL description: BLOCK UART_BLOCK

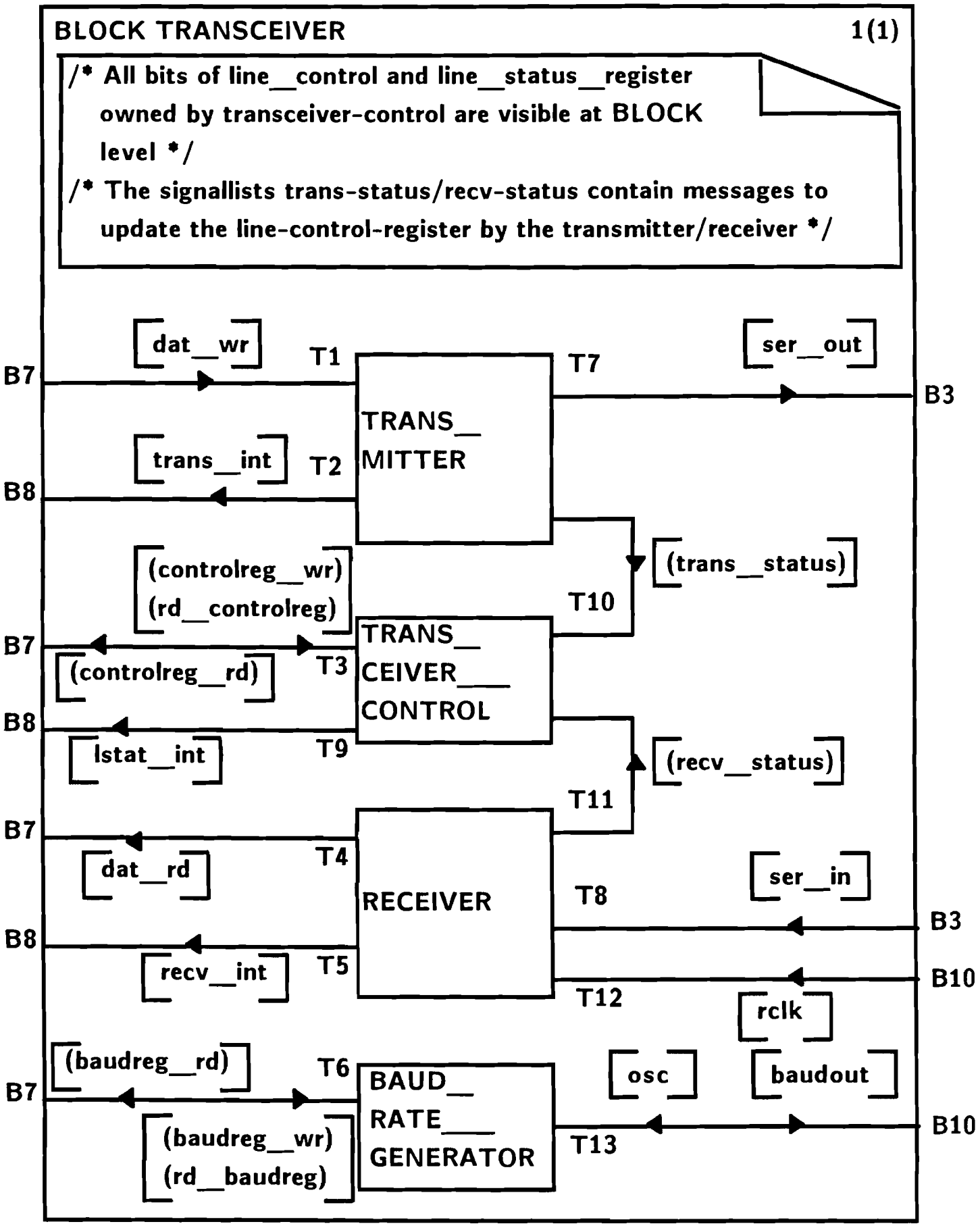


fig 25: SDL description: BLOCK TRANSCEIVER

BLOCK TRANSCEIVER_CONTROL (fig 26)

The only process contained by the block TRANSCEIVER_CONTROL is the process TRANSCEIVER_CONTROL. This process owns the variables of the line_control and line_status register visible to all other processes in block TRANSCEIVER. Updating of the status-register by the TRANSMITTER and RECEIVER is done via the signalroutes TC3 and TC4 respectively.

BLOCK TRANSMITTER (fig 27)

The figure speaks for itself.

PROCESS TRANSCEIVER_CONTROL (fig 28A, 28B)

The line control register is represented by the lcr variable of type STRUCT. This type has almost the same format as a record type in Pascal except for some special operations defined upon the STRUCTed variable type.
The line status register is also based on a STRUCTed variable.

The first page of the process definition describes the transitions upon reception of signals from the bus_interface. These signals are caused by read/write requests of the control- and status- register by the CPU.

The second page describes the transitions upon the reception of signals from the receiver and transmitter. These signals automatically update the status register variables stated in parentheses of every input action.

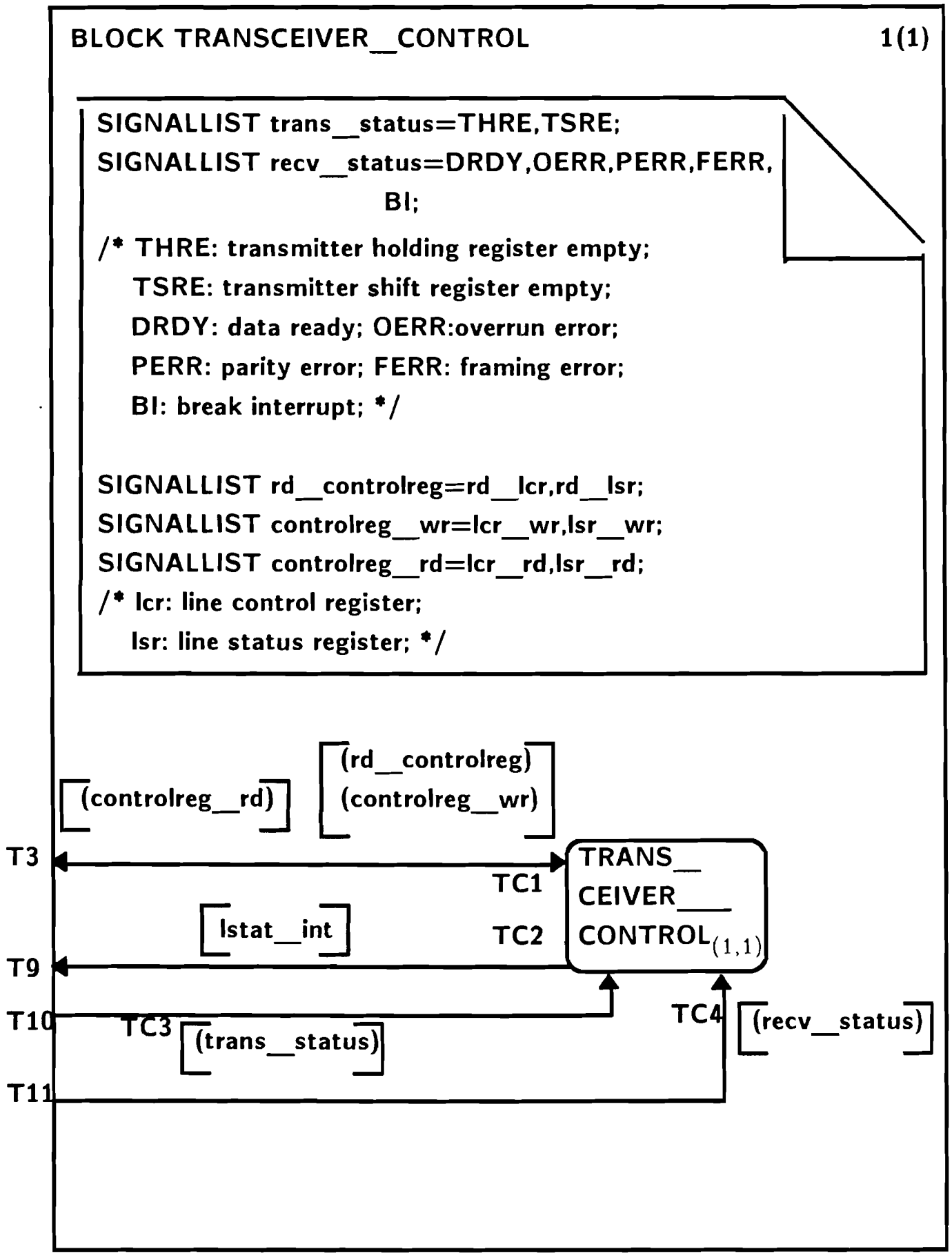


fig 26: SDL description: BLOCK TRANSCEIVER_CONTROL

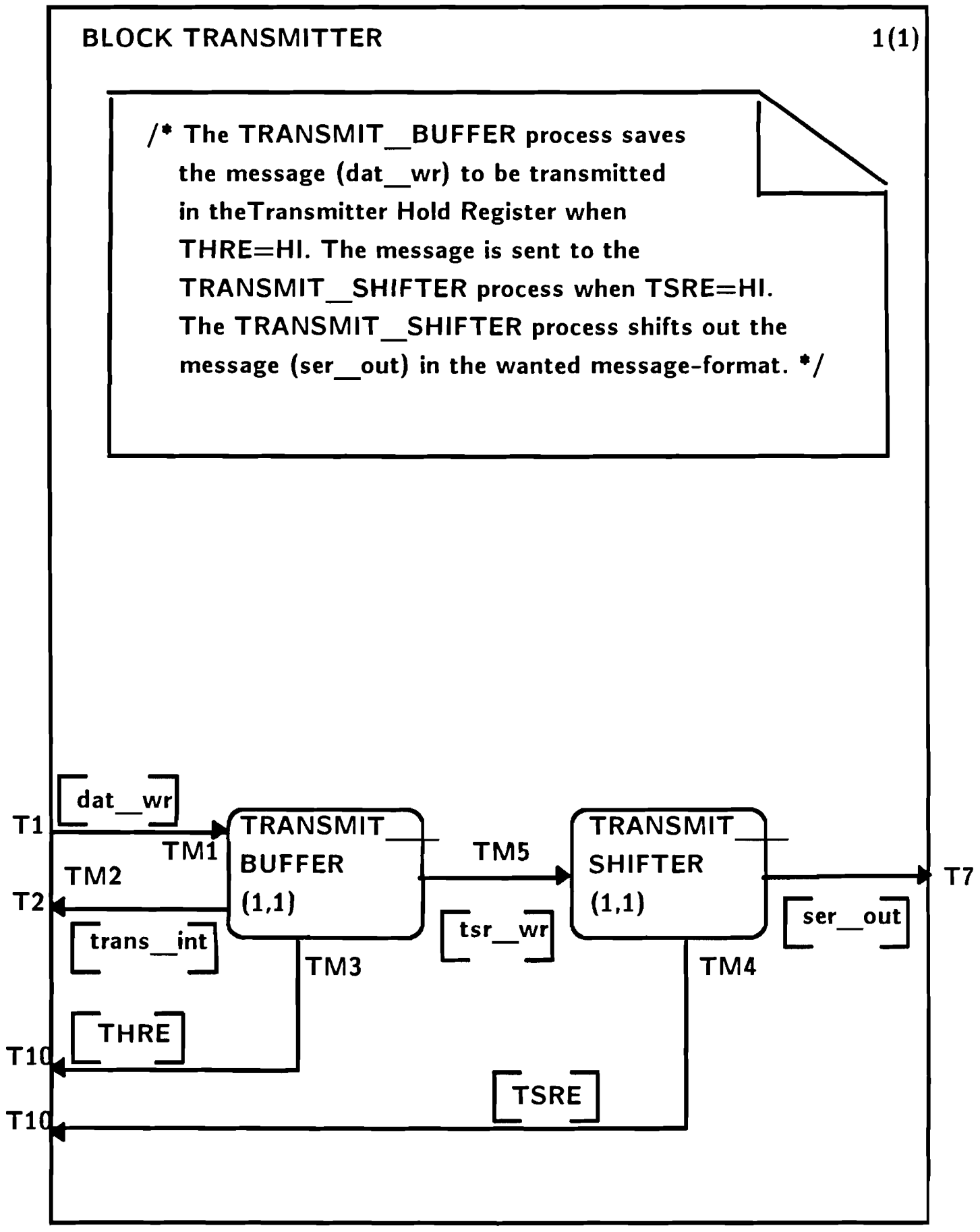


fig 27: SDL description: BLOCK TRANSMITTER

PROCESS TRANSCEIVER_CONTROL 1(2)

```
NEWTYPE lcr STRUCT WLS0,WLS1,STB,PEN,
EPS,SPAR,SETBRK,DLAB bit; ENDNEWTYPE lcr;
/* WLS0,WLS1: wordlength-selection;
   STB: stopbit-selection; PEN: parity enable;
   EPS: even parity select; SPAR: set parity;
   SETBRK: setbreak; DLAB: counter-register selection; */

NEWTYPE lsr STRUCT DRDY,OERR,PERR,FERR,BI,
THRE,TSRE,b7 bit; ENDNEWTYPE lsr;
DCL controlreg lcr, statusreg lsr;
SIGNAL lcr_wr(byte),lsr_wr(byte),rd_lcr,rd_lsr,lcr_rd(byte),
       lsr_rd(byte);
```

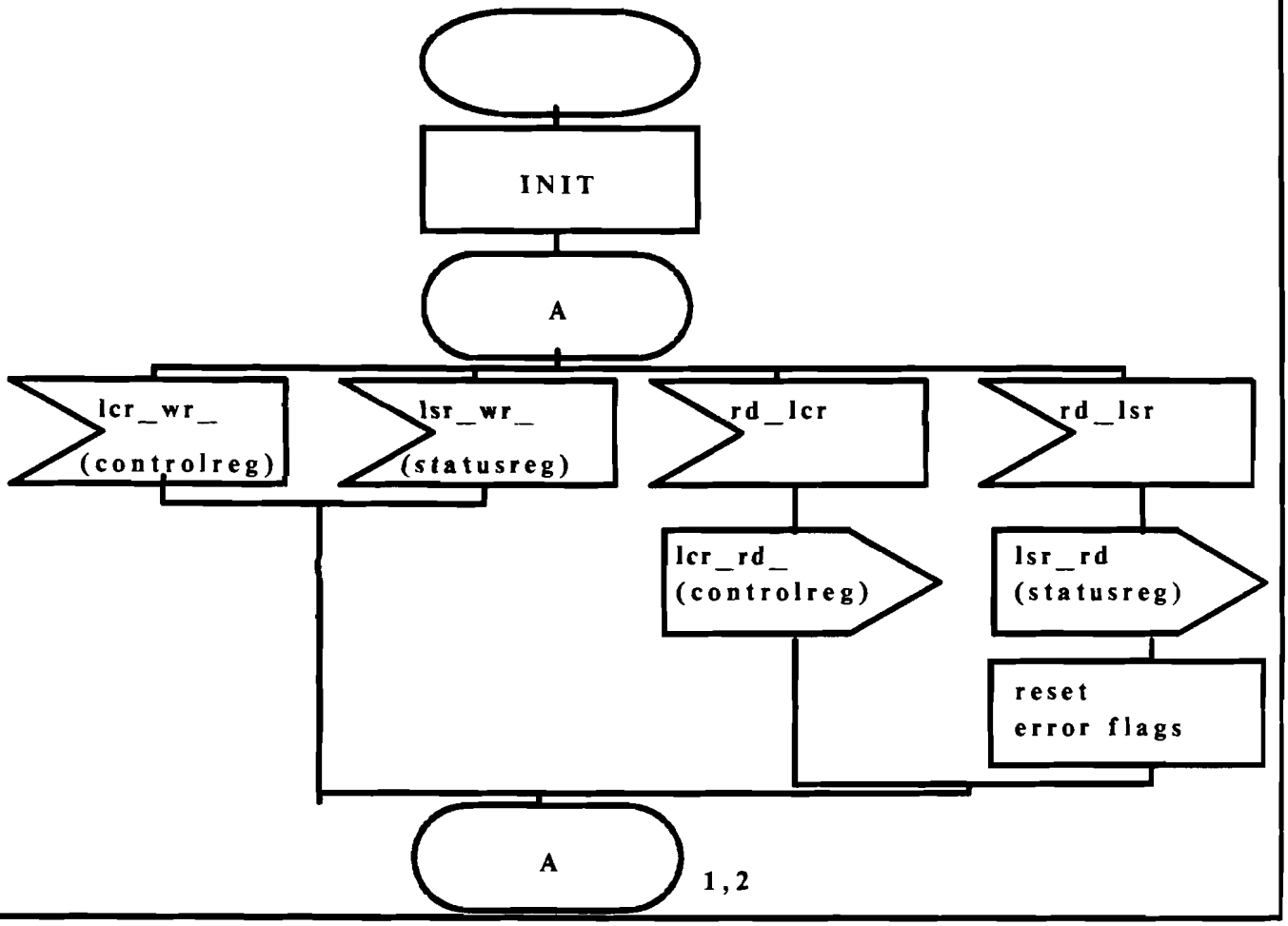


fig 28A: SDL description: PROCESS TRANSCEIVER_CONTROL(page1)

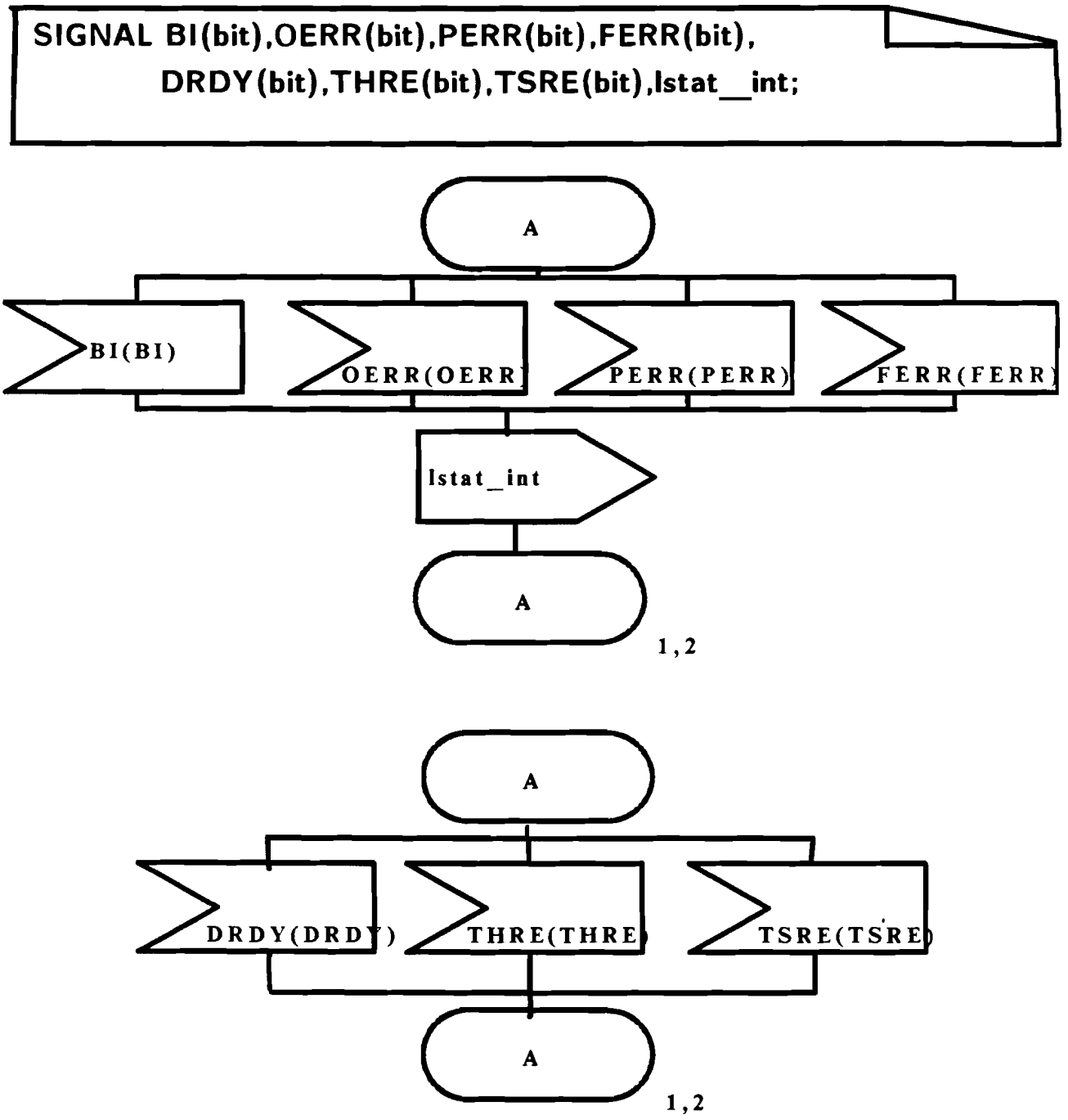


fig 28B: SDL description: PROCESS TRANSCEIVER_CONTROL(page2)

PROCESS TRANSMIT_BUFFER (fig 29)

The process graph shows two transition strings. The left transition is started upon reception of a message to be sent provided that the transmitter holding register is empty.
The right transition is started when the transmitter holding register contains a message and the transmitter shift register is empty. The transition will put the contents of the transmit buffer into the shift register.

PROCESS TRANSMIT_SHIFTER (fig 30A, 30B, 30C, 30D)

The first page describes the initializing of the process (TASK INIT) and the beginning of the shifting, including the updating of statusbits, wordlength and parity after reception of a message to be sent. The second transition is triggered by the positive edge of baudout (Note: this is not a formal description of a positive edge).

The second page first shows the generation of a startbit. The third transition is triggered on clockcount=16. Triggering on this particular clockcount value is caused by the implementation of the UART using a '16 times clock'. This means that one output bit takes 16 baudout clock ticks.

Depending on the wordlength a bit is taken from the shift register variable at a different place (tsr_reg!b<place>). After putting out the databit, the parity value is updated.

PROCESS TRANSMIT_BUFFER 1(1)

```
/* THRE and TSRE bits owned by the tranceiver__
   control process are visible */
SIGNAL THRE(bit),tsr__wr(byte),TSRE(bit),trans__int;

DCL thr__reg byte;
```

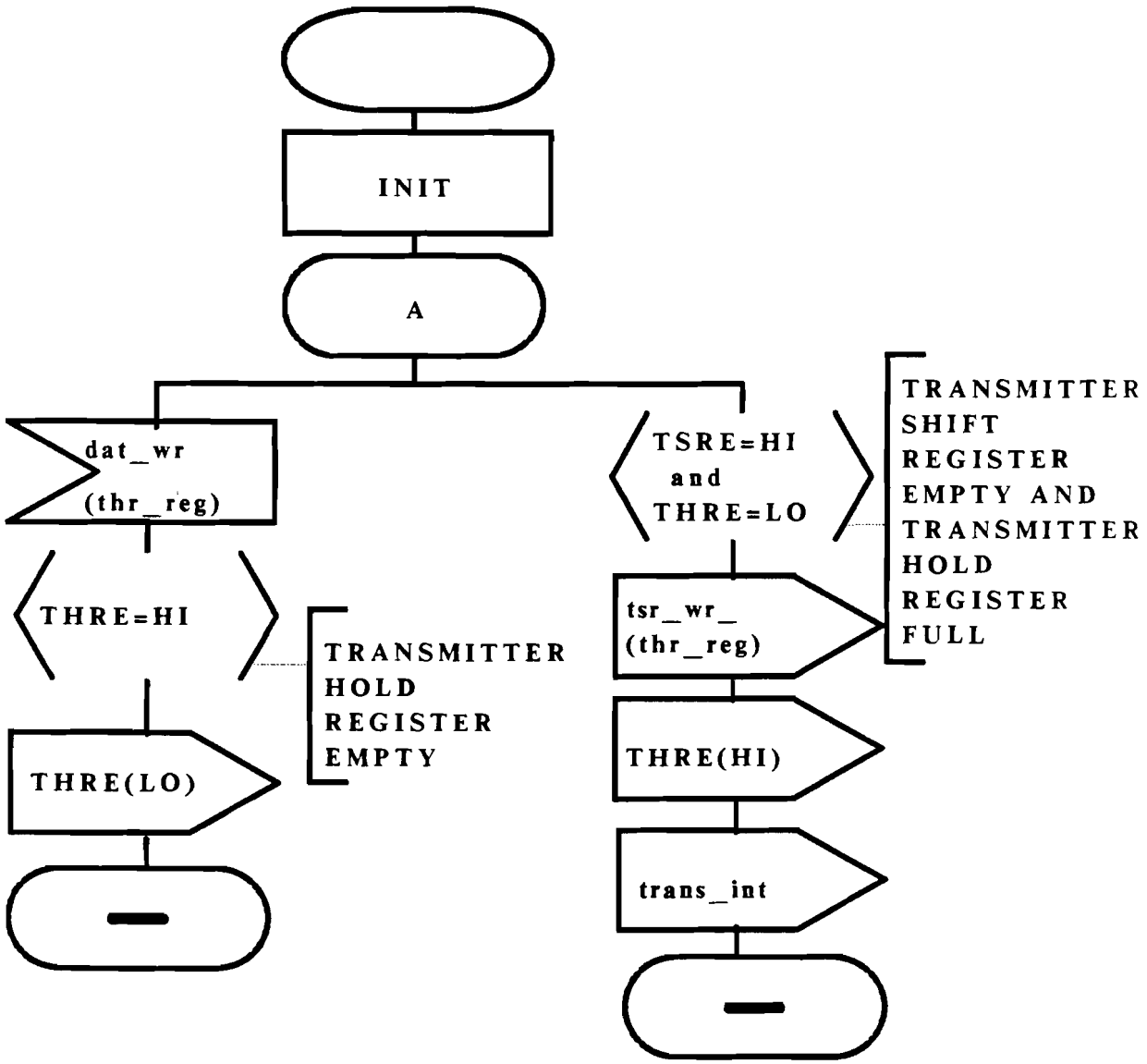


fig 29: SDL description: PROCESS TRANSMIT_BUFFER

PROCESS TRANSMIT_SHIFTER 1(4)

```
/* Variables which are visible are:
   baudout owned by BAUDRATE_GENERATOR;
   EPS,WLS1,WLS0,PEN and STB owned by the
   TRANSCEIVER_CONTROL process;
   clockcount owned by a counter_process. */
SIGNAL tsr_wr(byte),ser_out(bit),TSRE(bit);
DCL bitcount,wordlength integer, parity,sout bit;
```

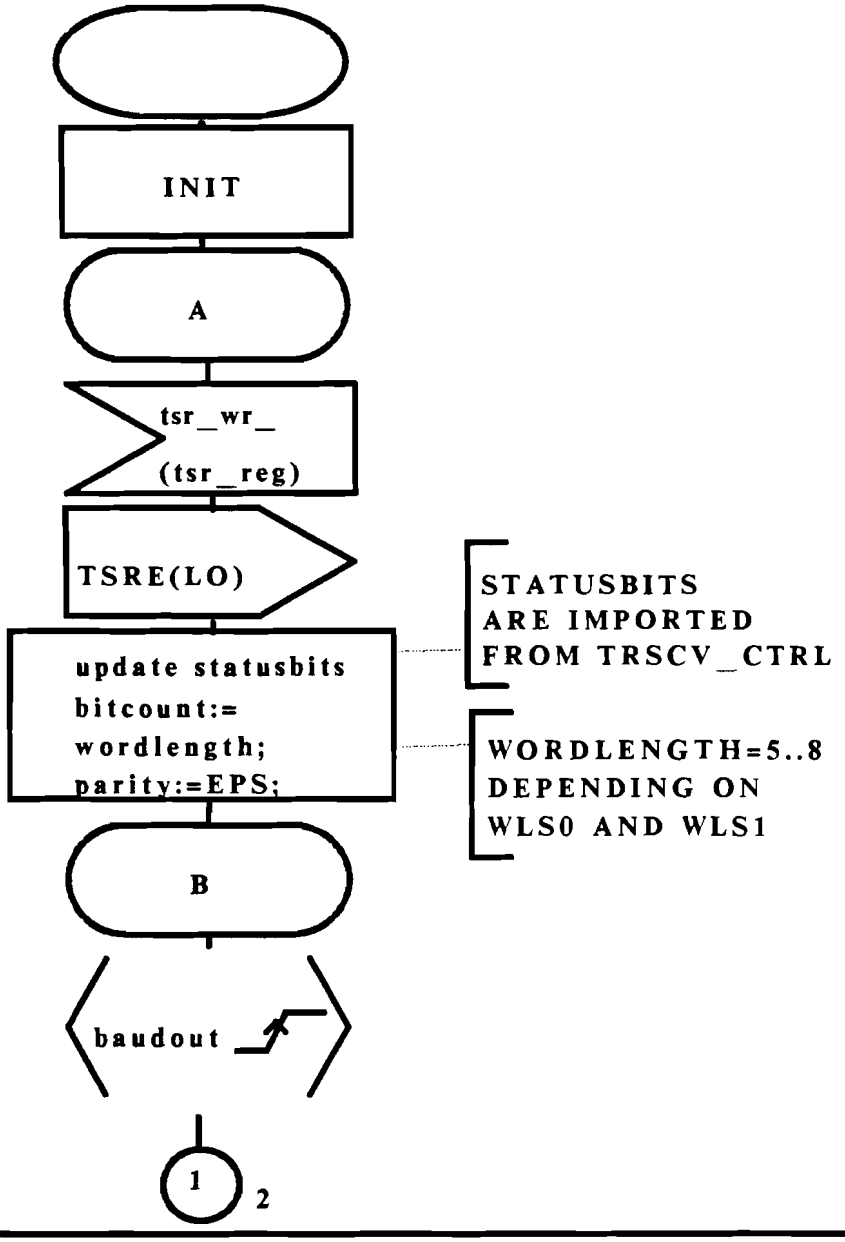


fig 30A: SDL description: PROCESS TRANSMIT_SHIFTER (page 1)

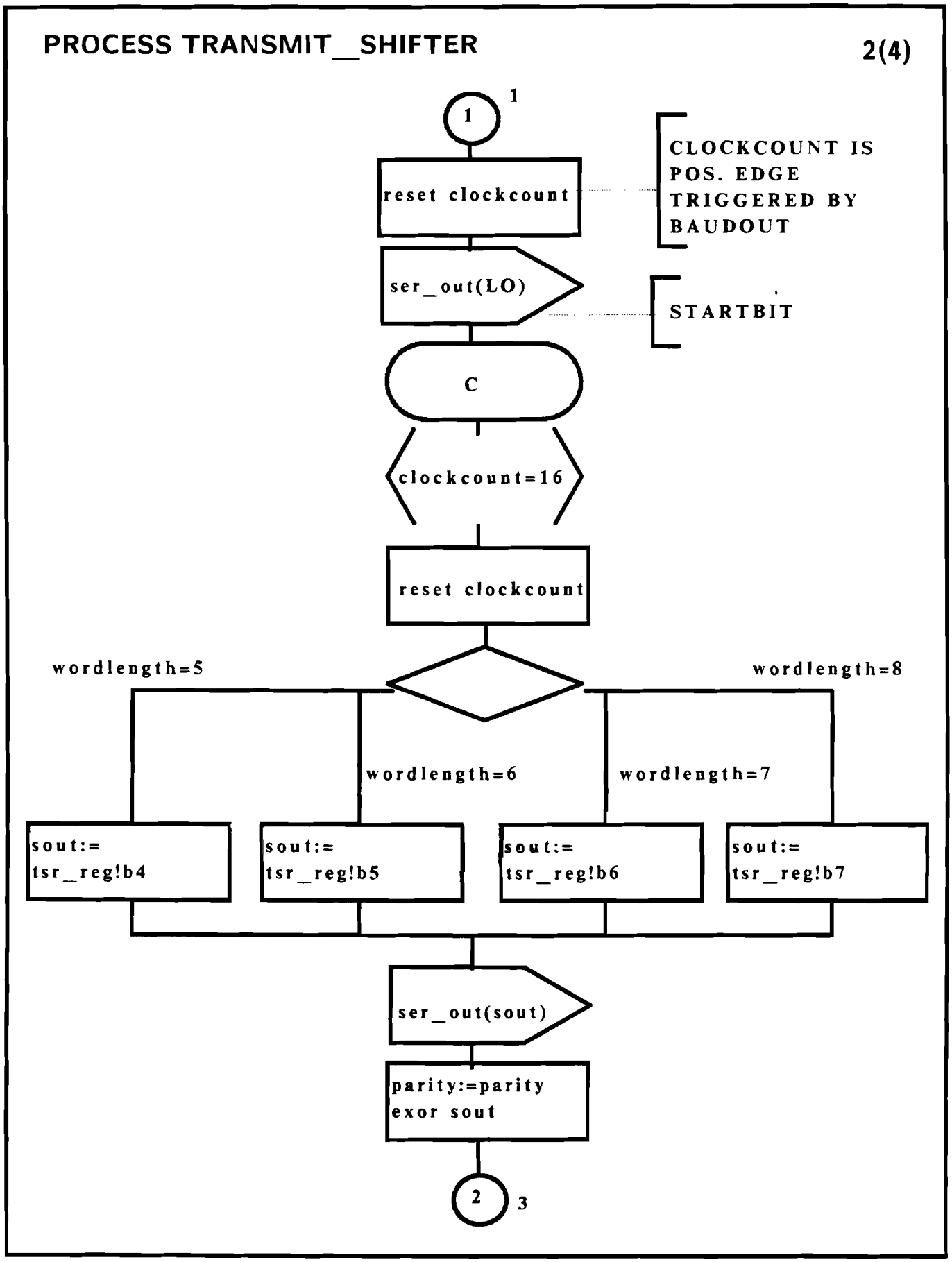


fig 30B: SDL description: PROCESS TRANSMIT_SHIFTER (page 2)

PROCESS TRANSMIT_SHIFTER (fig 30A, 30B, 30C, 30D)

The third page describes the counting of the number of data bits put out by the shift process. When this number equals the word-length, the process proceeds by putting out a stopbit or parity bit depending on the Parity Enable flag.

The last page describes the putting out of an extra stopbit depending on the stopbit flag. When wordlength=5, the extra stopbit is only 8 clockticks long (transition on clockcount=8). After putting out the last stopbit, the Shift Register Empty flag is set before returning to the startstate (A).

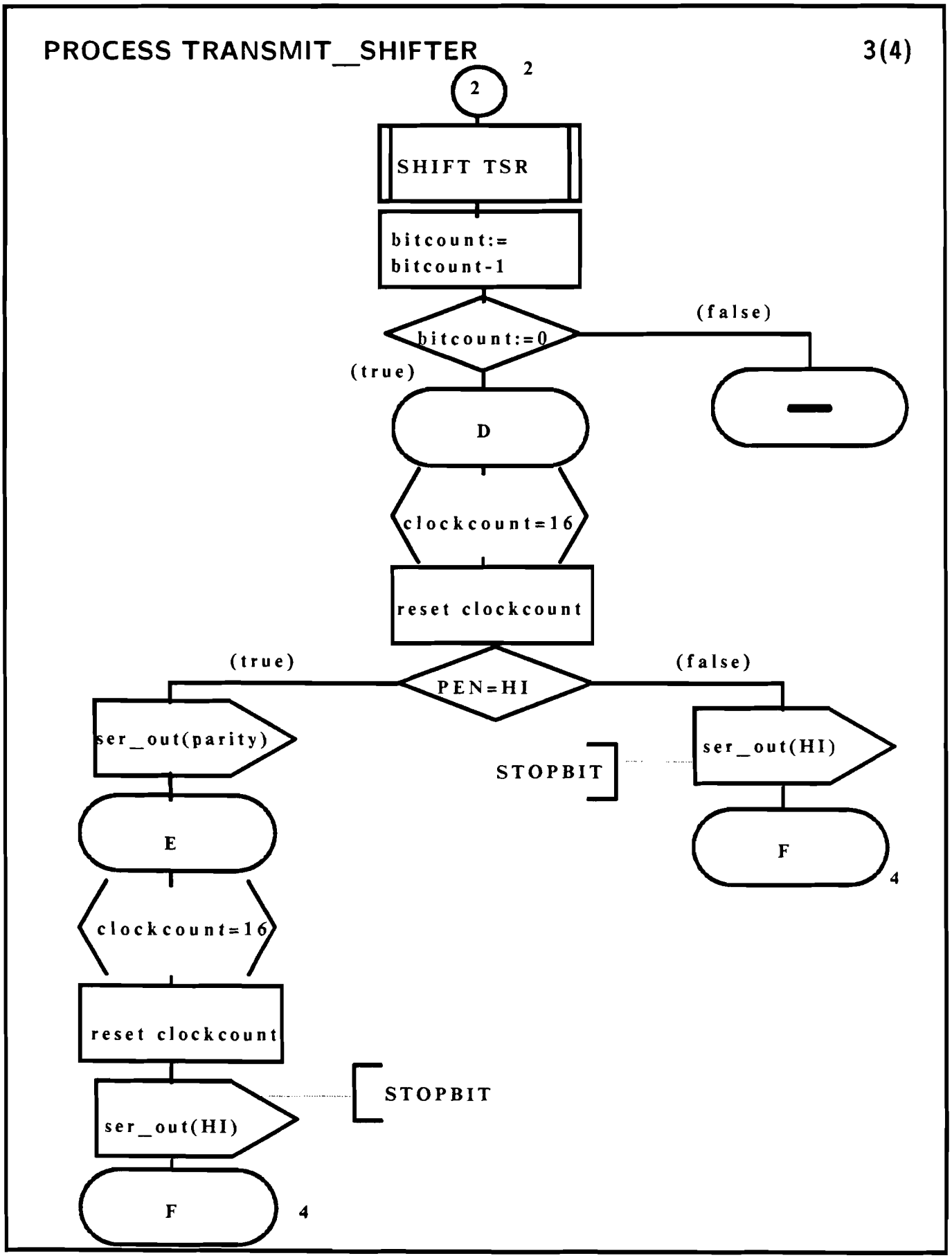


fig 30C: SDL description: PROCESS TRANSMIT_SHIFTER (page 3)

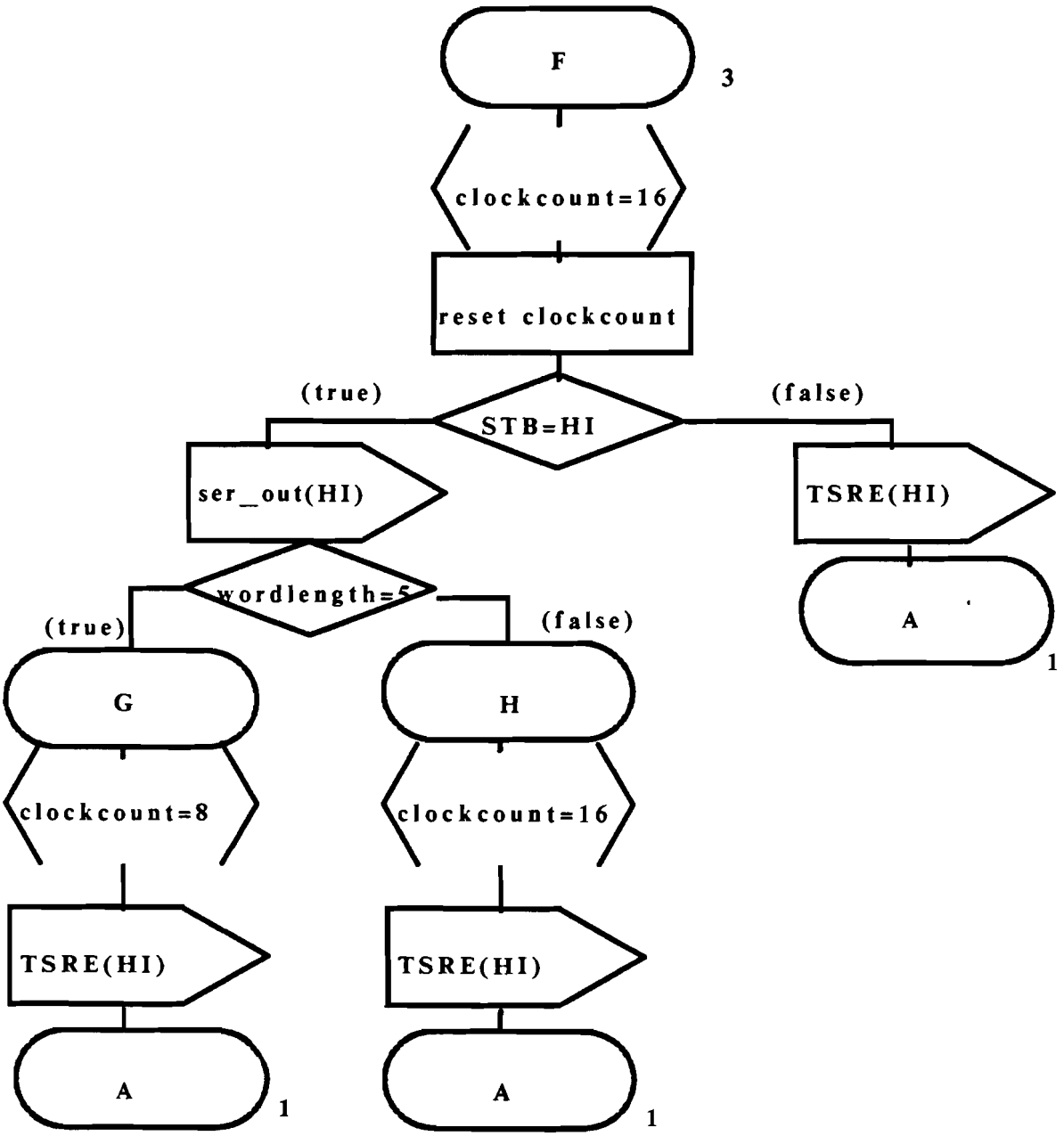


fig 30D: SDL description: PROCESS TRANSMIT_SHIFTER (page 4)

4.4. Conclusions

In this chapter it became clear that SDL can not be used as a hardware description language which can automatically be translated in a hardware implementation. But SDL must be seen as a behaviour description of which some parts must be evaluated before an implementation form is chosen.

Without notice we used another approach to describe state machines. That is, we introduced the concept **communicating** extended finite state machines to define state machines as processes and their interactions as communication. So we did not describe state machines as one big state machine defining all combinations of states, but actually synchronized the different state machines by their communication.

It became clear that in some way we have to avoid the implicit process queue use to get an economic use of hardware. Therefore we introduced synchronous communication upon the asynchronous communication model. However, we showed that sometimes the queue model must be used to get a clear behaviour description.

To avoid the loss in performance due to long wait periods at synchronous communication we introduced separate buffer processes. Especially the communication between processes has to be evaluated before an implementation form is chosen, to see if separate buffer processes are needed.

The message structure of SDL and the level driven nature of digital hardware differ in the way the communication path is structured. For instance, splitting of a channel in a message driven model does not mean splitting the message in two same messages as in a physical channel, like a signalline or bus. Therefore the decisions for the implementation of SDL-channels are postponed to a later design stage.

In some cases it is possible to react on more than one signal at a time in hardware in contrary to the restriction of one signal per transition in SDL. To specify these multiple input actions we need some kind of exception handling description not provided by SDL.

The implementation of the actual processes described in SDL is chosen to take place via splitting the process actions in a control and information processing part. The same functional splitting will take place on SDL signals. Implementation of process TASKs are assumed to take place automatically. That is, we will not bother about the implementation of the state machine part of the SDL process.

5. SDL design methodology towards hardware

In this chapter a guide is given on how to use SDL in the design process of VLSI circuits containing several hardware implemented parallel processes.
SDL is used in the specification phase of the design process for setting up a behaviour model.
In [VLSI.2] the specification phase is split in:

- Requirement elaboration
- Requirement specification
- System attributes
- Process definition
- Verification

During requirement elaboration a set of requirements is defined after thorough investigation and evaluation of the user needs.

The requirement specification can be split in:

- System requirements
- Functional requirements
- Performance requirements

The System requirements are those requirements imposed by the environment upon the design. These requirements are given as a set constraints which limits the final shape of the design. Some constraints are: maximum power dissipation, maximum input/output voltage, maximum signal rate, etcetera..

The functional requirements are a description of the systems input-/output relations. At this point SDL will be introduced to formalize the description of these input/output relations.

Performance requirements define the (minimum) ratings required for the dynamic behaviour of the system. Some performance requirements are: input rate, output rate, response time, etcetera.

System attributes have their influence on the overall design strategy. Some system attributes are: cost, reliability, availability, flexibility, etcetera.

The process definition gives the definition of system processes, required resources by the processes and the interaction between processes. At this point a decomposition into functional blocks takes place. This decomposition is based on the knowledge of the designer and the problem definition. The functional blocks not necessarily have their counterpart in hardware that is the functional decomposition at specification level does not force the same decomposition in hardware.
The process definition will be given in SDL. Decomposition is described by SDL using decomposition of the system into blocks, blocks into substructures, blocks into processes and processes into services.

Verification is done by simulation and evaluation of the formal specification to see if the specification satisfies the user requirements.

The SDL behaviour description together with system requirements, performance requirements and system attributes must be rewritten in a hardware oriented behaviour description when entering the next phase, the design phase (see fig 31 on the next page).
Note: We state that, when implementation of the hardware oriented behaviour description can be automated completely, that this description is the beginning of the implementation phase.

The rewriting of the SDL behaviour description can be done using an expert system with a knowledge base. This knowledge base must contain implementation facts regarding different communication forms between processes. The choice which communication form is taken, depends on performance requirements, communication rate and other related facts. These facts must be extracted from the SDL description using tools like CCS.

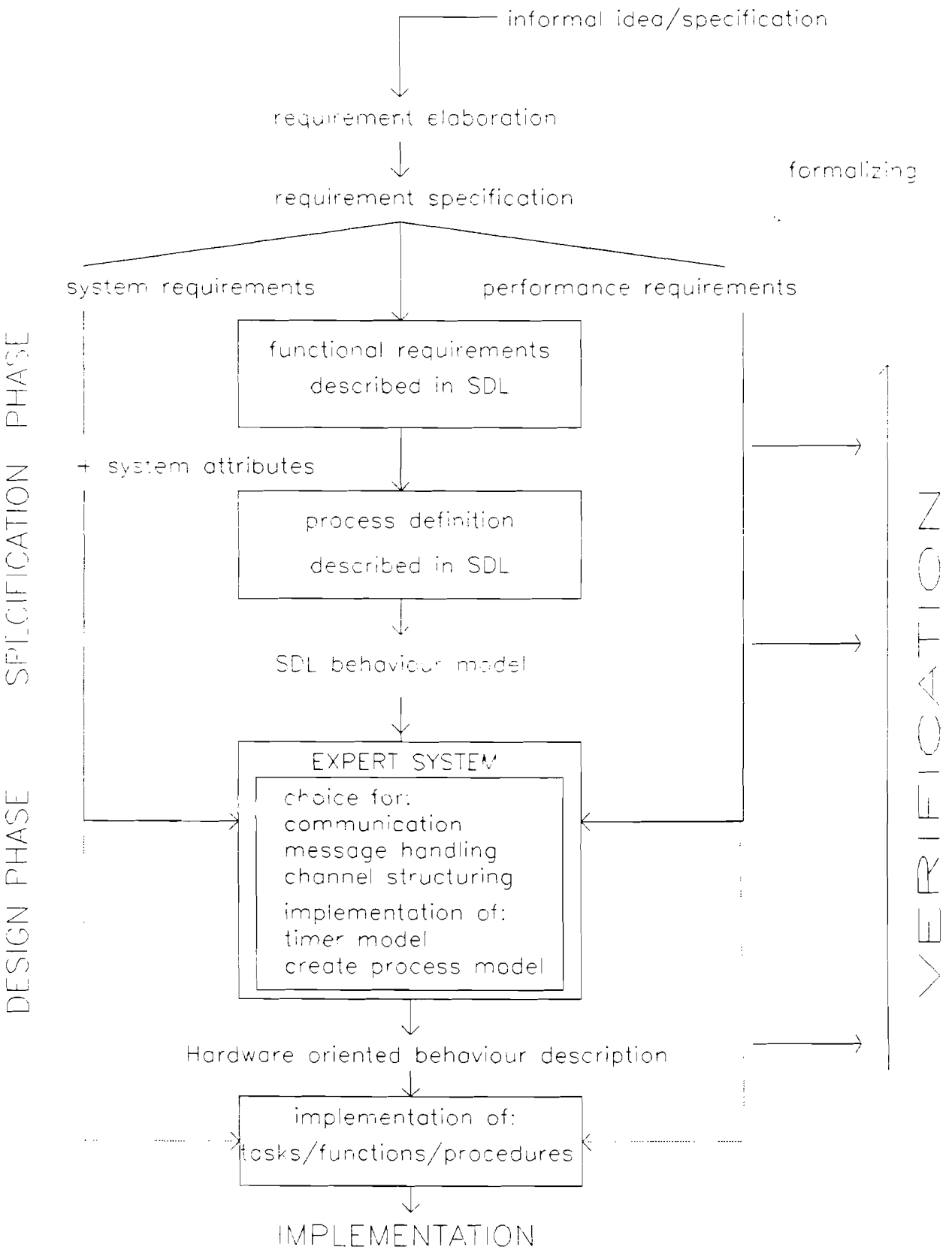


fig 31: the use of SDL in the specification phase of VLSI design

5.1. Setting up a behaviour model

Defining a SDL design methodology towards hardware means formalization of the specification phase in the design process. Complete formalization is not possible nor wanted because design partially is/has to be a creative activity.

In "The science of programming" [Gries], about formalizing the reasoning associated with system design is said:

> *"be assured that complete attention to formalism is neither necessary nor desirable. Formality alone is inadequate, because it leads to incomprehensible detail; common sense and intuition alone are inadequate, because they allow too many errors and bad designs"*

In practice it is very difficult to give a specification without use of design knowledge of later design phases. For the same reason it is difficult to give a standard method for setting up a behaviour model.
But when we restrict ourselves to the constructs allowed in SDL, it is possible to define some way to decompose the problem without use of (much) implementation knowledge.

In the next paragraphs some basic rules are given for setting up a behaviour description in SDL. First we give some rules for functional decomposition which takes place in the first two levels of the SDL description, the SYSTEM and BLOCK levels.
Second, some rules are given to set up the actual behaviour description, the PROCESS definition.

5.1.1. Functional decomposition (SYSTEM and BLOCK level)

SYSTEM LEVEL

* The SYSTEM definition shows the signals between the environment and the **single** system block.
* The signals are grouped by functionality using signallists.
* For functional grouping of signals it might be useful to define message sequences (visualizing causal relations between input and output signals) and to give informal descriptions of actions upon those signals.
* Use clear naming of signallists and separate signals.
* Build signallists of several signallists to show functional splitting at lower levels (that is, splitting of channels and blocks).
* Give a signallist **definition** just before it is split in signals or signallists at channel or signalroute splitting.
* Channels can also be used to visualize differences in functionality of carried signals, or can be used to visualize different interface points (in the UART description different channels were used for the CPU-interface and the serial-interface). The latter reason for channel splitting must be avoided when this leads to too many channels causing unclear specifications (and messy drawings).
* Define basic signals like clock and master-reset as global variables at system level informally. Formalization of this kind of signals by implementing extra input processes and export mechanisms can be automated when a complete description for simulation is needed.

  Note:

  In general some specific hardware related signal definitions best can be done informally regarding the SDL syntax, but formally regarding predefined specific constructs with their translations to the SDL syntax in an expert system. Then the translation to a complete formalized SDL description is possible anytime we need.

* Data type definitions like bit, byte and register only have to be defined once using the NEWTYPE construct, in order that we can use these types as any other standard data types.

BLOCK LEVEL

* At system level functional decomposition was prepared by the definition of signallists and especially those signallist definitions built out of several signallists connected to channels which can be split at block level.
* Functional decomposition is a tool to split a problem in manageable parts. Although not necessary, the decomposition mostly leads to the same decomposition in hardware. The latter must be avoided in the specification phase (the behaviour description).
* Decomposition of the system in several blocks (block substructuring) implies that the functions contained in different blocks are highly independent and may act concurrently. One of the critera for functional decomposition is that the communication rate between blocks must be low and most communication actions must have a data transfer function, not a mere synchronization (control) function.
* Block substructuring must be used to emphasize concurrency of functionality, not of actions. The latter will be done during a later stage when a block definition is split in process definitions.
* Use clear naming of blocks/processes. If possible give an informal description of the blocks/process' functionality.
* Do not split channels between blocks/processes in functionality, allow a maximum of one channel between every block/process pair to keep the description clear.
* Sharing of one process by other processes in the same block can be visualized by the create construct. Shared processes usually will contain specific functions which are used by several hardware implemented processes.

5.1.2. Process definition (PROCESS level)

PROCESS LEVEL

* The process definition is a sequential decomposition of the problem.
* In the model, states are synchronization points on input signals, tests on variable conditions, or combination of inputs and variable condition tests. The latter combination must be used carefully because the SDL model only starts a transition on a True condition when the input queue is empty.
* States must be used as a framework for sequential decomposition. That is, we must keep in mind that the state machine model is a tool for decomposition, not for state assignment.
* Service decomposition must be used carefully because at first sight it looks the same as process decomposition. But because the service model is restricted to one single service transition at a time, services can not act concurrently.

  In most cases, use of procedure decomposition is better than service decomposition.
* Use procedure definitions to group a set of related actions to one (virtual) action.
* Procedures are useful for a clear specification and for sets of actions which return at different points in the process description.

  Note:

  Possible implementation of a procedure is a micro subroutine in a micro programmed controller.
* Use procedures for specific hardware related functions like register-shifting, initializing, etcetera. These specific procedures must be defined as an extension of standard SDL (procedure library).
* Use informal task descriptions only to postpone formalization of certain tasks. Do not use them for large transition sequences. Use clear naming of these tasks, and connect comments to them with description of the effects upon variables.

* Formal task descriptions contain assignments within the scope of the SDL data types and related operators or within a NEWTYPE definition.
* Define common variables as revealed by the process owning the variables. The choice which process owns a particular variable is based on functional decomposition of the processes in the same block.

  Note:

  Only the owning process can change the value of a revealed variable.

  Change of the value of a variable by a process not owning the variable is modelled by write signals containing the new value sent to the owning process.

  The owning process updates the value of a variable at receiving a corresponding write signal.
* Write actions to a memory like environment do not need a write (request) signal but use the synchronization inherent to the (data) signal. Read actions do need a read request before data is put out by the process.

5.2. Evaluation of the behaviour model

At this point we state that we have a complete SDL description, all informal parts are formalized by hand or automatically giving a syntacticly correct description.
This description has to be evaluated in two different directions.

First we look if the described behaviour is the same as the behaviour we want (verification). When the SDL description is complete, verification can be automated. That is, we can use tools to evaluate the described behaviour and systems performance.

Second we must evaluate the description towards hardware implementation (refinement). Here the evaluation results are used to decide what communication form between processes is chosen.

5.2.1. Verification of the behaviour model

Not only the functional requirements and process definition must be checked but also the performance requirements.

The system description must be checked on:

a causal behaviour of signals

b causal behaviour of process' states

c needed queue length

d utilization of processes

f deadlocks

a: A systems behaviour partly is defined by its message sequences. That is, one way to check this behaviour is to determine the causality relations between signals.

For a single process description it is obvious which input signal preceeds a certain output signal. But for several processes communicating within a block the causality relations between input and output signals of the block are not described.

The causality relation of input/output signals at the block interface can be determined by parallel composition of the processes in the block.

We could use the expansion algorithm [Milner] with restriction of the internal signals of the block (after translating SDL to CCS). In this case we must use the expansion algorithm with value passing because the causal bahaviour can be influenced by variable values (we are not interested in synchronization only).

Another way to determine the causality relation between input/output signals at the block interface is simulation. That is, we define possible input sequences and evaluate the corresponding ouput sequences found by simulation. The latter is dangerous because the designer may forget some of the possible input sequences.

We can use the same approach to the system description to evaluate the overall causal behaviour at the environmental interface.

b: Closely related to causality relations between input and output signals are the causal relations between states of different processes.
These relations are important to see how communicating processes are synchronized and to check their behaviour.
Later we will see that the way processes are synchronized, influences the choice how communication is implemented.

c: When we simulate the behaviour description, it's useful to monitor queue lengths to check for specification errors, performance and choice of communication method.
it is obvious that when some queue length keeps growing during simulation, the specification is wrong (deadlock). Likewise when a queue is large, the specification might be wrong or at least not optimal in performance.
When certain signals are in the queue more than once, but are expected by the process in only one state, unbuffered synchronous communication might be bad or even impossible.
The dynamic behaviour of the queue length of different processes is a measure for the performance of the system. For instance, if a certain process queue grows to some maximum before signals are taken from it, then the process might have a negative influence on the overall performance (bottleneck).

d: Evaluation on the utilization of processes might lead to splitting of busy processes in smaller ones to get a higher performance. A measure for utilization of a process is the ratio of active transitions and wait states per clock tick.

e: Deadlocks mostly are caused by a wrong specification.
Because the SDL model has no restrictions on the queue length, "communication" deadlocks can only occur when two processes are waiting for each other's input signal causing the rest of the system to halt.
Another cause for deadlock is when one process waits on a true condition of a variable owned by another process while

the rest of the system is halted.
Unsuccesful process creation might also lead to a deadlock.

Checking possible deadlocks can be done by extensive simulation, checking the queue length during simulation or parallel composition of the processes (CCS).

5.2.2. Refinement of the behaviour model

The other direction in which the behaviour model must be evaluated, is towards a hardware oriented behaviour description. In this description the communication- form and path between every process is chosen and specific SDL constructs like process creation and the SDL timer are rewritten into a hardware model.

The choice which communication form is chosen depends on:
a synchronization of process' states of different processes
b needed queue length
c communication rate
d performance requirements

a: When different processes are tightly coupled, in most cases introduction of synchronous communication hardly affects the performance of all, because wait times for succesful communication will be small.
When different processes are loosely coupled, in most cases choice for buffered communication will give a better performance.
When an input action is a causal action of the preceeding transition (the process is expecting a particular input and nothing else), then there is no need for an input queue nor a communication protocol.

b: When monitoring the needed queue length at simulation, not only the length is important but also the contents of the queue

and the dynamic behaviour of the queue length.
It is possible that introduction of synchronous communication without buffering can cause deadlock in case that the needed queue length is always larger than one.
When the queue contains several identical messages from the same source, synchronous communication without buffering causes loss in performance of the source process or even deadlock.

c: When the communication rate between processes is high, the implementation of a separate input/output process gives a better performance because it relaxes synchronization constraints upon processes.

d: Introduction of synchronous communication in general will affect the performance of the system but saves hardware. When the performance requirements are high, we must choose for buffered communication, thus extra hardware.

When several processes must use the same communication path (a single signal bus), an arbitration algorithm/process must decide which process uses the path. The implementation form of this arbitration depends on the same points as the choice for the communication form.

In general the implementation form of communication path together with an arbitration algorithm must be chosen to get the best performance given the communication rate, process synchronization and communication form.

5.3. Rewriting the bahaviour model

In chapter 4 implementation models were given of communication, message handling, the SDL timer and process creation. These models were given in their general forms.

At rewriting the behaviour model, the exact implementation of these models depends on the evaluation facts mentioned in preceding paragraph.
This rewriting will be done in an interactive way given the evaluation facts, a knowledge base with implementation models and the knowledge of the designer.

Some choices for implementation still have to be made by the designer when the evaluation of the behaviour model gives several solutions or no solution at all. Still the expert system can help the designer by giving particular (estimated) effects of implementation choices or even by restricting the possible set of solutions.

The expert system must avoid "the re-inventing of the wheel".

In the final form of the rewritten behaviour model, the hardware oriented behaviour description, **all** *interprocess* communication is made explicit.
This description has to be verified against the behaviour model provided that the behaviour model is verified against user requirements. Verification against the behaviour model is not enough because not all user requirements are contained in the behaviour model.

Not only introduction of synchronous communication but also the introduction of clock synchronization and time consumption of tasks can affect the specified behaviour.

The hardware oriented behaviour description still has tasks, decisions and procedures defined as no time-consuming actions. The states in this description are synchronized to a system clock. In this case a more realistic behaviour description would be, a description with estimated time consumption of tasks expressed in the amount of clockticks needed by a task.

For simple expressions estimation of the time needed can be done automatically. The time consumed by complex expressions is dependent on the implementation. So in the latter case the consumed time must be given by the designer (if necessary with support from by the expert system).

6. Conclusions and suggestions

We need a formal description technique for specification of complex digital systems in VLSI design. Especially the design of parallel processes in hardware needs a description technique that copes with concurrency in a clear way.
The description technique must be formal to allow only one interpretation of the described behaviour
The technique must have a strong mathematical base to make automated specification, simulation and verification possible. But the mathematical base must not make the specification technique user unfriendly or difficult to read.
It must be possible to describe the behaviour in an abstract way without knowing implementation facts.

SDL is a good compromise for our purpose, the description of parallel processes in hardware. SDL has the advantage of two representation forms, a graphical and textual representation. The graphical representation gives a better human interface and the textual representation can be used as input for simulation-, verification- and evaluation- programs.

SDL is based on the extended finite state machine model providing an easy to use structure for decomposition of the description and a first step towards the implementation in hardware, in the form of state machines.

The asynchronous communication model between SDL processes using one common input buffer per process, makes specification of inter process communication easy. But the implementation of inter process communication must be synchronous to avoid errors due to races and to get an economic use of hardware.

Channels and signalroutes in the specification are implemented in hardware by busses, signallines and bus-arbiters.

The actual processes are implemented by splitting them in a control part and an information processing part and implementing these parts in a control unit and an operational unit respectively. The same functional splitting must be done on signals, where the data part is handled by the operational unit and the control/synchronization part is handled by the control unit.

SDL can not be used as a hardware description language, but must be used to set up a behaviour model of a system to be designed. The first two levels, SYSTEM and BLOCK level, are used for functional decomposition of the problem. At PROCESS level, the sequential decomposition is given.

Setting up the behaviour model in SDL must be supported by tools for setting up a SDL description, a SDL syntax checker and a library containing particular hardware constructs described in SDL.

When the behaviour model is proven to be correct by verification, the next step must be the translation to a hardware oriented behaviour description done by the designer in interaction with an expert system.

The expert system must help the designer in deciding which communication form a particular process uses, how channels are structured and how specific SDL constructs like TIMER and CREATE are implemented. These decisions must be based on evaluation facts upon synchronization of processes, needed queue length, communication rate and performance requirements.

LITERATURE

CCITT: SPECIFICATION AND DESCRIPTION LANGUAGE (SDL),
CCITT/Rec. Z.100, January 1987.

Copper: REAL-TIME DISTRIBUTED CONTROL WITH ASYNCHRONOUS MESSAGE RECEPTION,
Albert N. Copper III, John P. Kearns
Proc. of the Real-Time Systems Symposium,
(Cat No.85CH2220-2), San Diego, 1985.

Dijkstra: "COOPERATING SEQUENTIAL PROCESSES",
Programming Languages, F. Genuys ed.
pp. 43-112, Academic Press, New York, 1968.

Gries: THE SCIENCE OF COMPUTER PROGRAMMING,
D. Gries
Springer, New York, 1984.

Harel: STATECHARTS: A VISUAL APPROACH TO COMPLEX SYSTEMS,
David Harel
Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel, March 1986.

HHDL: HIERARCHICAL HARDWARE DESCRIPTION LANGUAGE,
SILVAR LISCO
condensed manual given in VLSI.1

ISO ESTELLE: A FORMAL DESCRIPTION TECHNIQUE BASED ON AN EXTENDED STATE TRANSITION MODEL,
ISO/TC97/SC21(DP 8806), Februari 1985.

ISO LOTOS: A FORMAL DESCRIPTION TECHNIQUE BASED ON THE TEMPORAL ORDERING OF OBSERVATIONAL BEHAVIOUR,
ISO/TC97/SC21(DP 8807), Febructari 1985.

Klemann: IMPLEMENTATION OF ALGORITHMS INTO HARDWARE,
O.M.R. Klemann
Eindhoven University of Technology, Department of Electrical Engineering, August 1988.

Koomen: CCS AS A POSSIBLE SEMANTIC BACKGROUND FOR SDL,
Cees J. Koomen
Technical note nr. 172/82
Philips Research Laboratics, 1982.

Milner: A CALCULUS FOR COMMUNICATING SYSTEMS,
Robert Milner
Springer, New York, 1980.

Schie: UART IN TOP DOWN DESIGN,
A.L.D. v. Schie,
Eindhoven University of Technology, Department of Electrical Engineering, 1987.

VLSI.1: STRUCTURED VLSI DESIGN COURSE, WEEK 1
Eindhoven University of Technology, Department of Electrical Engineering, 1987.

VLSI.2: STRUCTURED VLSI DESIGN COURSE, WEEK 2
Eindhoven University of Technology, Department of Electrical Engineering, 1987.

SDL '87: SDL '87: STATE OF THE ART AND FUTURE TRENDS
R. Saracco and P.A.J. Tilanus
Elsevier Science Publishers, North Holland, 1987.

APPENDIX A:
LIST OF ABBREVIATIONS USED IN THE UART DESCRIPTION

cont_in	control in
dostr	data output strobe
distr	data input strobe
cs	chip select
abus	address bus
ads	address strobe
cont_out	control out
csout	chip select out
ddis	driver disable
mod_in	modem in
ncts	not clear to send
ndsr	not data set ready
nrlsd	not received line signal detect
nri	not ring indicator
mod_out	modem out
nrts	not request to send
ndtr	not data terminal ready
nout0	not out 0 (programmable output 0)
nout1	not out 1 (programmable output 1)
dat_in(byte)	data in (type byte)
dat_out(byte)	data out (type byte)
ser_in(bit)	serial in (type bit)
ser_out(bit)	serial out (type bit)
intrpt	interrupt
osc	oscillator
rclk	receive clock
baudout	baudrate generator out
trscvreg_wr	transceive register write
trscvreg_rd	transceive register read
controlreg_wr	control register write
controlreg_rd	control register read
baudreg_wr	baudrate generator register write
baudreg_rd	baudrate generator register read
trscv_int	transceiver interrupt
trans_int	transmitter interrupt
recv_int	receiver interrupt
lstat_int	line status interrupt