

# A Unified Concept for Blocks and Processes in SDL

Eckhardt Holz

Humboldt-Universität zu Berlin, Institut für Informatik  
Rudower Chausse 5, D-12489 Berlin  
holz@informatik.hu-berlin.de

**Abstract.** The authors develop an unified concept for blocks and processes in SDL. The current principles for a structural decomposition of a system into a set of nested block definitions have been applied to SDL processes. This enables a decomposition of processes into contained processes and opens also the possibility of a dynamic creation of such complex structures. From a behavioural point of view this structural decomposition can be realized by a concurrent execution or by an interleaved (alternating) execution of the contained processes. The dynamic semantics of the proposed concept is explained by transformation rules, mapping the unified block/process concept onto basic SDL.

## 1 Introduction

Currently SDL [1] provides different concepts for the structural and behavioural decomposition of a system specification. The concept of blocks is used to specify the potential static structure of the system. Blocks are containers for concurrent behaviour elements (processes) or are again decomposed into blocks. A process on the other side may be just a single state machine or may again be decomposed. However, process decomposition is a one-step pure behavioural decomposition. The behaviour elements are alternating behaviour components (services), which live and die along with their containing process. Related to these concepts is also the ability for a dynamic creation of elements. Whereas blocks are static elements, processes within blocks may be created dynamically and may also cease to exist. Services of a process, however, may not be created dynamically. They are created when the process is created. When all services of a process have stopped (and ceased to exist), its containing process also ceases to exist.

In contrast to that many users request SDL to provide dynamic block creation and a more generalised decomposition concept. Associated with decomposition, but also as a separate requirement, is the support for shared data between behaviour elements in a container. Further demands in this direction is rooted in the future alignment between the Unified Modelling Language (UML, [2]) and SDL [6]. UML does prescribe neither a tight relation between structural decomposition and behavioural decomposition (concurrent or interleaved), nor between the composition kind and the ability for dynamic creation.

The proposal in this paper addresses these topics. It proposes an extended process concept for SDL, covering properties of the current blocks as well as properties of the current services. To keep compatibility to previous versions of SDL and to allow a step-wise migration, these additions are not introduced as a replacement for the current con-

cepts, but as an add-on to the process. The semantics of the new concept is defined on basis of a transformation to basic SDL.

Although applied to SDL, the paper specifies a semantic model for shared data between processes that may be applicable to other languages and notation techniques (e.g. UML) as well.

The next sections give first a more informal explanation of the new concepts and shows the syntactic notation. This is followed by some smaller application examples. Afterwards the underlying semantics is explained by means of the transformation rules. Finally, an outlook is given, how these new concepts integrate with other new concepts in SDL-2000.

## 2 Block and Process Decomposition

The general idea is to include all concepts of structural and behavioural decomposition in the concept of processes and to de-couple the two kinds of decomposition from each other and from the dynamic generation of instances. The decomposition principles apply for type definitions as well as for concrete instance set definitions. A further decomposition kind which has been applied throughout the paper, is the notion of a state machine in terms of composite states as part of processes and mixed with the contained process sets. A detailed description of composite states is given in [5].

### 2.1 Structural and Concurrent Decomposition

Structural decomposition is obtained by a decomposition of processes. To show this new feature such processes are called *block process*, what is also expressed by the new symbol for *block processes*. A *block process* is defined as any other process, that is it

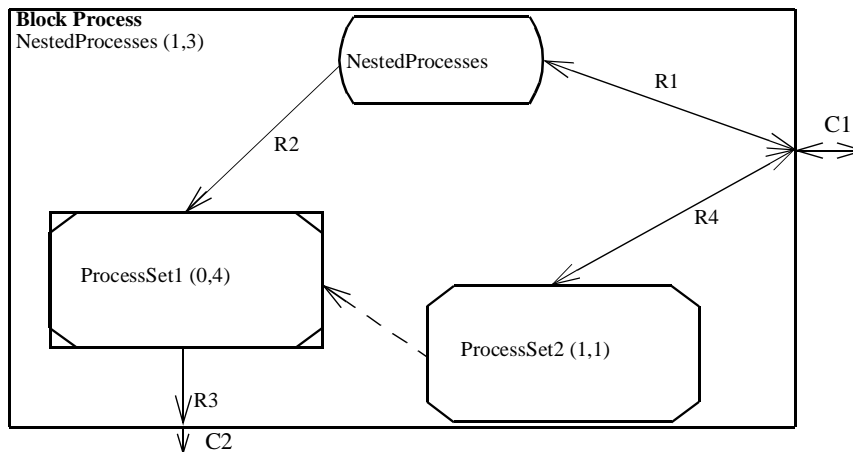


Fig. 1. Structural Decomposition of processes.

may have an associated state machine, local variables, parameters etc. However, in addition such a *block process* may also contain process instance sets, as its is known from

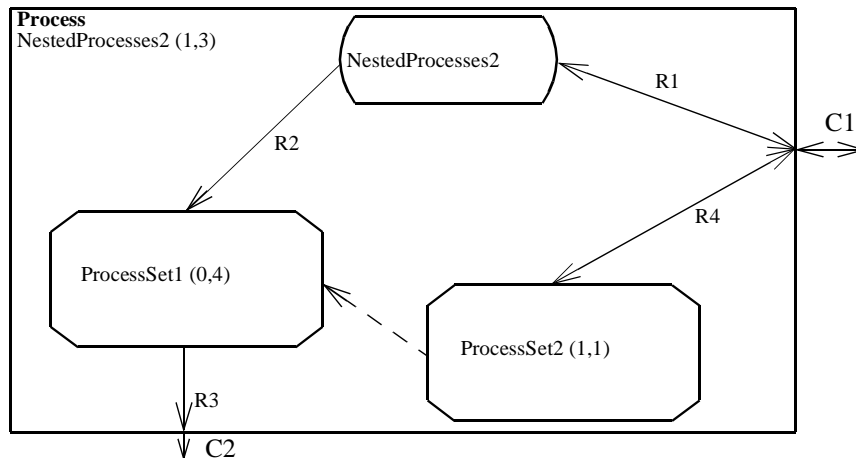
blocks. Fig. 1. shows an example for the application and the definition of a *block process*. The *block process* `NestedProcesses` has an own state machine (denoted by the state symbol) and two contained processes (`ProcessSet1`, `ProcessSet2`). The `Process` `ProcessSet1` is again a (decomposed) *block process*, what is shown by the new *block process* symbol. It should be noted that a decomposition of a *block process* implies a concurrent execution of the contained process instances and of the state machine of that *block process*.

The conventional concepts of block and process can be seen as two special cases of a *block process*:

- A block is a *block process*, which has no state machine nor local variables, parameters, etc.; it only contains process definitions.
- A process is a *block process*, which only has a state machine, optional local variables, parameters, etc., but no process definitions.

## 2.2 Alternating Decomposition

In contrast to concurrent decomposition, alternating decomposition implies that every contained process instance executes its transition to completion. This means, there is only one process instance executing at a time. Alternating decomposition is expressed very similar to concurrent decomposition, it is denoted by the conventional process symbol and the keyword `block` is omitted.



**Fig. 2.** Alternating decomposition of a process.

The alternating execution of the processes in a process resembles the existing concept of services, however, with the following differences:

- Each process instance within a process has its own input queue and its own unique process identifier (Pid), while services shared these features with their containing process.
- Process instances within a process may be dynamically created and destroyed, while services live and die with their containing process.
- The valid input signal sets of the services of a process must be disjoint; that is not

the case for processes in processes.

### 2.3 Data Sharing.

Local variables defined in a *block process* or in an alternating decomposition of a process are shared by all contained processes and by the state machine of the container, so they are global for all contained process instances. However, due to the different type of execution model, there are some important differences between data sharing in a concurrent and in an alternating decomposition.

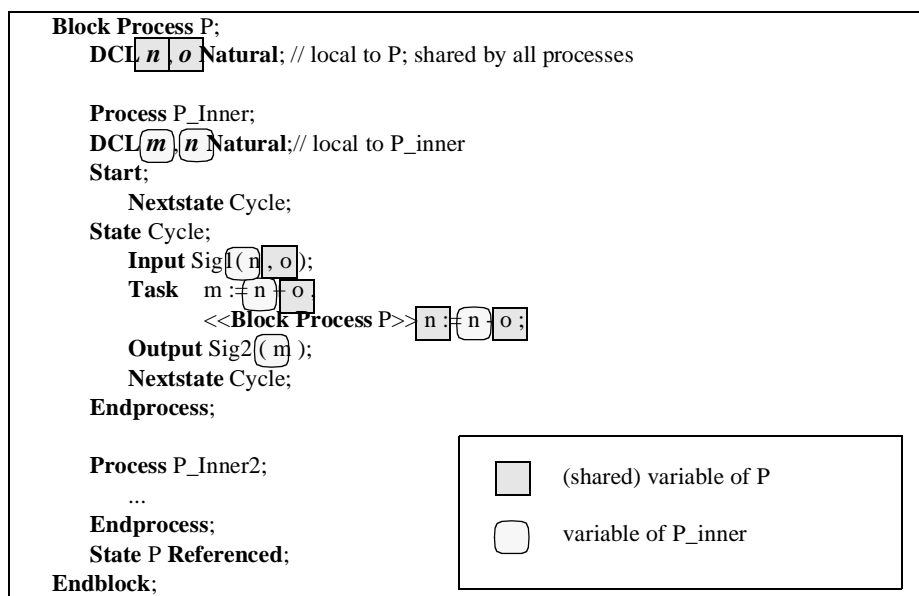


Fig. 3. Sharing of Data.

**Concurrent Decomposition.** All processes have equal rights to read and write such a global variable. It is guaranteed, that there are no simultaneous read/write operations, as the reading/writing is performed by the owning container process (by execution of get-/set-procedures). On the other side, due to the concurrent nature of the processes it can not be guaranteed for a process, that two subsequent access operations to such a variable will return the same value.

**Alternating Decomposition.** Global variables can be considered as truly shared variables. The forced interleaved execution of the transitions of all contained processes and the run-to-completion semantics guarantees not only the absence of read/write conflicts between different processes, but also the property that two subsequent read operations to such a variable will always return the same value. Only the currently running process may change the value of such a variable.

To use a global variable in an expression no special syntactic constructs are needed, due to the visibility rules it can be used in the same way as a locally defined variable. It may be, though, necessary to qualify the name of the variable.

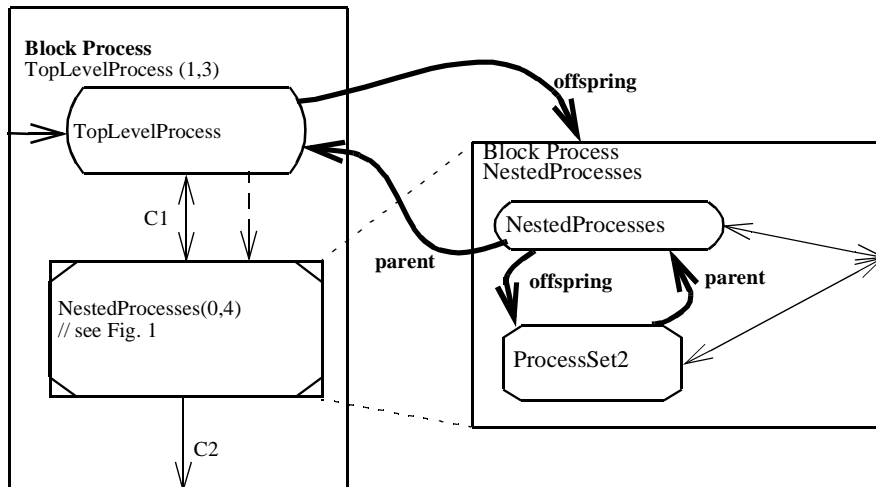
## 2.4 Dynamic Creation of Decomposed (Block) Processes

As *block process* and alternating decomposed processes still have all the features of conventional processes, it is possible to create such containers dynamically within the constraints given by the initial and maximum number of instances.

The dynamic creation is a two-step procedure:

- First, the conventional process with the state machine of the container and all local variables, procedures etc. of the container is created.
- Second, this initial process then creates the appropriate initial number of process instances for all contained processes as first actions of its start transition. Now all contained process instances do execute their start transition.

Afterwards the normal concurrent or alternating execution of the processes begins.



**Fig. 4.** Creation of block process NestedProcess by TopLevelProcess.

As a result the following different values for the implicit parent and offspring variables may be observed:

- Offspring of external creator: PID of the container process.
- Parent of container process: PID of the external creator.
- Parent of all contained processes: PID of the container process.
- Offspring of container process: PID of the last contained process created last.

In case that the contained processes are again containers, this procedure will be repeated on that level. The same rules apply also for the introduction of the initial instances, although there is no external creator.

Complementary, also the rules for the stopping of processes have been adapted. Stopping of a contained process is just the same as stopping a conventional process. Stopping of the state machine of the container itself on the other side results in a stopping of all its contained processes too. That is, no process will outlive the container it belongs to.

### 3 Type Based Definitions

The definition of process types has also been enhanced by composition principles. It is possible to define a *block process type* (specifying a concurrent and structural decomposition) as well as a *process type* (specifying alternating decomposition). Type based

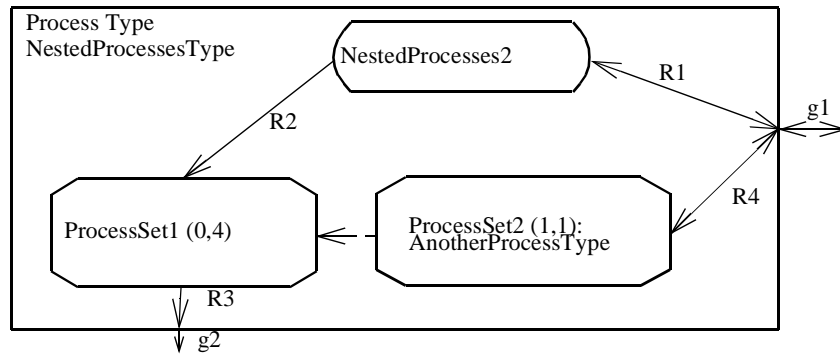


Fig. 5. Process type definition.

instance definitions can than be defined derived from such a type definition . To allow

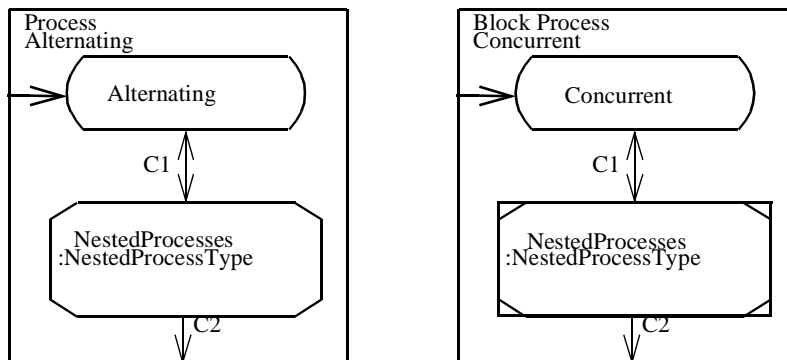


Fig. 6. Different kinds of type based definitions.

a higher degree of flexibility, a process type definition can be used for as well a type based *block process* definition as for a type based process definition. That implies, that the kind of type based definition is critical for the execution type (concurrent or alternating). A *block process* type on the other side may only be used for type based *block process* definitions. For this type of processes always concurrent execution is required.

### 4 Semantics

There are two general ways to define the semantics for the new enhanced process concept:

- Provide a direct mapping to the formal semantics model of SDL,
- Provide a transformation to basic SDL (which has a well defined semantics).

The authors have chosen the second solution. Many existing enhanced concepts of SDL have their semantics defined also by a transformation to basic SDL constructs, so this will also be the natural way here. Additionally, the development of a new semantic model for SDL is on the way, so for the future also a direct semantic foundation could be given.

#### 4.1 Concurrent Decomposition

A *block process* is modelled by a collaboration of processes. There is one process instance for the state machine of the container and a set of process instance sets for all contained processes. For each variable of the *block process* a corresponding local variable is defined in its process. Additionally this process exports for each local variable two procedures, one procedure to get the value of the variable and one procedure to write the value. Within the contained process each access to a variable of the container is replaced by a call of the appropriate set or get remote procedures to the container process.

The start transition of the container process is extended by a series of create-request actions, which create all initial instances of the contained processes. These create-request actions are the first actions in the modified start transition. The order of the create-requests is arbitrary. For each contained process exist a signalroute/channel from the container process to the contained process, which at least transmits the signal kill. The stop of the container is replaced by:

```
output kill via all; // send a kill to all instance sets
stop;
```

and the following state is introduced:

```
state *;
priority input kill;
output kill via all; // send a kill to all instance sets
stop;
```

This replacement is also applied to all procedures of the container process. The signal kill will be added to the valid input signalset of all contained processes. The following state is added to all contained processes and to all of their procedures:

```
state *;
priority input kill;
output kill to this;
// forward kill to next process of same set
stop; // if no instances exist, kill is abandoned
```

These additions guarantee a well-ordered stopping of all contained instances for the case that the container process stops.

#### 4.2 Alternating Decomposition

The transformation model for alternating decomposition is significantly larger. This is due to the fact, that is not only in charge of shared data and creation/stopping of instanc-

es, but also for the scheduling of the different process instances.

A process with alternating processes is modelled by introducing a monitor process that serialises the otherwise concurrent processes. A process set  $p$  with enclosed process sets ( $ps$ ) and state machine is transformed into the following (as part of the enclosing block):

- one process set, monitor, with the same number of instances as for  $p$
- a process  $tp$  that has the process graph of the state machine and all properties of  $p$ .
- Each  $tps$  as well as  $tp$  have one additional formal parameter of type  $Pid$ ;
- transformed  $ps$  processes,  $tps$ , with the following properties:
  - Initial number of instances of  $tp$  and of  $tps$  will be zero.
  - Maximum number of instances of  $tp$  is the same as for  $p$
  - Maximum number of instances of  $tps$  is  $\max\text{-number-}p * \max\text{-number-}tp$
  - Initial number/maximum number of instances of monitor is equal to initial/maximum number of  $p$

**Step 1.** A creation of  $p$  is realized by a creation of a monitor instance. Monitor then creates the  $tp$  it belongs to and all initial  $tps$ 's of that  $tp$ , giving its own  $Pid$  (self) as a parameter.

**Step 2.** All initial start transitions of the contained processes and of the process  $tp$  itself are scheduled.

**Step 3.** Normal scheduling of the transitions of the alternating process instances begins (under control of the monitor).

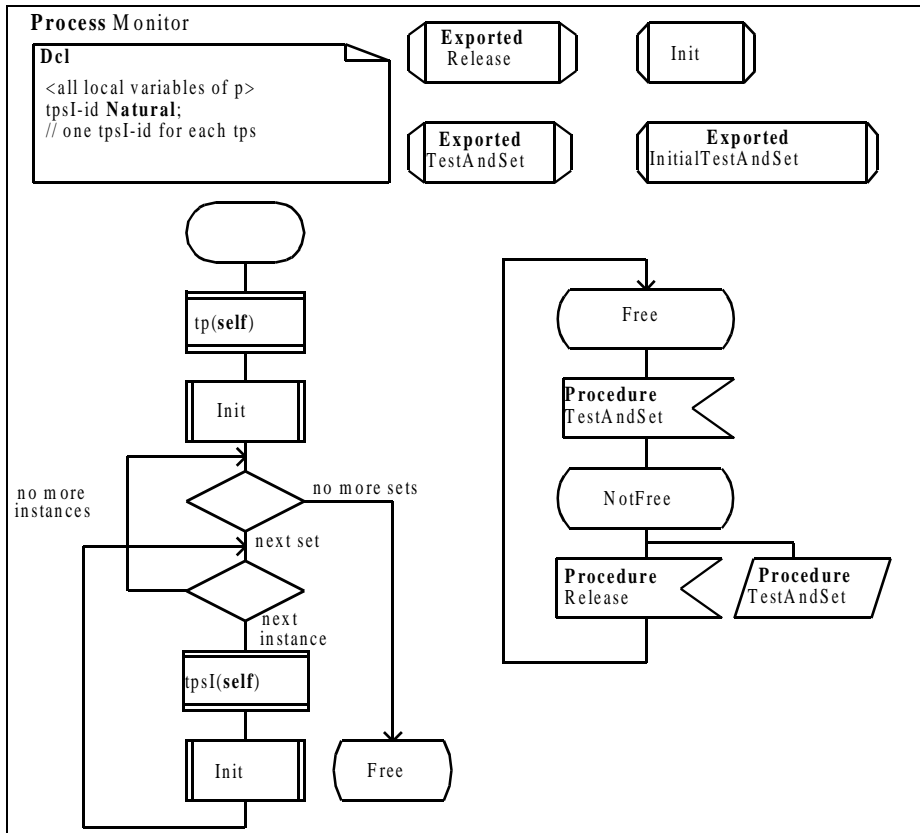
As a result, each instance of  $p$  is modelled by a set of collaborating process instances, which belong to different process sets. There is one instance of monitor, one instance of  $tp$  and the appropriate number of  $tps$ . For each of instance in  $tp$  there will be two concurrent processes executing: the common monitor instance and one of the  $tp$  instance and its  $tps$  instances. Monitor serialises  $tp$  and  $tps$ .

In Fig. 7. the definition of the process monitor is given. In its start transition the monitor creates all initial instances in the contained processes and schedules the start transitions of these instances (nested loops in start transition of monitor). After that the monitor goes into the Free-state and the normal alternating scheduling of the transitions of the contained processes begins. A process requesting the permission to execute a transition call the procedure `TestAndSet` (exported by monitor). As soon as this call returns, he is enabled to proceed with his transition. A call of `Release` at the end of the transition gives control back to the monitor.

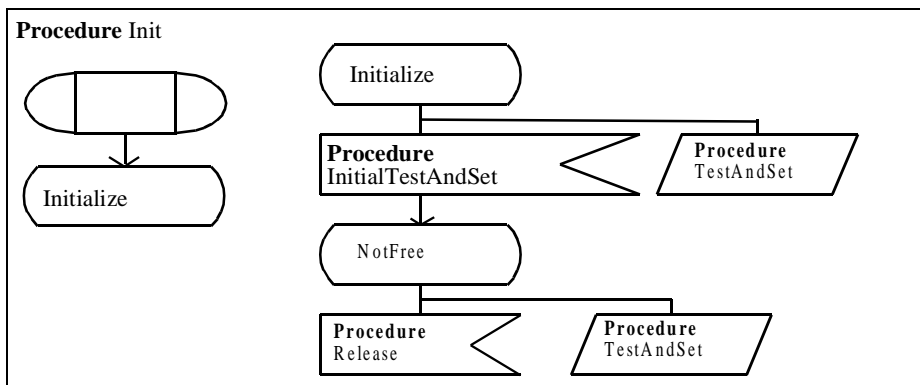
The body of the procedure `Init` (see Fig. 8.) is very similar to that of the monitor. It does, however, schedule only transition execution request made with a call of `InitialTestAndSet`, all calls to `TestAndSet` are deferred by a `save` statement.

The exported procedures `TestAndSet`, `InitialTestAndSet` and `Release` consist of a start transition only. They are explained at the end of this section in more detail.

All contained processes, the state machine of the container and all local procedures of these processes are modified according to the following scheme:



**Fig. 7.** Process Monitor.



**Fig. 8.** Procedure Init

Every start transition of p and ps (but not in local procedures) will be changed in tp resp. tps to:

```
Start;
  Decision parent = monitor;
    True: Call initialtestandset to monitor ;
           // this is an initial instance
    Else: // this is an instance created by somebody else
           Call testandset to monitor ;
           // require execution; only one will get it
  Enddecision;
  // now the remaining original transition
```

Every nextstate of p and ps (also in local procedures) has to be preceded by:

```
Call release(...) to monitor; // give execution back to monitor
Call testandset to monitor; // require execution; this will
                               // block the process until the
                               // monitor is in state free and
                               // executes this rpc
```

```
Nextstate <state>
```

Because TestAndSet switches the monitor from state Free to NotFree, exactly one process at a time will get the permission to execute a transition. In order to give the permission back in case of an empty input queue, each tp and tps gets a state:

```
State *;
  Provided true;
    Call release to monitor;           // release execution control
    Call testandset to monitor;       // require execution again
  Nextstate -;
```

Because the monitor is in charge of the maximum numbers of the tps belonging to it, all create and stop actions have to be changed. Each create-request action for a tps in tp or tps is replaced by:

```
Decision (call tps-create to monitor);
// the tps-create procedure of the process to be created
  True: Create <tps>;
  Else: Task offspring :=null;
enddecision;
```

Each **stop** in tps has to be preceded by

```
call tps-stop to monitor; // the tps-stop of itself
call release(...) to monitor;
```

The procedures tps-screate and tps-stop increment resp. decrement the appropriate instance counter in the process monitor. If an attempt is made to create more than the maximum number of instances, tps-create will return false, otherwise it returns true. The stopping of the container is solved similar to the stopping of a concurrent decomposition (cf. Sect. 4.1), and results in the stopping of all contained processes, the state machine of the container and the stopping of the related monitor.

To prevent two contained instances from a concurrent execution of remote procedures, all implicit remote procedure inputs are made explicit. Remote procedure inputs in p and ps has to be transformed as follows:

**Explicit remote procedure inputs:** Nothing has to be done; nextstate is already changed, procedure itself has also been transformed

**Implicit remote procedure inputs.** For each exported remote procedure *rp* the following state is introduced:

```
State *(st1,st2,...) // st1,st2 are those states having the
Input procedure rp;// procedure explicitly in an input/save
Call release to monitor;
Call testandset to monitor;
Nextstate -;
```

*Remark.* Because the processes execute alternating, a remote procedure call between to processes in the same containing process will result in a deadlock.

This transformation model assures that transitions of *p* and *ps* are executed to completion and interleaved. Access in these transitions to variables of *p* can therefore not be modelled by having *tp* execute remote set- and get procedures. The model for global variables is therefore as follows:

- The variables of *p* will all be made local variables of its monitor. Each instance of *tp* as well as *tps* has for each variable *v* in *p* a local variable *localv* of the same type for each variable of *p*. Global variable access in transitions is transformed into local variable copy access.
- For each variable of *p* there is a corresponding parameter in the monitors TestAndset and Release procedures.
  - InitialTestAndSet and TestAndSet have these parameters as in/out, and they assign the variables in the monitor to these parameters. The call of InitialTestandSet and TestAndSet will have the local copies as actual parameters
  - Release has these parameters as in, and it assigns the parameters to the variables of the monitor. It is also called with the local copies as actual parameters.

## 5 Related work

Many object-oriented analysis and design techniques base the description of behaviour similar to SDL on extended finite communication state machines.

UML provides many different means for the structural and behavioural decomposition. Active objects may contain other active or passive objects, implying a single or multiple threads of control. A single state of a state machine can be decomposed into (orthogonal) concurrent substates. The execution of transitions follows always a run-to-completion (RTC) semantics. Due to this fact, and the orthogonality of concurrent substates, an interleaving of concurrent transitions is always possible. Although UML has special annotations to specify the rules for a concurrent external access to an object (call of operations), the semantics guide does currently not contain rules for the access to shared attributes of a container. The three ways to handle concurrent operation calls are:

- sequential: external synchronisation is required
- guarded: concurrent calls are sequentialized by the called object
- concurrent: all calls are executed simultaneously.

A detailed execution model is not (yet) given in the UML definition

The ROOM notation (Real-Time Object-Oriented Modeling, [3]) has the concept of concurrent entities called actors. An actor may have both state machine and a decomposition into sets of actors. Actors are defined by actor types. ROOM does, however, not support nesting of definitions. This means that actor types can not be locally defined to an actor type, but only in packages. An actor can be instantiated in another actor, but during runtime the actors are flattened out as completely encapsulated objects. All forms of communication between actors are via ports and bindings and there is no such thing as global shared variables in ROOM. Variables in the container actor are thus not visible in the contained actors. The implication of this is that a container actor is more or less just a container. The structure of contained actors can not reflect the effect of their execution in the common container actor, so the reason that they are contained in the same actor is only for generation of actor structures.

The general approach of BETA [4], with patterns that generalises the concepts of classes, types, functions and methods, and with a general nesting of pattern definitions, the approach to shared variables is like the one presented here. A pattern may be used to generate a concurrent object, and nested patterns may be used to generate concurrent objects. Variables of the container concurrent object are thus visible from the contained concurrent objects.

## 6 Conclusion and Further Work

The unified block and process concept does solve the need of SDL users for extended possibilities to define the structure of an SDL system. It also provides a solution for the dynamic creation of decomposed units and a clear separation between structural and behavioural decomposition. After an introduction of these new features into SDL the existing concepts of block and service can be first characterized as special cases of *block processes* and be removed completely in a later language revision.

Although the semantics of the concepts is entirely explained using a transformation to basic SDL, a direct semantic foundation would be the preferred solution. It would not only help to reduce the restrictions, which are due to the transformation rules, but also make tools for the analysis of specifications more efficient. The current work towards a new formal semantics for SDL [7] based on the ASM concept (Abstract State Machine), intends to provide sufficient concepts for such a direct semantic foundation.

## References

1. ITU-T: Rec. Z.100 - *SDL - ITU-T Specification and Description Language*, ITU-T, 1996.
2. OMG: *The OMG Unified Modeling Language Specification (UML V1.2)*, OMG, 1998.
3. B. Selic, G. Gullekson, P. T. Ward: *Real-Time Object-Oriented Modeling (ROOM)*, Wiley&Sons, 1994.
4. O.Lehrmann Madsen, B. Møller-Pedersen, K. Nygaard: *Object Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993.
5. B. Møller-Pedersen: *SDL/UML Alignment: Composite States*, Ericson AS, 1998
6. E. Holz: *Application of UML in the SDL Design Process*, Proc. of SAM98, 1998.
7. U. Glässer, A. Prinz: *An ASM semantics for SDL*, submitted for SDL'99, 1999.