

The formal semantics of a domain-specific modelling language

Colin Grealish

Kellogg College

M.Sc. by Research
University of Oxford
Trinity Term 2009

Abstract

The Specification and Description Language (SDL) was developed by the telecommunications industry as a common language for the design of asynchronous communicating systems. Designs expressed in SDL can be given a formal, mathematical interpretation to facilitate analysis and to guide correct implementation. The standard formalisation of SDL is adequate for the purposes of language definition, but is not oriented towards either analysis or implementation of specific designs.

This thesis investigates alternative approaches to the formalisation of SDL. A specific, constrained version of the language is considered: one more amenable to code generation. A state-based formalisation is presented using the Z notation, together with a process-based account using the language of Communicating Sequential Processes. A more abstract, elegant solution is then proposed, in terms of a calculus of buffered systems, itself given an interpretation using Hoare and He's Unifying Theories of Programming. A case study is presented to demonstrate the practical application of the formalisations.

Acknowledgment

I would like to thank the staff of the Computing Laboratory for their help and support. I am also very grateful to Ericsson Ltd, who gave me the opportunity to undertake this research.

Contents

I	Introduction	1
II	Modelling Buffered Systems	3
II.1	Representing communication	3
II.2	Specification and Description Language	3
II.3	SDL-10	4
III	A formal semantics for SDL-10	6
III.1	Processes	6
III.1.1	Example	8
III.2	Models	10
III.3	Behaviours	11
III.4	Application	15
IV	A Calculus of Buffered Systems	16
IV.1	Notation	16
IV.2	Input and output	16
IV.3	Termination	18
IV.4	Composition	18
IV.5	SDL Models	18
IV.6	Example	20
V	Case Study	25
V.1	SDL into UML	25
V.2	Example: a game application	27
V.3	Targetting SDL-10	39
VI	Conclusion	41
VI.1	Related work	42

I

Introduction

We deal with the complexity of modern systems by constructing models: abstract representations of structure or behaviour. We use these models to understand, to describe, and to examine the properties of the systems they represent; we may even use them to generate components of the systems themselves. Even for simple models, a rigorous examination of properties requires a formal foundation: We need languages, methods and tools to help us calculate the consequences of design and modelling decisions. Several formal languages have been developed to support rigorous analysis, from VDM [8] and Z [21] to precisely-defined subsets of the widely-used Unified Modeling Language [17]. However, the analysis of a particular class of systems, or a particular class of models, remains problematic.

In distributed systems, such as those found in networked (or grid) computing environments, or in telecommunications, it is often necessary to consider the sending of a message separately from its receipt. In order to examine key properties, we need to consider what may occur between these two points; We may need also to consider what happens when several messages may be ‘in flight’ between different components. The resulting multiplication of states and entities can lead to models that are too complicated to understand, and in which the effort required for any rigorous analysis may be prohibitive.

The Specification and Description Language (SDL-92) [9] was designed for the representation of exactly this kind of *buffered* system. In SDL, processing entities communicate asynchronously, through ‘signalroutes’ in which messages may be delayed or even lost, cause and effect are properly separated, and yet the model remains comprehensible. Despite the initial promotion of SDL as a ‘formal technique’ [22], little has been done to support practical, rigorous analysis of models written in the language.

This thesis makes three principal contributions:

- A formal semantics for a domain-specific version of SDL, called SDL-10 [3], developed by Ericsson for the purposes of software component generation. This semantics has been published in IEE Proceedings [4], and has had some impact upon development within Ericsson: it was used as a means of illustrating the differences between alternative SDL toolsets—none of which provided full support for SDL-10—and explaining key issues regarding their use in supporting actual SDL-10 implementations.
- A process calculus—the Calculus of Buffered Systems (CBS)—based upon the Communicating Sequential Processes (CSP) [6], which can be used to provide a more abstract characterisation of SDL processes. The calculus can be used in the same way as SDL-10 can, in model creation and analysis, but without limiting the user to a particular subset of SDL.
- A case study, which demonstrates the application of these languages to the design of a simple

embedded system on a standard, commodity platform. It demonstrates also that the approach works equally well with concurrent designs.

We may bring these contributions together in the form of a framework for the design and development of buffered systems: see Figure I.1. The leftmost column in the diagram represents the process of software development in PLEX for AXE-10 systems. The middle and rightmost columns represent software creation for Java environments (although these processes are equally applicable for similar languages, such as C#).

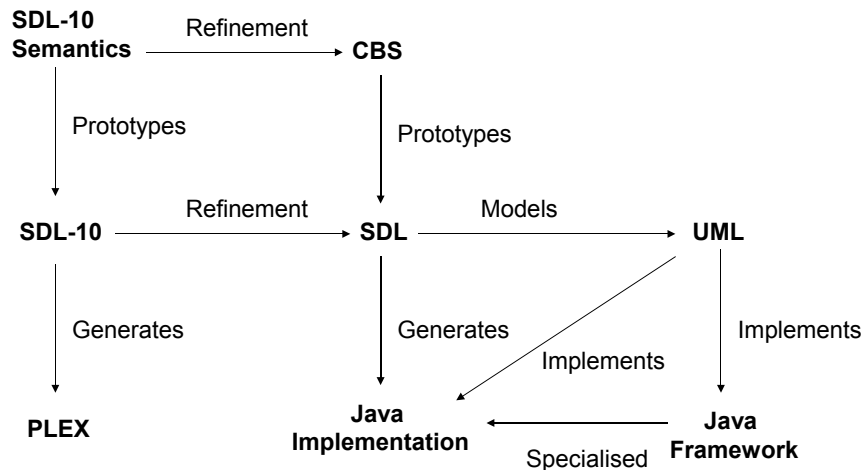


Figure I.1: A framework for modelling and development

The top level of the diagram represents prototyping activity using the semantics. This has the advantage of allowing rigorous exploration of the design at the concept stage—the semantics serves as a halfway house between informal concept and fully-fledged SDL-10 model. The middle level deals with more conventional modeling, using the SDL-10, SDL, and the use of a UML profile. The bottom level deals with the actual generation of executable code. Different refinement relations can be defined, both horizontally and vertically, between the languages in the diagram.

During the development of this body of work SDL has undergone some significant changes. It is based upon the SDL-92 version of the language, which remains the most widely used. The semantics of SDL were updated with SDL-2000 [10], although by that point UML was emerging as a realistic or preferred alternative in many organisations. Any significant semantic differences between SDL-92 and SDL-2000 are highlighted in the text.

II

Modelling Buffered Systems

II.1 Representing communication

The simplest form of communication to reason about is the *synchronisations* or *transaction*, in which all parties agree on the point at which the communication takes place, the fact that it has taken place, and the value of any data transferred. This is the natural building block for models of distributing computing, and indeed for any model of communicating or interacting behaviour.

However, if the model is a representation of a system in which entities are communicating asynchronously, with messages being buffered in queues, then a description based simply upon synchronisations may not be *fully abstract*: it may include information that we do not need to consider—and our descriptions of properties and components may be rendered incomprehensible.

As a simple example, consider a system in which two entities communicate asynchronously by means of some possibly unreliable medium. A description based upon synchronisations might represent a single communication as the sending of a message, its successful transmission, and its eventual acceptance or removal from an input queue: three pieces of information, where one might suffice.

For this reason, modelling languages have been proposed in which asynchronous communication is available as a primitive notion.

II.2 Specification and Description Language

The Specification and Description Language (SDL) was first proposed in 1972, although it was not until 1980 that the first language specification was made available as an ITU (International Telecommunication Union) recommendation [23]. The specification has been updated since then, with the last update being approved in 2002, and an ‘implementer’s guide’ issued in 2007.

SDL is intended for the description of the kind of reactive, discrete systems found in telecommunications architectures: systems whose behaviour is completely characterized by discrete interactions with the environment. An SDL description is couched in terms of extended finite state machines, or *processes*. The level of abstraction or granularity used in defining processes will vary from specification to specification: typically, however, a node in a network will be represented as a single process. For example, in a model of a transit telephone exchange, the activity involved in processing an incoming call, and switching it to a destination, might be captured as a process—whether this process represents a candidate implementation, or merely an encapsulation of a requirement.

In concrete designs, each sequential program or process is typically modelled as a single SDL process, and the SDL model allows us to explore the interactions between them. In the case of the

telephone exchange, one process might represent the program handling incoming calls, another might represent the program handling the route selection, and another might represent the program handling the setting up of outgoing calls.

Communication between processes, and between processes and the system environment, is achieved by means of *signals*. The mode of communication is asynchronous, with signals being held in input queues and possibly-unreliable delaying *signal routes*. When a signal is received from an input queue, it is absorbed by the process, whether or not the process is ready to respond, unless the *save* construct is employed—in which case the signal is held locally, as part of the state of the process, until the process is ready for it.

Process behaviour is then defined in terms of transitions, triggered by signals, between regions or states. In a telephone exchange model, a process might begin in an “idle” state, move to a “call processing” state, and then to a “releasing circuits” state. Each state will have a number of signal entries, declaring actions to be performed should the corresponding signal be received when the process is in that state.

Transitions between states are functional, in that they point to a single target state, but non-deterministic behaviour can be captured using unlabelled transitions, which link pairs of states without the need for an observable trigger—in this way, a tree of alternative outcomes can be defined for a single initial transition, triggered by a single signal. Timed behaviour is also represented using signals: a process may register a timeout, and then receive a signal when the specified time expires.

II.3 SDL-10

The SDL language is an international standard developed by experienced engineers from a range of organisations. Like many other notations, it has a graphical syntax for ease of use by, and communication with, customers and engineers without a deep technical understanding of the textual, declarative ‘programming notation’. Unlike many other notations, including those of the UML, it is associated with a single, standard dynamic semantics and interpretation model.

For this reason, the Ericsson organisations developed a revised, proprietary version of the language specifically for the purpose of automatic code generation. The SDL-10 [3] notation uses some of the standard SDL syntax, but extends it with directives that define precise behavioural semantics. For compatibility with existing tools, the directives are included as comments: SDL-10 can thus be parsed as a subset of standard SDL.

Much of the software developed for the Ericsson AXE-10 [2] series of digital telephone switches was written in SDL-10, and compiled into a low-level language called PLEX (for Programming Language for EXchanges). The design of PLEX, and the interpretation of SDL-10, reflects the hardware architecture of the AXE-10 switches. The operating system on AXE-10 employs a single hardware buffer for task distribution. In SDL-10, it is assumed that there is only one buffer for internal process communication; signal output from a process is always possible, and signal buffers may overflow.

Furthermore, the AXE-10 architecture has all processes executing on a single processor, without time slicing or process scheduling. As a consequence, we assume that SDL-10 processes are interruptible only when they reach a stable state: no interrupt can occur between a signal entry—where the processing of a signal begins—and the arrival at the subsequent state—once the processing has ended. This greatly reduces the number of ‘control states’ that need to be considered in an analysis.

SDL-10 can also be used to describe the behaviour of a system at a higher level of abstraction, above the level at which code generation is possible. A single SDL-10 process can be presented as a ‘black box’ view of the system, in which only the inputs and the outputs are specified, and no

mention is made of any of the internal structure. Such a *process model* contains more information than *instance* models such as sequence diagrams or sequence charts: It describes all of the possible sequences of execution, at that level of abstraction.

A process model in SDL-10 can be used to check and complement message sequence charts in a component specification. We can test the message flows from sequence charts to see if they are allowed by the process model, and use sequence charts to illustrate use cases and message flows of particular importance. We may then use the validated charts to test subsequent, design-level descriptions.

At the design level, a model consisting of multiple SDL processes, each provided with sufficient detail to support automatic code generation, can be checked in this way using tools such as the Ericsson SDLtool. A failed check may lead to a revision of the design, or it might also lead to a modification of the sequence chart in question, as the design level description might reveal some inadequacy or error in its design not uncovered by the more abstract 'black box' model.

The final stage of development involves translating the blocks of SDL into blocks of PLEX for execution on the AXE-10 platform. Each SDL-10 process corresponds to an 'individual' in PLEX, typically representing a single call. The directives in the SDL-10 provide the necessary information about signal priorities, variable size, and memory location for tables. SDLtool generates the PLEX code, which is tested using test cases generated from the original sequence charts and the 'black box' process model.

The Ericsson SDLtool is a proprietary implementation of the Telelogic Tau tool suite, which was based upon the SDL-92 semantics of the SDL language. The SDLtool provides proprietary extensions for the generation of PLEX code for AXE-10 switches. The language semantics embodied in the tool and the extensions is that of SDL, not SDL-10, and this was one of the motivating factors in this research: to understand these differences, and better inform users and developers.

The complexity of the language and the orientation of the semantics makes a fully formal analysis of the complete language impractical. Instead, we have chosen to focus upon differences in the core elements of the language, and in particular upon the way in which processes interact using buffers. This entails a focus upon *State*, *Signal*, *Process*, and the interplay between them.

III

A formal semantics for SDL-10

We will give a semantics to SDL-10 using the set-theoretical Z notation [21]. The semantics described is targeted towards an AXE-10 and PLEX implementation: It is essentially a modeling of the SDL-10 subset of SDL as a number of interworking state machines. As stated, our focus will be on *State*, *Signal*, *Process* and their interplay. The hardware limitation of buffer overflow is *not* considered—in practice, most buffer overflows are due to low-level programming errors, rather than mistakes in modelling.

III.1 Processes

We will need a set of names for signals and states, as we do not need to explore their representation in any detail:

$$[SIGNAL, STATE]$$

We will find it useful to define an abbreviation for the set of all sequences of signals:

$$TRACESEGMENT == seq\ SIGNAL$$

We will need the notion of a task set in order to model process activation and termination: two values will be enough for the moment.

$$TASK ::= null \mid stop \mid \dots$$

We will complete this definition later.

We will refer to the combination of a trace segment and a task as an *SDLSEGMENT*:

$$SDLSEGMENT == TRACESEGMENT \times TASK$$

The combination of a state and an *SDLSEGMENT* models a sequence of signals, a task, and a state—this constitutes the output component of an SDL transition:

$$OUTPUTS == STATE \times SDLSEGMENT$$

The input of a transition, on the other hand, may be represented as simply the combination of a state and a signal:

$$INPUTS == STATE \times SIGNAL$$

We require at least one, dummy instance of *STATE*: this will be the other component when the second component of the pair is *stop*.

$dud : STATE$

The body of an SDL process may then be represented as a binary relation between inputs and outputs

$$SDLBODY == INPUTS \leftrightarrow OUTPUTS$$

Each input and state combination is related to one or more states, sequences of signals and/or a task termination.

The SDL-10 language includes an operator called ANY, which represents a nondeterministic choice of consequences: a single input can give rise to multiple, alternative outputs. The same is true at initialisation, although here there is no input to consider:

$$SDLINITIAL == \mathbb{P} OUTPUTS$$

The combination of a process body and an initialisation is enough to completely characterise an SDL process. For convenience, we define

$initial : SDLProcess \leftrightarrow SDLINITIAL$ $body : SDLProcess \leftrightarrow SDLBODY$ $inputs : SDLProcess \leftrightarrow \mathbb{P} INPUTS$ $outputs : SDLProcess \leftrightarrow \mathbb{P} OUTPUTS$ $states : SDLProcess \leftrightarrow \mathbb{P} STATE$ $isStateOf : STATE \leftrightarrow SDLProcess$
$initial = fst \wedge body = snd$ $inputs = (dom) \circ body$ $(\forall p : SDLProcess \bullet$ $\quad outputs\ p = initial\ p \cup \text{ran}(body\ p) \wedge$ $\quad states\ p = \text{dom}(inputs\ p) \cup \text{dom}(outputs\ p) \wedge$ $\quad states\ p = isStateOf \sim (\{p\} \Downarrow))$

$\overline{[X]}$ $state : (STATE \times X) \leftrightarrow STATE$
$state = fst$

$segment : OUTPUTS \leftrightarrow \text{seq } SIGNAL$ $task : OUTPUTS \leftrightarrow TASK$
$segment = snd \circ fst$ $task = snd \circ snd$

This simplifies the expression of the single additional constraint upon the semantics of processes:

$SDLProcess : \mathbb{P}(SDLINITIAL \times SDLBODY)$
$\forall p : SDLProcess \bullet$ $(\forall o : outputs(p) \bullet task\ o = stop \Rightarrow state\ o = dud)$

The lack of any further constraint in this declaration reflects the lack of any further restriction upon language usage. In SDL-10, it is possible to reference a state that does not exist, and we will need to consider this possibility in our semantics.

To obtain a description of the complete behaviour of a process, we need to consider its response to inputs outside the domain of the body relation. In SDL-10, any such input is discarded without comment, output, or change of state. We may include this information in our semantics using the following closure operator:

$$\begin{array}{|l}
 \hline
 _b : \text{SDLProcess} \rightarrow \text{SDLProcess} \\
 \hline
 \forall p : \text{SDLProcess} \bullet \\
 \quad \text{initial}(p_b) = \text{initial}(p) \wedge \\
 \quad \text{body}(p_b) = \text{body } p \cup \\
 \quad \{ st : \text{states } p; si : \text{SIGNAL} \mid (st, si) \notin \text{inputs } p \bullet (st, si) \mapsto (st, (\langle \rangle, \text{null})) \} \\
 \hline
 \end{array}$$

III.1.1 Example

Consider as an example the simple SDL-10 process shown in Figure III.1, and presented in textual form below. This process has two ANY symbols, two states, three input signals, three output signals and a termination.

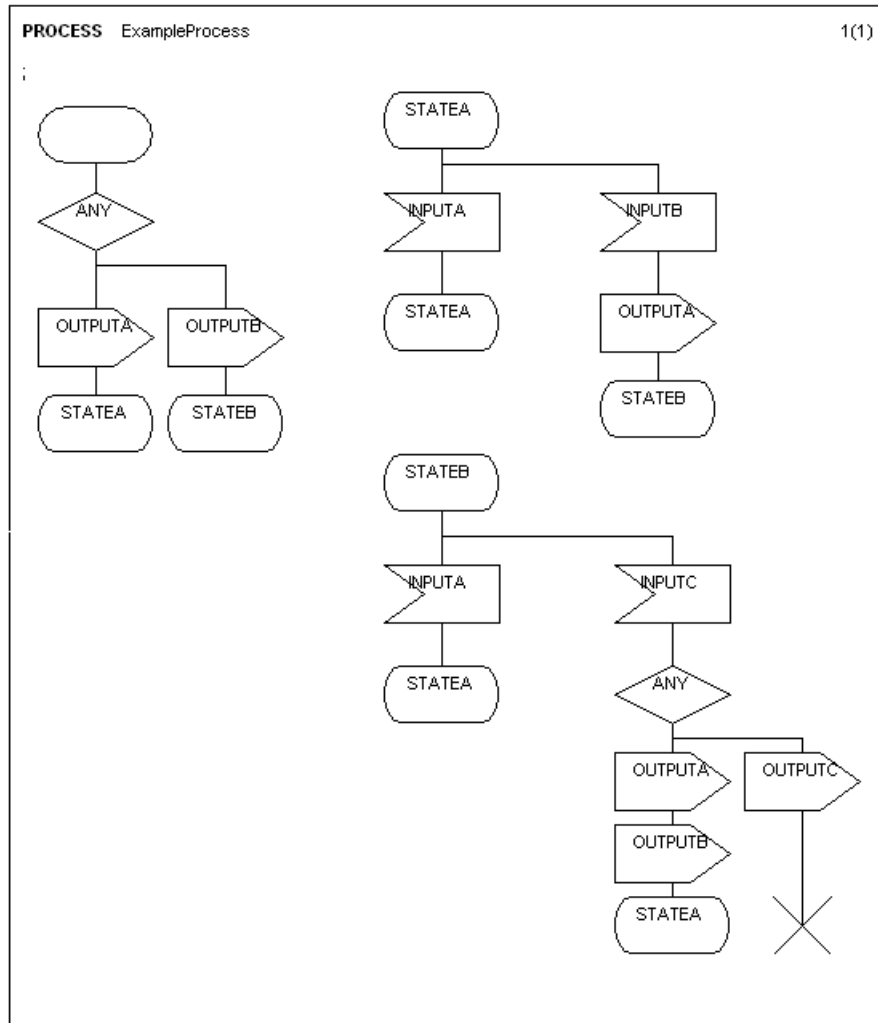


Figure III.1: ExampleProcess process

```

Process ExampleProcess;
  start ;
  decision ANY;
  ():
    output OUTPUTA;
    nextstate STATEA;
  ():
    output OUTPUTB;
    nextstate STATEB;
  enddecision;

  state STATEA;
    input INPUTA;
    nextstate STATEA;
    input INPUTB;
    output OUTPUTA;
    nextstate STATEB;
  endstate;

  state STATEB;
    input INPUTA;
    nextstate STATEA;
    input INPUTC;
    decision ANY;
    ():
      output OUTPUTA;
      output OUTPUTB;
      nextstate STATEA;
    ():
      output OUTPUTC;
      stop;
    enddecision;
  endstate;
endprocess ExampleProcess;

```

In our semantics, with the definitions

	<i>STATEA, STATEB : STATE</i>
	<i>OUTPUTA, OUTPUTB, OUTPUTC,</i>
	<i>INPUTA, INPUTB, INPUTC : SIGNAL</i>

we can model this process as

ExampleProcess : *SDLProcess*

ExampleProcess =

({ (STATEA, (⟨OUTPUTA⟩, null)),
 (STATEB, (⟨OUTPUTB⟩, null)) },
 { ((STATEA, INPUTA) ↦ (STATEA, (⟨⟩, null))),
 ((STATEA, INPUTB) ↦ (STATEB, (⟨OUTPUTA⟩, null))),
 ((STATEB, INPUTA) ↦ (STATEA, (⟨⟩, null))),
 ((STATEB, INPUTC) ↦ (STATEA, (⟨OUTPUTA, OUTPUTB⟩, null))),
 ((STATEB, INPUTC) ↦ (dud, (⟨OUTPUTC⟩, stop))) }) ⊢

III.2 Models

Where an SDL-10 process is used as a model of required behaviour, it will appear as a single, sequential component: it is not allowed to activate another.

SingleProcess : $\mathbb{P}$ *SDLProcess*

$\forall P : \text{SingleProcess} \bullet$

($\forall o : \text{outputs } P \bullet \text{task } o \in \{\text{null}, \text{stop}\}$)

We will often wish to determine the set of all reachable states allowed by a specification—we define

next : *SDLProcess* $\leftrightarrow$ (*STATE* $\leftrightarrow$ *STATE*)

$\forall p : \text{SDLProcess} \bullet$

next *p* = { *i* : *INPUTS*; *o* : *OUTPUTS* | *i* ↦ *o* ∈ *body p* • *state i* ↦ *state o* }

and then

reachable : *SDLProcess* $\leftrightarrow$ $\mathbb{P}$ *STATE*

$\forall p : \text{SDLProcess} \bullet$

reachable p = (*next p*)^{*} ((*state* (*initial p*)))

Models of designs in SDL-10 will consist of multiple, cooperating processes: the inputs of some processes may be drawn from the outputs of others, and one process may *activate* another. We extend our notion of tasks accordingly:

TASK ::= *null* | *stop* | *activate*(⟨⟨*OUTPUTS*⟩⟩)

and define

activations : *TASK* $\rightarrow$ $\mathbb{P}$ *OUTPUTS*

activations = *activate*[~] ∪ { *null* ↦ ∅, *stop* ↦ ∅ }

Within a multiple-process model, each state name must be unique to a single process, and each signal name may appear as input in at most one process. A process may activate others by referring to states within their bodies. A process that is activated by another will have an empty initialisation.

The set of input and output signals of a process may be determined by considering, respectively, the inputs in the body, and the range of trace segments:

$inputSignals, outputSignals : SDLProcess \rightarrow \mathbb{P} \text{ SIGNAL}$
$inputSignals = (ran) \circ inputs$
$(\forall p : SDLProcess \bullet$
$outputSignals\ p = \bigcup \{o : outputs\ p \bullet$
$ran(segment\ o) \cup$
$\bigcup \{o' : activations\ (task\ o) \bullet ran(segment\ o')\} \}$

We may then define

$SDLProcesses : \mathbb{P}(\mathbb{P} \text{ SDLProcess})$
$\forall ps : SDLProcesses \bullet$
$(\forall p1, p2 : ps \mid p1 \neq p2 \bullet$
$states\ p1 \cap states\ p2 = \emptyset \wedge$
$ran(inputs\ p1) \cap ran(inputs\ p2) = \emptyset \wedge$
$state \sqcap \bigcup (activations \sqcap (task \sqcap outputs\ p1 \sqcap)) \sqcap states\ p2 \neq \emptyset \Rightarrow initial\ p2 = \emptyset)$

For a given collection of processes, we may determine the set of all internal signals:

$internalSignals : SDLProcesses \rightarrow \mathbb{P} \text{ SIGNAL}$
$externalSignals : SDLProcesses \rightarrow \mathbb{P} \text{ SIGNAL}$
$\forall ps : SDLProcesses \bullet$
$internalSignals\ ps = \bigcup (inputSignals \sqcap ps) \cap \bigcup (outputSignals \sqcap ps) \wedge$
$externalSignals\ ps =$
$\bigcup ((inputSignals \sqcap ps) \cup outputSignals \sqcap ps) \setminus internalSignals\ ps$

We may define also the closure of a collection, in which every process may accept but ignore any signal for which it has no corresponding output.

$_ \# : SDLProcesses \rightarrow SDLProcesses$
$\forall ps : SDLProcesses \bullet$
$ps \# = \{p : ps \bullet p \flat\}$

III.3 Behaviours

For the purposes of specification and comparison, we wish to consider only the notionally observable behaviour of a collection of SDL-10 processes: that is, the sequences of inputs and outputs that may be recorded. To do this, we must consider, at each stage, the current state of each process in our collection, the contents of the (single) signal buffer, and the history of signals observed thus far. We require an additional component, *tasks*, to identify any outstanding tasks (which would add to the output signals without any additional input).

$ModelState$
$processes : SDLProcesses$
$currentState : SDLProcess \rightarrow STATE$
$bufferState : seq \text{ SIGNAL}$
$history : seq \text{ SIGNAL}$
$tasks : seq \text{ TASK}$
$\text{dom } currentState \subseteq processes$

Processes that have yet to be activated are not associated with any current state.

The initial configuration for the model depends upon the set of SDL processes whose future behaviour we wish to explore:

<i>ModelInit</i>
<i>ModelState'</i>
<i>processes?</i> : <i>SDLProcesses</i>
<i>currentState?</i> : <i>SDLProcess</i> $\leftrightarrow$ <i>STATE</i>
<i>processes'</i> = <i>processes?</i>
<i>currentState'</i> = <i>currentState?</i>
<i>bufferState'</i> = $\langle \rangle$
<i>history'</i> = $\langle \rangle$
<i>tasks'</i> = $\langle \rangle$

Furthermore, we will assume that inputs may be added only when the collection of processes is in a quiescent state: that is, every sequence of outputs currently underway has been completed, and every internal signal generated has been added to the buffer.

<i>InputSignal</i>
Δ <i>ModelState</i>
<i>in?</i> : <i>SIGNAL</i>
<i>processes'</i> = <i>processes</i>
<i>bufferState</i> = $\langle \rangle$
<i>bufferState'</i> = $\langle in? \rangle$
<i>processState'</i> = <i>processState</i>
<i>history'</i> = <i>history</i>
<i>tasks'</i> = <i>tasks</i>

When a signal is processed, a process may change state, and a sequence of output signals may be produced. A signal may be processed only when there are no outstanding tasks or activations. If the change in state is associated with an activation task, then further sequences of signals may be produced, and other processes—those with states named in the activation set—may change state.

ProcessSignal

$\Delta \text{ModelState}$

$\text{signal} : \text{SIGNAL}$

$\text{outputs!} : \text{seq SIGNAL}$

$\text{activationSequences} : \text{seq}(\text{seq SIGNAL})$

$\text{activationStates} : \mathbb{P} \text{ STATE}$

$\text{target} : \text{SDLProcess}$

$\text{localOutput} : \text{OUTPUTS}$

$\text{tasks} = \emptyset$

$\text{tasks}' = \text{tasks} \cup \{\text{task localOutput}\}$

$\text{bufferState} \neq \langle \rangle$

$\text{head bufferState} = \text{signal}$

$\text{bufferState}' = \text{tail bufferState} \hat{\ } (\text{outputs!} \upharpoonright \text{internalSignals processes})$

$\text{history}' =$

$\text{history} \hat{\ } \langle \text{signal} \rangle \hat{\ } (\text{outputs!} \upharpoonright \text{externalSignals processes})$

$\text{target} \in \text{processes}$

$(\text{currentState target}, \text{signal}) \mapsto \text{localOutput} \in \text{body target}$

$\text{outputs!} = (\text{segment localOutput}) \hat{\ } (\cap / \text{activationSequences})$

$\text{ran activationSequences} = \text{segment} \langle \text{activations}(\text{task localOutput}) \rangle$

$\text{processes}' = \text{processes}$

$\text{activationStates} = \text{state} \langle \text{activations}(\text{task localOutput}) \rangle$

$\text{currentState}' = \text{currentState} \oplus (\text{processes} \triangleleft \text{isStateOf} \sim) \triangleright \text{activationStates}$

The finite traces of a combination of SDL-10 processes can be generated by repeated application of the functions corresponding to these two operation schema's. If we define the schema *step*, which provides a single state transition:

$$\text{Step} \hat{=} (\neg \text{pre ProcessSignal} \wedge \text{InputSignal}) \vee \text{ProcessSignal}$$

then we may consider the set of all states (of collections of processes) that arise from repeated applications of this operation, starting from the (empty) initial state.

$\text{nextState} : \text{ModelState} \leftrightarrow \text{ModelState}$

$\forall \Delta \text{ModelState} \bullet$

$\emptyset \text{ModelState} \mapsto \emptyset \text{ModelState}' \in \text{nextState} \Leftrightarrow \text{Step}$

$$\text{ReachableStates} == \text{nextState}^* \langle \text{ModelInit} \rangle$$

Provided that a combination of processes includes no loops of activation or internal signalling, the effect of any input is determined entirely by sequences of output signals associated with either *null* or *stop*. The behaviour that interests us is recorded in the *history* component of *ModelState*:

$\text{traces} : \text{SDLProcesses} \leftrightarrow \mathbb{P}(\text{seq SIGNAL})$

$\forall \text{ps} : \text{SDLProcesses} \bullet$

$\text{traces ps} = \{s : \text{ReachableStates} \mid \text{processes?} = \text{ps} \bullet \text{history}\}$

Given our trace semantics, we can define a corresponding notion of refinement between collections of SDL processes:

$$\begin{array}{|l} \hline _ \sqsubseteq _ : \text{SDLProcesses} \leftrightarrow \text{SDLProcesses} \\ \hline \forall ps, qs : \text{SDLProcesses} \bullet \\ \quad ps \sqsubseteq qs \Leftrightarrow \text{traces } qs \sqsubseteq \text{traces } ps \end{array}$$

For the comparison of SDL-10 models, this means that for every signal sequence and signal input to be found in the implementation the same must be found in the specification. The outputs of the signal entry must be the same as or less non-deterministic in the implementation than the specification.

This can be demonstrated by a refinement of the previous example.

Process RefinedExampleProcess ;

```
start;
output OUTPUTA ;
nextstate STATEA ;

state STATEB ;
  input INPUTC ;
  output OUTPUTC ;
  STOP;
  input INPUTA ;
  nextstate STATEA ;
endstate;
```

```
state STATEA ;
  input INPUTB ;
  output OUTPUTA ;
  nextstate STATEB ;
  input INPUTA ;
  nextstate STATEA ;
endstate;
```

Endprocess RefinedExampleProcess;

This process is shown in graphical form in Figure III.2.

For completeness the SDL-10 representation is detailed.

$$\begin{array}{|l} \hline \text{RefinedExampleProcess} : \text{SDLProcess} \\ \hline \text{RefinedExampleProcess} = \\ \quad (\{ (STATEA, (\langle OUTPUTA \rangle, null)) \}, \\ \quad \{ ((STATEA, INPUTA) \mapsto (STATEA, (\langle \rangle, null))), \\ \quad ((STATEA, INPUTB) \mapsto (STATEB, (\langle OUTPUTA \rangle, null))), \\ \quad ((STATEB, INPUTA) \mapsto (STATEA, (\langle \rangle, null))), \\ \quad ((STATEB, INPUTC) \mapsto (dud, (\langle OUTPUTC \rangle, stop))) \}) \text{ } \flat \end{array}$$

So in terms of refinement the following will hold.

$$\text{RefinedExampleProcess} \sqsubseteq \text{ExampleProcess}$$

The availability of state information in our defining function means that we are not restricted to the notion of traces refinement defined for CSP [5]: we could address also the successful/unsuccessful nature of each trace in terms of the resulting task value. However, if we are working at the level of a concrete design, where the intention is that the specification should entirely determine the response of the system to every signal of interest, a notion of traces refinement will suffice.

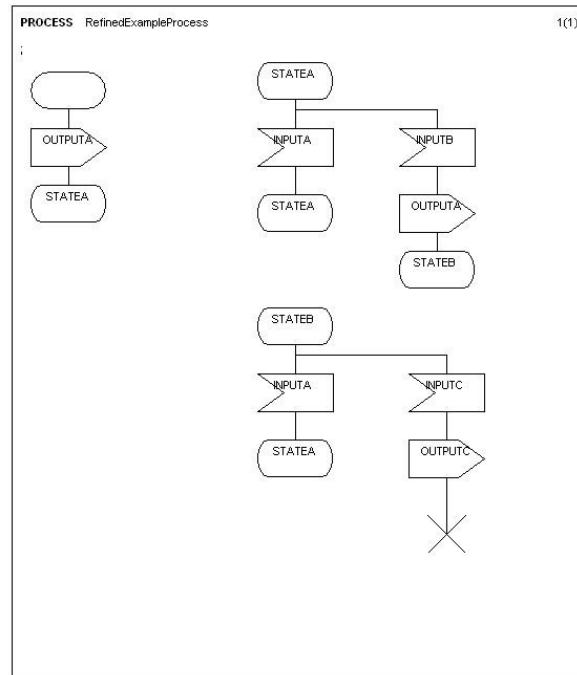


Figure III.2: A refinement of ExampleProcess process

III.4 Application

SDL-10 models within Ericsson are validated using the SDLtool, based on the semantics of plain SDL, rather than SDL-10. The SDL-10 semantics developed here has been used to characterise situations in which the SDLtool would not properly represent the behaviour of the target AXE-10 execution environment. This had an impact on the development processes: although no change was made to the SDL toolsets, due to the high cost of doing so, processes were put in place to identify situations where a difference in behaviour would occur, and to ensure that the engineered solution would cope.

Another use of the semantics would be as the basis of a simulator, as a guide to implementation or for checking that one SDL-10 process would be a suitable replacement for another. Of these, the first would seem the most likely to provide an some return on effort invested. Nevertheless, to go from concept to working model requires a considerable amount of work from the modeller. System, block, and process diagrams have to be created, signals have to be declared and aggregated into lists, and more besides.

This leads us, in the next chapter, to propose a ‘shorthand version’ of the language, removing the system and block diagrams while retaining the essential behavioural information. Along with self-declaring signal definitions, this shorthand version of the language will allow rapid prototyping for proof of concept, and if combined with tool support can be used more easily to generate an SDL model. Generation, and re-generation of models avoids the problems of trying to modify complete models where the structure—even with object-oriented features—may be fragile and components may be difficult to re-use.

To increase the value and utility of the shorthand version, and the calculus we develop around it, we will address the full SDL language, rather than just the SDL-10 subset.

IV

A Calculus of Buffered Systems

Our ‘shorthand version’ of SDL will be presented as a calculus: the ‘Calculus of Buffered Systems’ or CBS. This language will be given a semantics intended to facilitate compilation and model generation, based upon Hoare and He’s Unifying Theories of Computation. This semantics is intended to be extensible to the whole of SDL, although this would be a subject for further research.

IV.1 Notation

We will use a notation similar to that of CSP, but with modifications to the notion (and the representation) of event prefix. We will write $\rightarrow$ to denote the standard event prefix for processes: that is, for event a and process P , the term $a \rightarrow P$ will denote a process that is ready to perform a and—should a be performed, will then behave as P . $\leadsto$ is the standard input prefix of Hoare and He [7], and $\hookrightarrow$ is a variant of this—defined below—more appropriate to SDL. $\gg$ is used to denote a generalisation of the piping operator of CSP, to represent a delaying channel composition. Finally, *KILL* is used instead of *skip* to represent process termination, emphasising the difference between the distributed, consensual termination of CSP and the more active form found in SDL.

IV.2 Input and output

SDL models are grouped into *blocks*, which may contain further blocks as well as processes. Blocks make no direct contribution to behaviour, but affect the interpretation of communication expressions [4, p.39].

Hoare and He assume that the order in which an event is processed is non-deterministic if the event is input on separate channels to the same process [7, p.224, L11]. In SDL, this is not always the case. There are two alternatives [15, p.238].

1. If the message is sent to a process within the same block the message is assumed to enter the input queue of the receiving process immediately; there will be no nondeterminism.
2. If the message is sent to a process of a separate block the message is to pass through a delaying channel; here, there will be a non-deterministic delay.

The definition for the input prefix is given in [7, p.224]:

$$a?x \leadsto Q(x) = IN \gg (a?x \rightarrow Q(x))$$

(IV.1)

and we must add to this the following law [7, p.224, L11]:

$$a?x \rightsquigarrow (b?y \rightsquigarrow P(x, y)) = b?y \rightsquigarrow (a?x \rightsquigarrow P(x, y)), \quad \text{if } a \neq b$$

A separate input buffer is applied to each input channel and it is therefore not possible to determine which input channel is serviced first.

This form of input prefix is not suitable for modelling process-to-process communication within an SDL block, as communication is considered immediate without any delay. Instead, we require a buffer that can handle multiple channels for input, this varies from the Hoare and He version [7, p.219]: $BUF^{cbs}_{A:\{b\}}$ aggregates several input channels into one queue, which can then be chained to another process.

$$\begin{aligned} inchan(BUF^{cbs}_{A:\{b\}}) &=_{df} A \\ outchan(BUF^{cbs}_{A:\{b\}}) &=_{df} \{b\} \\ BUF^{cbs}_{A:\{b\}} &= BUF^{cbs}_{A:\{b\}}(<>) \end{aligned}$$

where

$$\begin{aligned} BUF^{cbs}_{A:\{b\}}(<>) &= (s : A \bullet s?x \rightarrow BUF^{cbs}_{A:\{b\}}(<x>)) \\ &\quad | b?kill \rightarrow SKIP \\ BUF^{cbs}_{A:\{b\}}(<x>\hat{\sim}s) &= (s : A \bullet s?y \rightarrow BUF^{cbs}_{A:\{b\}}(<x>\hat{\sim}s\hat{\sim}<y>)) \\ &\quad | b!x \rightarrow BUF^{cbs}_{A:\{b\}}(<s>) \\ &\quad | b?kill \rightarrow SKIP \end{aligned}$$

This is a variation of the definition given for BUF in Hoare and He: the *inchan* and *outchan* definitions are theirs [7, p.218], and can be used with the other data flow definitions and constructs. All output will be directed to a single output channel. The special event *kill* is used when terminating the buffer, and can be received from an outgoing channel. Following Hoare and He's specialisation to IN [7, p.221], we can define IN^{cbs} , and hence a new and more appropriate input prefix, $\hookrightarrow$, for describing SDL inputs

$$a?x \hookrightarrow Q(x) = IN^{cbs} \gg (a?x \rightarrow Q(x))$$

Hoare and He define a notion of *data flow* for CSP: a data flow network is one in which the channels are capable of buffering an arbitrary number of messages [7, p.217]. They define a closure operator $[]$ which composes a process with families of buffers:

$$[P] = IN \gg P \gg OUT$$

where IN and OUT are parallel combinations of buffers, with one infinite buffer for each input or output channel. A process P then has the data flow property if and only if $P = [P]$ (Hoare and He refer to this as the 'foam rubber wrapper postulate'.)

A simple modification of this notion can be used to characterise *SDL-like* processes, and hence to help define the semantics of the SDL language. We define

$$[P] = IN^{cbs} \gg P$$

and say that P is *SDL-like* if and only if $P = [P]$.

It is often necessary to consume signals in a different order than the order of arrival of signals in the input queue [15, p.75]. SDL provides a construct, *save*, for this purpose. To model this, we may extend the input prefix with a subscript:

$$a?x_s \hookrightarrow Q(x) = (IN' \triangleright s) \gg (a?x \rightarrow Q(x))$$

For a particular input the events to be saved are removed, temporarily, from the buffer.

Hoare and He define also an output prefix [7, p.223]: for the purposes of modelling SDL, this is redundant, and the conventional CSP $\rightarrow$ prefix operate will suffice. However, where the output is destined for another block or for the environment, we must allow also for a non-deterministic delay: the ‘foam rubber wrapper’ closure operator, $[]$, is exactly what is required for this purpose.

IV.3 Termination

The obvious approach to termination would be to re-use the CSP notions of *STOP* and *SKIP*, as in [6, p.25], but this doesn’t adequately represent the input behaviour. Using Hoare and He’s semantics, using either process leaves us with a composition that behaves as *SINK*, accepting any further input, but providing no output. Instead, what we would wish is to have a signalled termination (as in *SKIP* for CSP). The buffer itself must be terminated, which we can do using our special *kill* event, defining a new process expression:

$$a?x \hookrightarrow KILL = IN^{cbs} \gg (a?x \rightarrow a!kill \rightarrow SKIP)$$

The buffer chained to the process will now synchronise on the *kill* event and terminate as required.

IV.4 Composition

The process composition provided by Hoare and He, $\gg$, needs to be extended to cope with the multiple chains present in SDL models. We need to extend it to produce a symmetric, generalised parallel operator, akin to the generalised piping of machine-readable CSP:

$$p^{composed} == p^{from} \leftrightarrow p^{to}$$

$$\forall P : p^{composed} \bullet$$

$$IN \notin \text{ran}(P)$$

$$\wedge$$

$$OUT \notin \text{dom}(P)$$

where *IN* and *OUT* are the infinite buffers of Hoare and He, representing foam rubber insulation from the environment. If communication with the environment is required, then a composed process will need to include *IN* and/or *OUT*.

Managing without such a generalisation would be impractical,

$$\begin{array}{c} B \\ \nearrow \searrow \\ IN \gg A \ll C \gg OUT \end{array}$$

and yet a form of the composition written in extension would be useful: we write

$$P = \{IN \mapsto A, A \mapsto B, B \mapsto C, C \mapsto A, C \mapsto OUT\}$$

as a shorthand for the composition illustrated above.

IV.5 SDL Models

In SDL, communication is based on communication primitives called **signals**: atomic communications between the system and its environment, or between different parts of the system. [15, p.60]. Communication using signals is via a `signalroute`.

For example, suppose that a signal *launch* is to be sent from one process *P* to another process *Q* via signalroute *fasttrack*. The sending expression

```
output launch via fasttrack
```

has the semantics

$$fasttrack!launch \rightarrow P$$

and the receiving expression

```
input launch;
```

is modelled as

$$fasttrack?launch \hookrightarrow Q$$

An SDL model does not need to declare an explicit `signalroute` for every output statement: the dynamic semantics of a signal route are described through implicit addressing [15, p.68] and therefore can be determined when translating to CBS.

An SDL process is an extended finite state machine, expressed in terms of states and transitions, using signals for communication. We can use the SDL state together with a unique (system-wide) identifier to represent the process state in the semantics. The constructs *KILL*, $\hookrightarrow$, and $\rightarrow$ (including the extended notion of input saving) will then be enough to represent any SDL process in a shorthand fashion.

We have, however, to consider the block constructs used to organise a specification, as they have an influence upon signal communication [15, p.38]. Where a block consists of a single process, the data flow definition of Hoare and He [7, p.221] will suffice:

```
block ExampleBlock;
  process P referenced;
  signalroute FromEnv from env to P with ExampleSignals;
  signalroute ToEnv from P to env with ExampleSignals;
endblock ExampleBlock
```

can be modelled as

$$[P] = IN \gg P' \gg OUT$$

As all communication is buffered, channel by channel, this provides for non-deterministic signalling.

For simple processes with unidirectional communication, without loops, we might also use the Hoare and He chaining operator:

```
block ExampleBlock;
  process P referenced;
  process Q referenced;
  signalroute ExamplePath from P to Q with ExampleSignals;
  signalroute FromEnv from env to P with ExampleSignals;
  signalroute ToEnv from Q to env with ExampleSignals;
endblock ExampleBlock
```

can be modelled as:

$$P \gg Q$$

In more complex models, the chaining operator will not work, and we need to adopt the generalised composition:

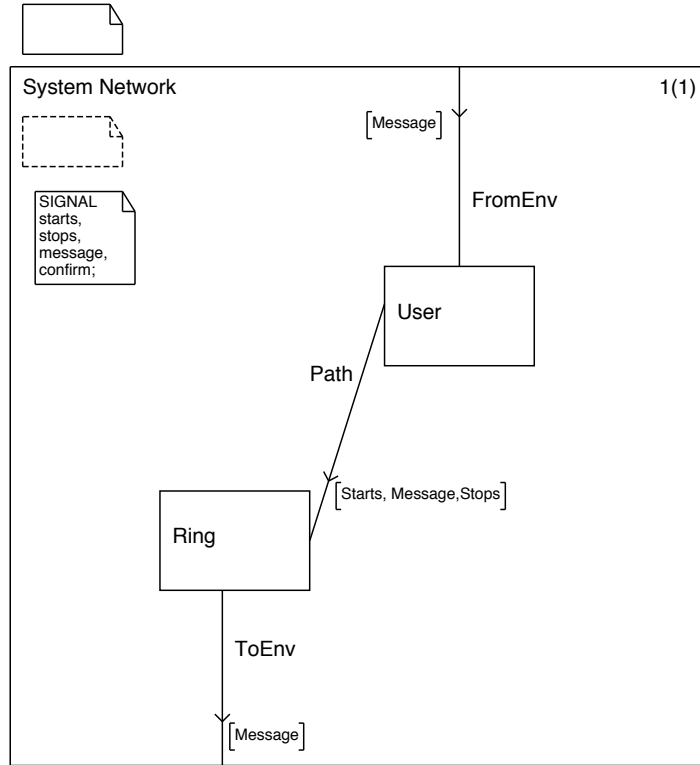


Figure IV.1: SDL System Network

```

block ExampleBlock;
  process A referenced;
  process B referenced;
  process C referenced;
  signalroute BottomPath from C to A with ExampleSignals;
  signalroute LeftPath from A to B with ExampleSignals;
  signalroute RightPath from B to C with ExampleSignals;
  signalroute FromEnv from env to A with ExampleSignals;
  signalroute ToEnv from C to env with ExampleSignals;
endblock ExampleBlock

```

can be modelled as

$$P = \{IN \mapsto A, A \mapsto B, B \mapsto C, C \mapsto A, C \mapsto OUT\}$$

Finally, system is the top-level declaration in SDL, enclosing the blocks of the model. The same notion of process composition can be applied here.

IV.6 Example

The system Network—see Figure IV.1—is composed of two blocks, User and Ring. The block User receives messages from the environment and sends them to the Ring which returns them to the environment. User is composed of one process, SingleUser—see Figure IV.2—, which segments the messages and starts and stops communication with the Ring. The Ring is composed of three process chained in a circle, and can only handle one message transfer at a time.

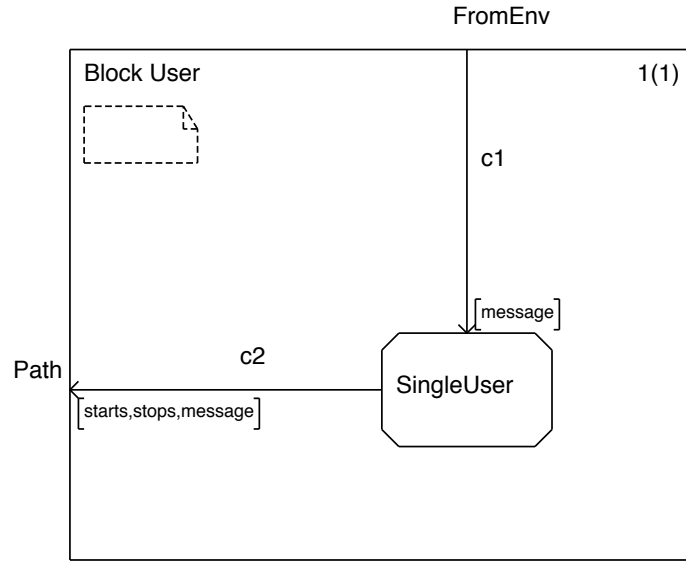


Figure IV.2: SDL block *SingleUser*

We can model this as $Network = User \gg Ring$ where $User = [SingleUser]$ and

$$SingleUser = FromEnv?message \hookrightarrow Path!starts \rightarrow$$

$$Path!message \rightarrow Path!stops \rightarrow SingleUser$$

as in Figure IV.3.

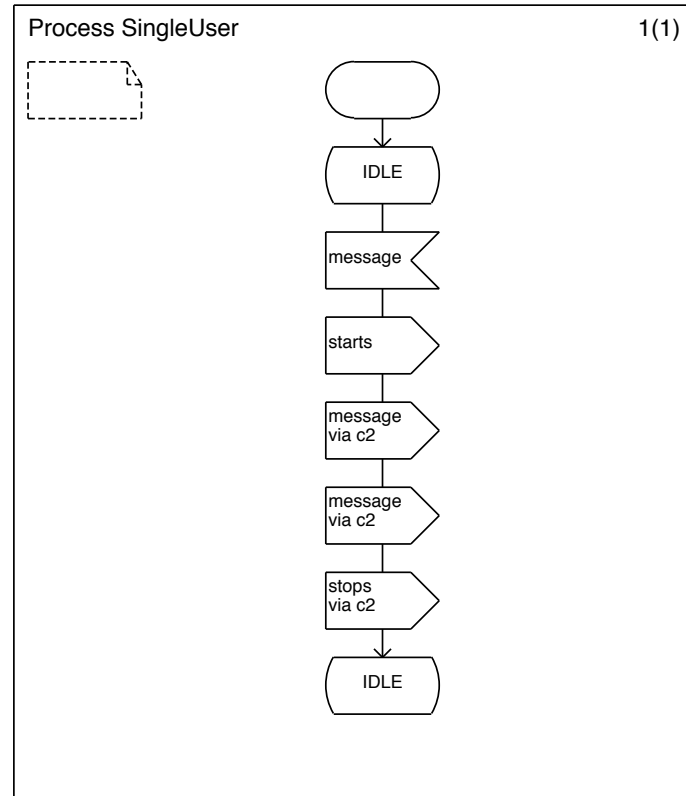


Figure IV.3: SDL Process *SingleUser*

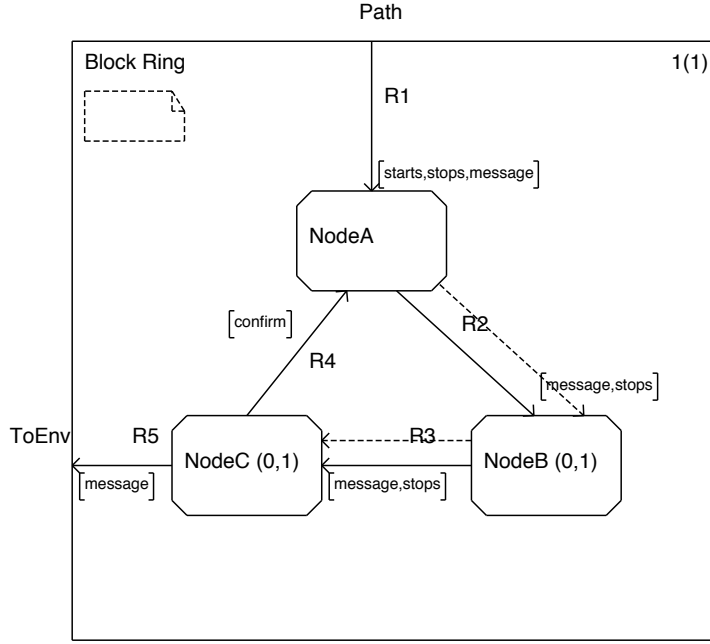


Figure IV.4: SDL Block Ring

The block Ring shown in Figure IV.4 can be described as a parallel composition:

$$Ring = \{IN \mapsto NodeA, NodeA \mapsto NodeB, NodeB \mapsto NodeC, NodeC \mapsto NodeA, NodeC \mapsto OUT\}$$

and the process for NodeA as:

$$\begin{aligned} NodeA &= Path?starts \mapsto (ACTIVATED_A \parallel NodeB) \\ ACTIVATED_A &= Path?message \mapsto (R2!message \mapsto TRANSMIT_A) \\ &\quad | Path?stops \mapsto (R2!stops \mapsto NodeA) \\ TRANSMIT_A_{\{message,stops\}} &= R4?confirm \mapsto ACTIVATED_A \end{aligned}$$

The process for NodeB within the same block can be modelled as:

$$\begin{aligned} NodeB &= NodeC \parallel ACTIVATED_B \\ ACTIVATED_B &= R2?message \mapsto (R3!message \mapsto ACTIVATED_B) \\ &\quad | R2?stops \mapsto (R3!stops \mapsto KILL) \end{aligned}$$

where

$$\begin{aligned} NodeC &= R3?message \mapsto (ToEnv!message \mapsto R4!confirm \mapsto NodeC) \\ &\quad | R3?stops \mapsto KILL \end{aligned}$$

describes the process of NodeC. These processes are illustrated in Figures IV.5, IV.6 and IV.7.

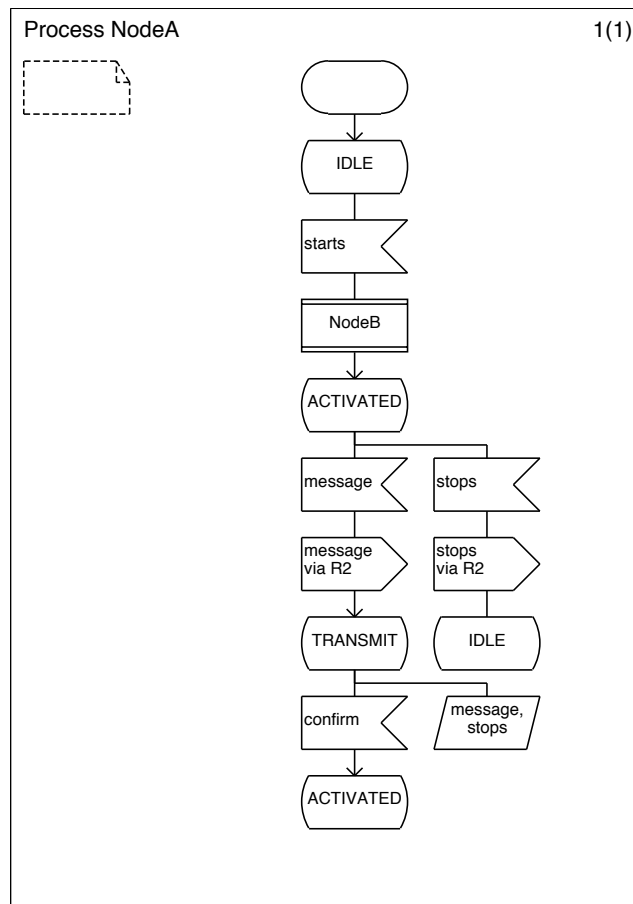


Figure IV.5: SDL Process NodeA

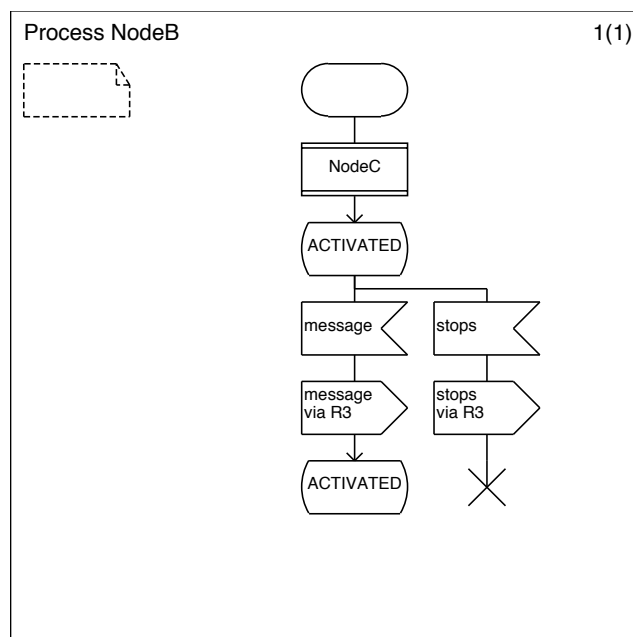


Figure IV.6: SDL Process NodeB

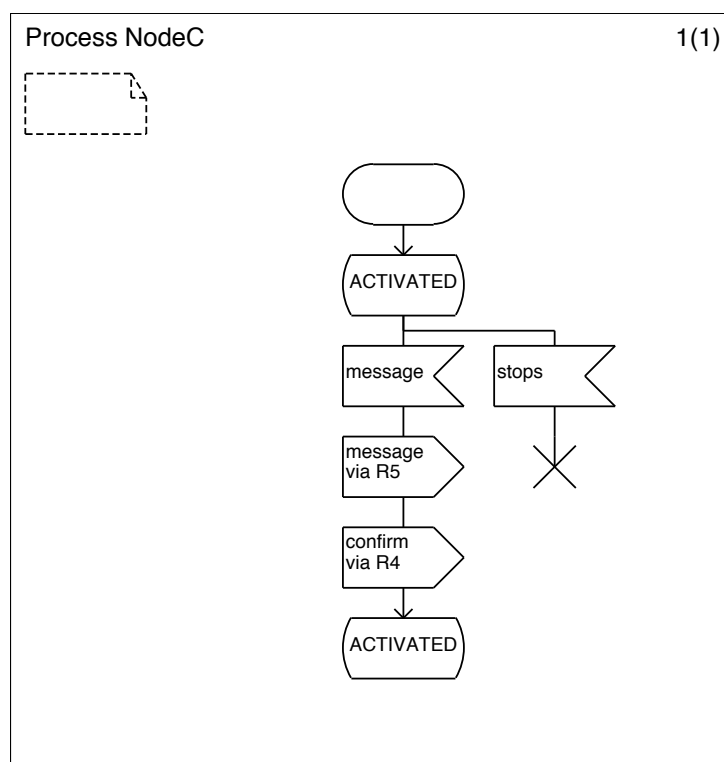


Figure IV.7: SDL Process NodeC

V

Case Study

We can now demonstrate the intended usage of the semantics with a simple case study, using CBS as a basis for translation into UML and then implementation into Java. We wish to explore the use of the semantics in relating high level models to target executables, and to illustrate differences between the SDL semantics and the behaviour of SDL-10 implementations. Furthermore, we will explore the application of these techniques in the context of UML-based development, and the construction of a candidate, generic Java framework for implementing designs expressed in SDL.

An additional aspect of the framework will be candidate support for multi-process interaction in a single-threaded or resource-limited environment, such as that found in many mobile devices. The subject of the case study is a multi-player game, implemented using Java Microedition, on a mobile phone.

V.1 SDL into UML

An SDL model can be represented first as a set of (mostly abstract) classes assembled into a package. We can then use the semantics to guide specialisation, in the ordinary sense of UML, rather than as a basis for direct translation from one language to another—the approach taken to unification of UML and SDL in the standardisation activity [14].

Classes for `process`, `signal`, `state`, `machine`, `system`, and `environment` (as an interface) are provided. This covers the core of the language, and will suffice for the demonstration at hand. The UML required is shown in Figure V.1.

This can be used as a development framework, allowing the user to remain within the SDL paradigm or shift incrementally into a wider OO context. Communication between the processes can be via signal sending and receiving, brokered by `machine`. The developer can use direct method calls or remote operations if desired, but needs to understand the implications of this in terms of real time behaviour. Additional objects can be included within the abstracted objects, and new data types can be declared within the model. However, the more the developer adds to the framework, the harder it will be to reverse engineer the design into pure SDL should that be required later.

The class `machine` represents the imaginary SDL machine that manages the communication between processes and/or the environment. The implementation of this class depends upon the target environment used for the actual implementation of the model. If the modeller intends on keeping the model true to the semantics of SDL all communication between processes and/or the environment will have to be through the machine object.

This class is abstract, and makes the following static operations available:

- `addProcess`, used to add a process to the machine, during system initialisation and process

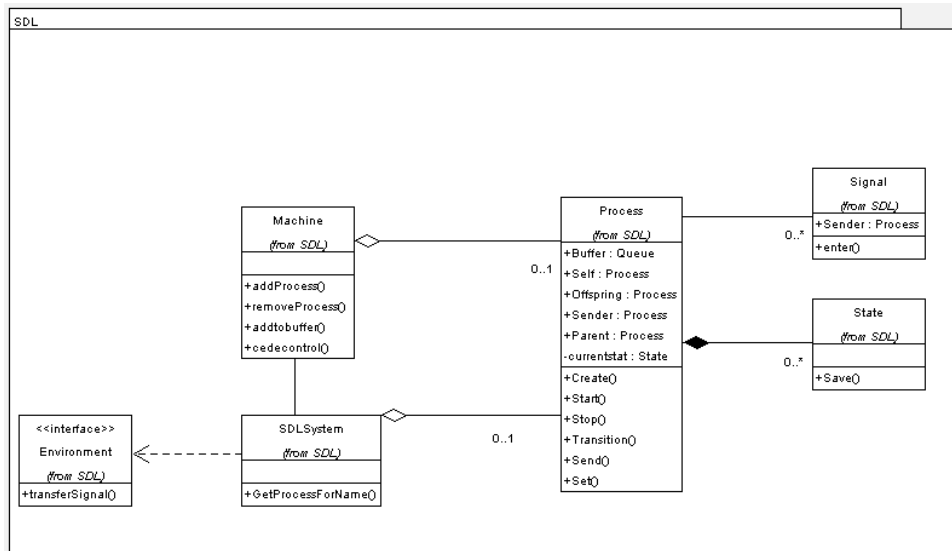


Figure V.1: SDL in UML

activation; provides a reference for subsequent communication.

- **removeProcess**, used to remove a process from the machine, following a **stop** instruction in SDL; the process is removed and can no longer be presented as a destination for signals.
- **addtobuffer**, used when sending a signal, which is added to the end of a queue or sent to the environment—for the latter, the receiving class must implement the interface **environment**.
- **cedecontrol**, required to interwork with the thread control in the deployed system. When this operation is called, a thread is initiated and returns at the end of operation call. The implementation can either re-call the operation immediately, or wait to give another operation a turn. This is true to a certain implementation of the non-deterministic SDL machine, but we will need to take care to ensure proper utilisation in a multi-processor environment.

In declaring an SDL system, the initial instantiations of the processes must be determined, in order to bootstrap the imaginary SDL machine. Class **SDLSystem** is used to reference the processes that are to be activated at initialization. It provides also a facility for the environment to identify processes by name, and hence send a signal into the implementation system. The **environment** interface contains a single abstract operation, **transferSignal**, which must be implemented if the signal is to have a destination.

Process is an abstract class with attributes and operations corresponding to the features of an SDL process. Every process will have a buffer, a queue for handling signals that allows signals to be saved or expedited. **Self**, **Offspring**, **Sender** and **Parent** represent processes following the naming convention used in the SDL language. Any implementation of this model must set these to appropriate process references.

A process could have no state, simply starting, performing an action, and then stopping. Alternatively, and more usually, it will have a state described as a specialisation of the class **state**:

- attribute **currentstate** refers to the state class currently available—there can be at most one of these.
- **create** is an operation used to create another process, and will set offspring and parent attributes appropriately.
- **start** is called by the SDL machine when the process is instantiated, and corresponds to the start symbol in SDL.

- `stop` is used to terminate a process, and destroy its reference.
- `transition` takes a state as a parameter, and is used to move to the next state: a new state reference is passed, and `currentstate` is updated.
- `send` is used to send a signal from one process to another, or to the environment. Signal sending is implemented by transferring signal objects; signal object and destination are passed as parameters.
- `set` is used to start timers, and takes the timeout name and duration as parameters.

Communication between processes is asynchronous, managed by the broker class `machine`, and implemented using signals. The `signal` class is abstract, and requires specialisation for each and every signal. The parameters are attributes of the class: the `sender` attribute is assigned when the signal is sent, and used to populate the sender reference in the destination process. The `enter` operation is called by the receiving process, and calls an operation with the same name as the signal, allowing a direct correspondence with SDL signal entries.

Only one `state` object can be active at any one time. Within the `state`, each signal entry is represented by an operation—something that should be familiar to users of the SDL language, in which signal entry initiates the behaviour known as ‘logic processing’. The operation `save` allows for signal entry deferral by parameter name. As each state has its own class, the same operation call, or signal entry, can represent different functionality.

V.2 Example: a game application

We may construct a corresponding Java framework, targeted at—for the sake of a suitable demonstration—an embedded device, such as a mobile phone platform. The standard Java 2 Micro Edition, J2ME, will suffice [25]. The framework that we define will be usable for any SDL model, including those which specify concurrent behaviour.

The game begins by asking for the number of players: up to four may play concurrently, using the four sides of the keypad, provided that their fingers are sufficiently small and nimble. The players navigate lines representing *snakes* across the screen, the idea being that the lines should not coincide. Players drop out as they hit other player’s snakes, and the last one left is the winner: see Figure V.2.

We can create an SDL model for the game, represent it in the UML framework, specialise it, and then implement it in Java. We can use CBS as a shorthand notation to define our model:

$$\begin{aligned}
 &Snakefactory = IDLE_{-1} \\
 &IDLE_n = \\
 &\quad env?Createsnake \hookrightarrow (Snake \parallel (env!Snakecreated \rightarrow IDLE_{n+1})) \\
 &\quad | \\
 &\quad SC?Died \hookrightarrow if((n - 1) = 0)then(env!Winner \rightarrow IDLE_{n-1})else(IDLE_{n-1})
 \end{aligned}$$

where

$$\begin{aligned}
 &Snakepart = \quad SetTimeout \rightarrow Command \\
 &Command = \\
 &\quad env?Left \hookrightarrow Command \\
 &\quad | \\
 &\quad env?right \hookrightarrow Command
 \end{aligned}$$



Figure V.2: Game: starting, running, and finishing

```

|
Timeout → MC!UseElement → Check
Check =
  MC?OKLeft,Right ↪ env!Move → SetTimeout → Command
|
  MC?NotOKLeft,Right ↪ SC!Died → KILL
Timer = Settimeout → Timeout → Timer
Snake = (Snakepart || Timer)

```

We define also

```

Movecontroller = Activated
Activated =
  MC?UseElement ↪
    (MC!OK → Activated)
  □
  (MC!NotOK → Activated)

```

These processes are then composed to produce a complete model:

```
Supersnake = (Snakefactory || Movecontroller)
```

The SDL model thus defined contains a single block, shown in Figure V.3. This block contains a set of three processes, which communicate amongst themselves and with the environment: Snakefactory, Snake and Movecontroller. There will only be one instance of Snakefactory, used to initialise and manage the instances of Snake, and to identify the winning player. There will be one instance of Snake per player initially, decreasing as the game progresses.

When a snake process is ready to move, it checks on the state of the next screen pixel by contacting the process Movecontroller, which contains a bit map representation of the screen. This map indicates whether a position has already been used, and hence allows the Movecontroller to

determine the validity of a move. The SDL diagrams for the model are shown in Figures V.3 to V.9, following.

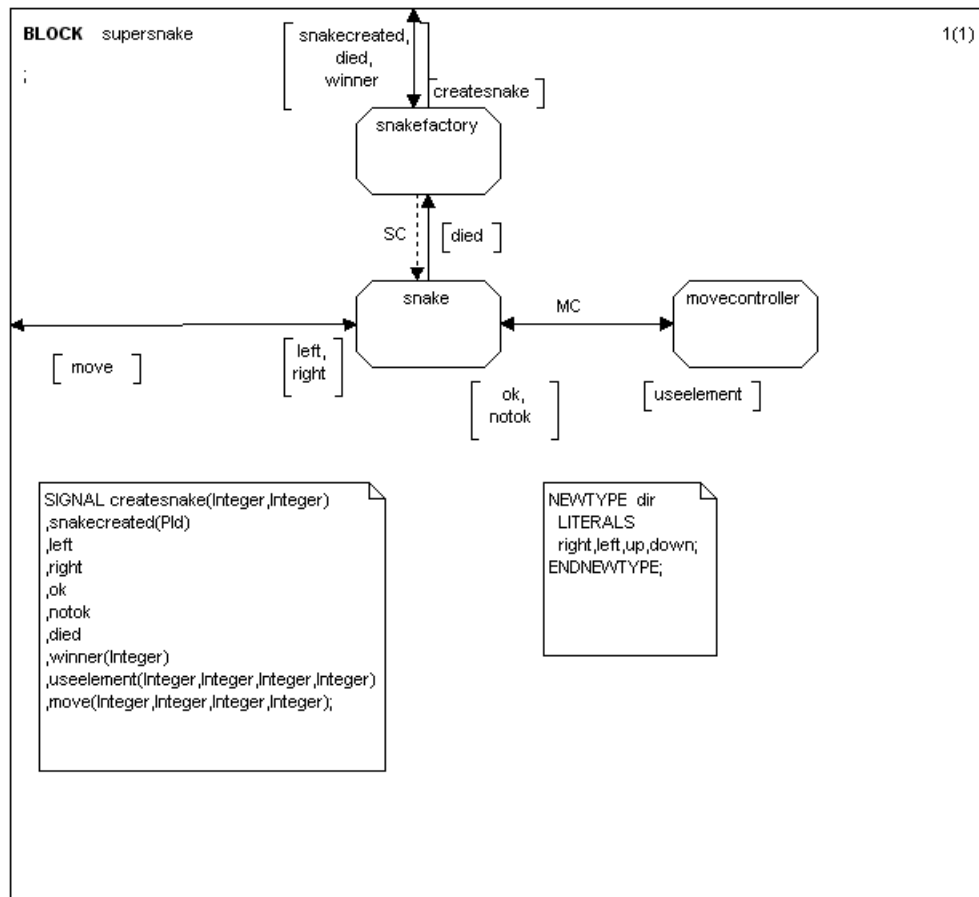


Figure V.3: SuperSnake block

The block SuperSnake contains the processes Snake, Snakefactory and Movecontroller and their channels of interaction. The process Snakefactory is used to process requests to create one or more Snake processes. The first page of process Snake which is where the process is initialised. The second page of process Snake which is where the requests to move are received from the user. The third page of process Snake which is where to process the timeout which initiates a request to move. The fourth page of process Snake which is where to process response to the request to move. The process Movecontroller is used to process requests to move to a position for a particular snake.

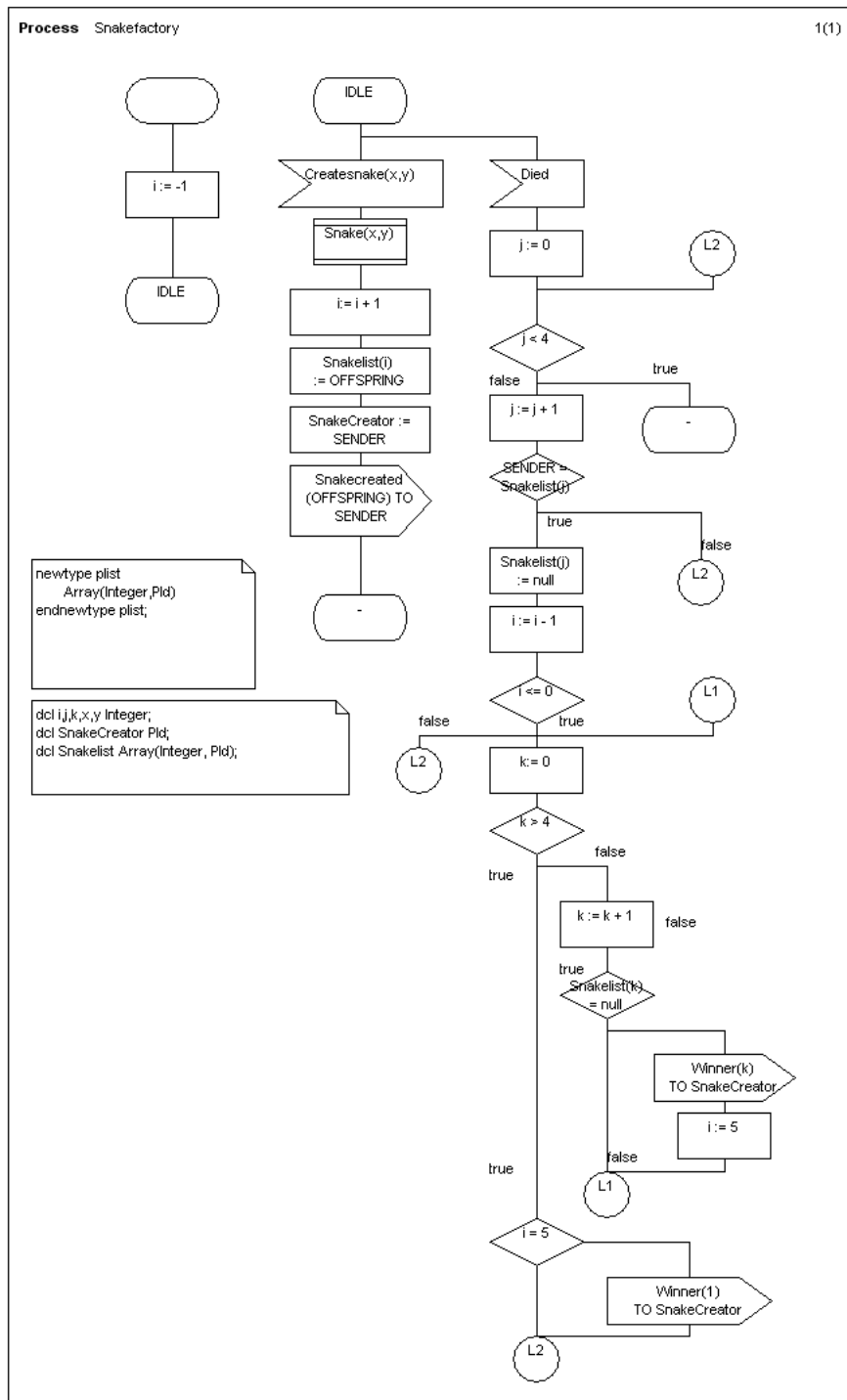


Figure V.4: SnakeFactory process

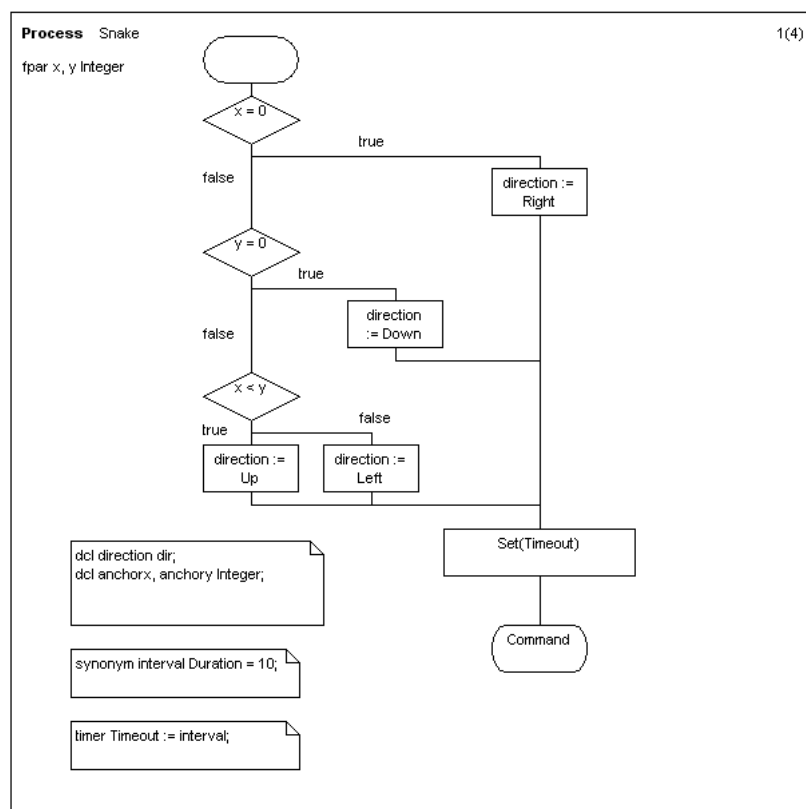


Figure V.5: Snake process

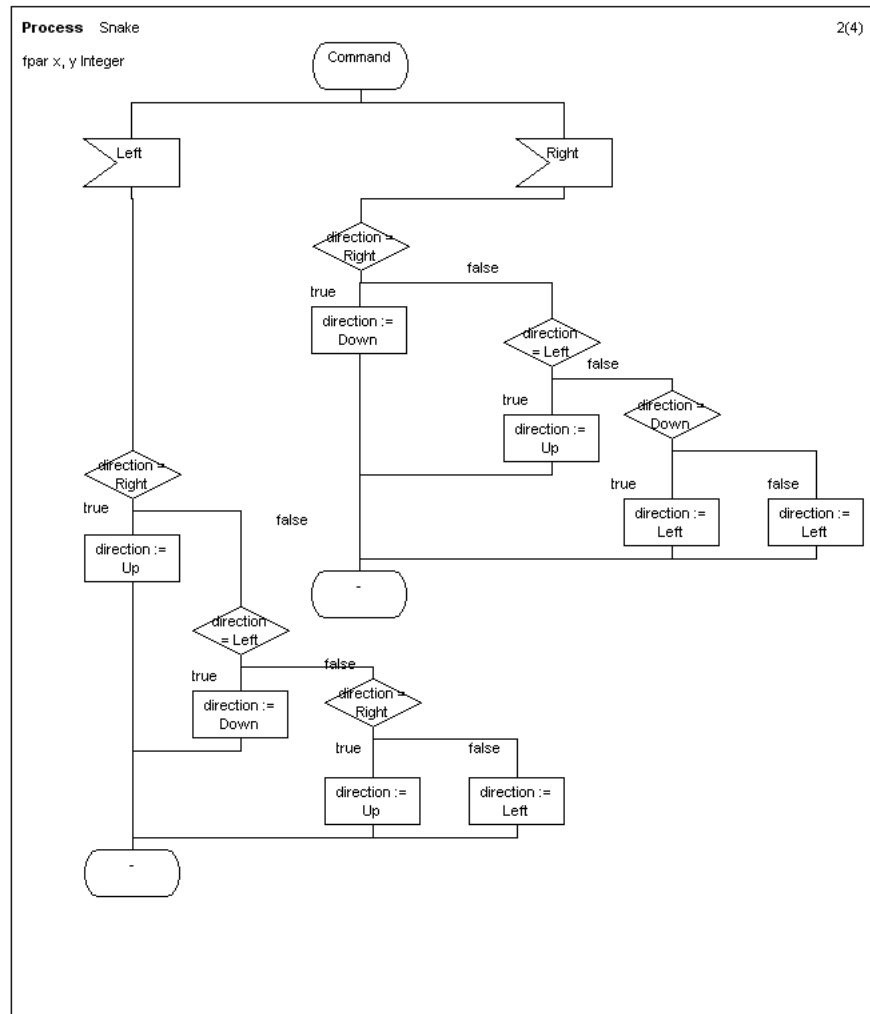


Figure V.6: Snake process (continued)

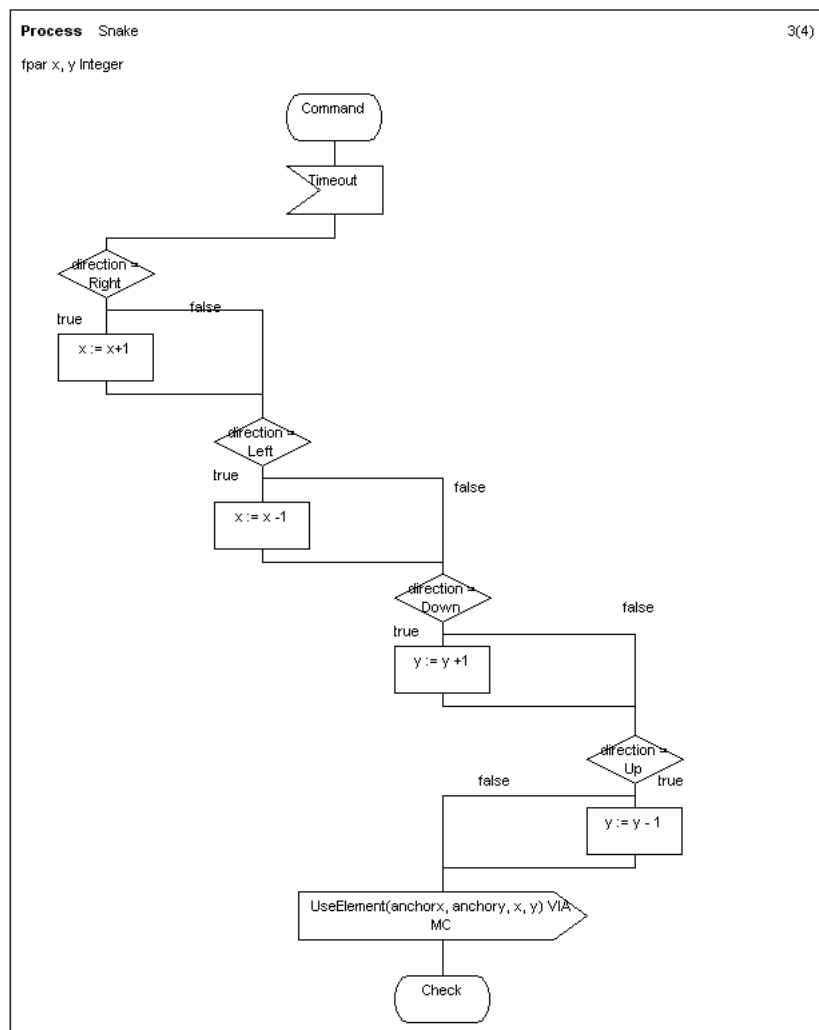


Figure V.7: Snake process (continued)

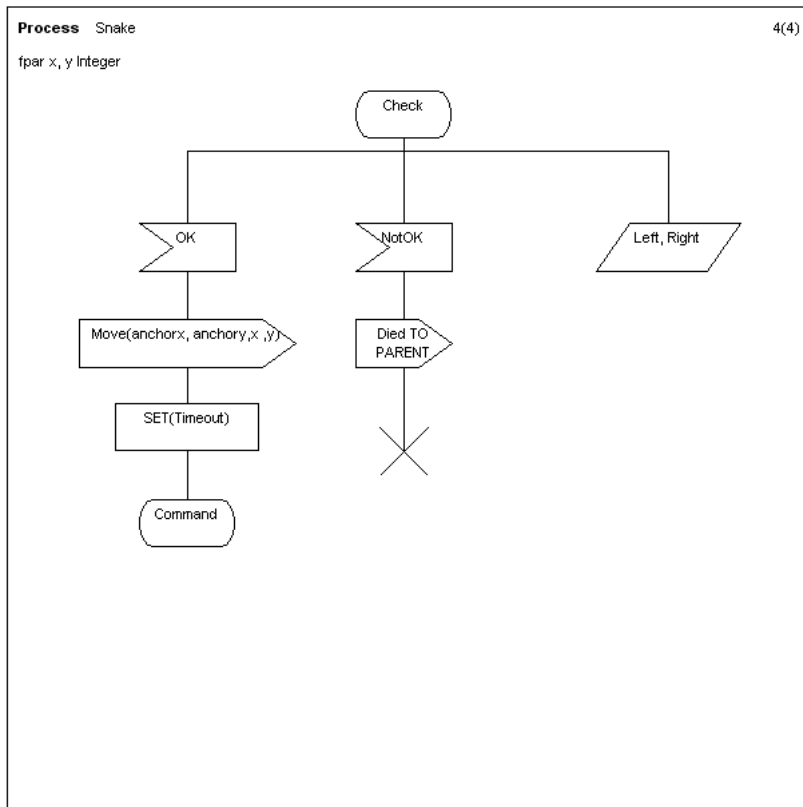


Figure V.8: Snake process (continued)

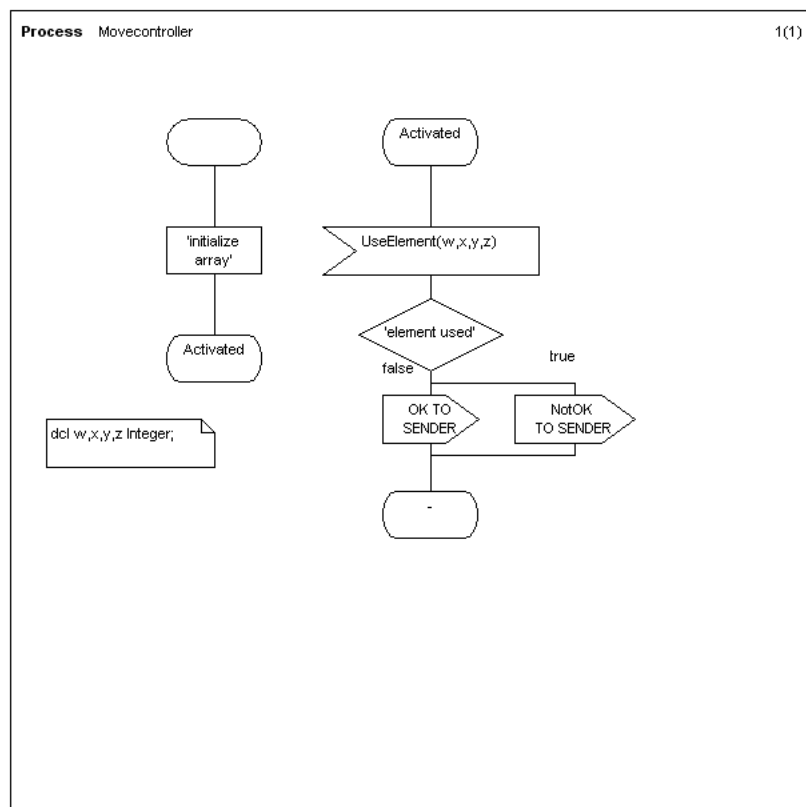


Figure V.9: Movecontroller process

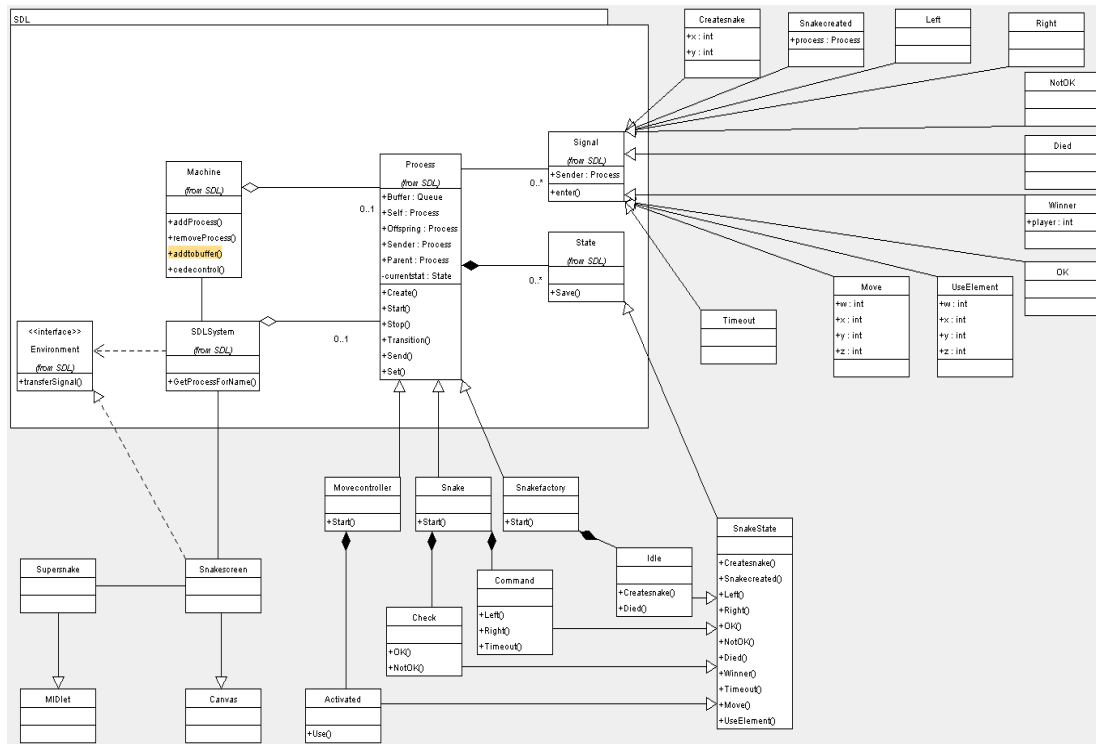


Figure V.10: A UML model for the game

The Mobile Information Device Profile (MIDP) [24], combined with the Connected Limited Device Configuration (CLDC), is a Java runtime environment for mobile information devices (MIDs). It provides core functionality—the user interface, network connectivity, local data storage, and application life cycle management—packaged as a standardised runtime environment.

The MIDP profile uses the `MIDlet` class to reference the application. This will be specialised, in our case, by the `Supersnake` class. The actual screen is represented in MIDP with the canvas, which will be specialised by the `Snakescreen` class. The ease with which we can do this demonstrates the advantage of translating via abstract classes and specialisation. The UML model for the game is shown in Figure V.10.

Most of the remaining specialisations follow immediately from the SDL model. Specialisations of `signal` are required for each signal, with the constructors being overridden to pass the parameters required. The class `SnakeState` contains definitions for each of the signal entries within the SDL model: as every state inherits and specialises this class, we know that every signal can be received in every state, even if a specific entry has not been created for a signal. The classes `MIDlet` and `Canvas` provide for interaction with the device platform.

We may produce an implementation for this UML model using the J2ME environment, starting with a machine class that represents the target architecture:

```
package SDL;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;

public class Machine {

    protected static SDLSys systemname;
```

```

public static Vector processes = new Vector();

public static Vector buffer = new Vector();
//This buffer is added. to represent a single buffer

protected static synchronized void addProcess(Process process){
    processes.addElement(process);
}

protected static synchronized void removeProcess(Process process){
    processes.removeElement(process);
}

public static synchronized void addtobuffer (Signal signal, Process process){
    if (process == null)
        systemname.Env.transferSignal(signal);
    else
        process.buffer.addElement(signal);
};

public static synchronized void cedecontrol() {

    if (!processes.isEmpty()) {
        Process p;
        int i = -1;
        while (true) {

            if (!processes.isEmpty()) {
                try { p = (Process)processes.elementAt(++i); }
                catch (ArrayIndexOutOfBoundsException ex) {break;}
                if (!p.Started) {
                    p.Started = true;
                    p.Sender = null;
                    p.Start();
                }
                else
                {
                    if (p.duration == 1) {
                        p.duration = 0;
                        addtobuffer (p.timeSignal, p);
                    }
                    else if (p.duration > 0){
                        --p.duration;
                    }
                    Signal signal = p.currentstate.Nextsignal(p);
                    if (!(signal == null)) {
                        p.Sender = signal.Sender;
                        signal.enter(p.currentstate);
                    }
                }
            }

        }
    }
}
}

```

This implements a simple single-threaded model, running through each process in turn to check for outstanding signals. This could be extended to more complicated implementation with a thread for each process, but in a single-processor device this would not be likely to result in any increased

performance.

We require a signal class for each signal within the system: for example, here is the implementation for Left:

```
package Supersnake.Signals;

import SDL.*;

public class Left extends Signal {
    public void enter(State state) {
        SnakeState s = (SnakeState)state;
        s.Left();
    }
}
```

This has to be reproduced for each signal, but it is a mechanical task and lends itself to automatic generation. For a signal with parameters, the parameters have to be explicitly coded into the signal:

```
package Supersnake.Signals;

import SDL.*;

public class Createsnake extends Signal {

    private int x, y;

    public Createsnake (int X, int Y) {
        x = X; y = Y;
    };

    public void enter(State state) {
        SnakeState s = (SnakeState)state;
        s.Createsnake(x,y);
    }
}
```

The SnakeState class ensures that every signal is understood, even if the result is that it will be discarded (for the current state):

```
package Supersnake.Signals;

import SDL.*;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public abstract class SnakeState extends State {

    public void Createsnake (int x, int y) {
    };

    public void Snakecreated (Process p) {
    };

    public void Left() {
    };

    public void Right() {
    };

    public void OK() {
    };
}
```

```

    public void NotOK() {
    };

    public void Died() {
    };

    public void Winner(int i) {
    };

    public void Timeout() {
    };

    public void Move(int w, int x, int y, int z) {
    };

    public void UseElement(int w, int x, int y, int z) {
    };
}

```

The processes are created directly from the SDL processes. The Movecontroller class controls the movement of snakes:

```

package Supersnake;

import SDL.*;
import Supersnake.Signals.*;
import javax.microedition.lcdui.*;

public class Movecontroller extends Process {

    private static int anchorx, anchory, x, y;
    boolean[][] screen = new boolean[90][60];

    public void Start() {
        for (int i = 0; i < x; i++)
            for (int j = 1; j < y; j++)
                screen[i][j] = false;
        Transition (new Activated());
    }

    class Activated extends SnakeState {
        public void Use(int W, int X, int Y, int Z) {
            anchorx = W;
            anchory = X;
            x = Y;
            y = Z;

            try {
                if (!screen[x][y]) {
                    screen[x][y] = true;
                    Send (new OK(), Sender);
                }
                else
                    Send (new NotOK(), Sender);
            } catch (java.lang.ArrayIndexOutOfBoundsException ie) {
                Send (new NotOK(), Sender);
            }
        }
    }
}

```

The limited support for datatypes in the SDL language meant that we were unable to fully specify the two-dimensional array at the modelling level: instead, we can introduce this information during the process of specialisation and implementation.

V.3 Targetting SDL-10

Although the target environment for SDL-10 is the Ericsson AXE-10 switch—not often used for multi-player games of this type—that language, and its implementation platform, will serve as a useful test of the framework, in terms of platform independence and flexibility. We can code up the model using our SDL-10 semantics,

```

IDLE, COMMAND, CHECK, ACTIVATED : STATE
CREATESNAKE, SNAKECREATED,
LEFT, RIGHT, OK, DIED,
WINNER, TIMEOUT, MOVE, USELEMENT : SIGNAL
SnakeFactory, Snake, MoveController : SDLProcess

SnakeFactory =
  ({(IDLE, ({}), null))}
  , {((IDLE, CREATESNAKE) ↦
    (IDLE, ({SNAKECREATED}, activate(({COMMAND, ({}), null))}))),
    ((IDLE, DIED) ↦ (IDLE, ({WINNER}, null))),
    ((IDLE, DIED) ↦ (IDLE, ({}), null)))})

Snake =
  ({
  , {((COMMAND, LEFT) ↦ (COMMAND, ({}), null)),
    ((COMMAND, RIGHT) ↦ (COMMAND, ({}), null)),
    ((COMMAND, TIMEOUT) ↦ (CHECK, ({USELEMENT}, null))),
    ((CHECK, OK) ↦ (COMMAND, ({MOVE}, null))),
    ((CHECK, NOTOK) ↦ (dud, ({DIED}, stop)))})

MoveController =
  ({(ACTIVATED, ({}), null))}
  , {((ACTIVATED, USELEMENT) ↦ (ACTIVATED, ({OK}, null))),
    ((ACTIVATED, USELEMENT) ↦ (ACTIVATED, ({NOTOK}, null)))})

```

and use this as a basis for analysis (prototyping or Java implementation would be inappropriate here). We can, for example, demonstrate using the above definition that the model is free of deadlocks (even though individual process instances, representing snakes, are expected to halt).

The model does specify that if `left` or `right` signals are received in state `check`, then they are to be saved, and it is this that reveals an inadequacy of the implementation platform (and hence a limit to our platform independence): SDL-10 does not support `save`, and hence an implementation here may lose an order to change direction. Harmless enough in the context of a game, but clearly problematic for other embedded systems.

To demonstrate formally how this can be proved we can create trace that in the SDL version of the system would not occur. In this case the Snake process is created and ingests a request to move to the right, but when determining the validity of the move a request is serviced to move left. In SDL the request would be saved, but due to the limitations of SDL-10 the request is serviced immediately.

$$FaultyTrace : TRACESEGMENT$$
$$FaultyTrace = \langle CREATESNAKE, SNAKECREATED, RIGHT, \\ TIMEOUT, USEELEMENT, LEFT, OK, MOVE \rangle$$

This can be proved as *FaultTrace* is part of the traces that combination of processes can produce then we can demonstrate a characteristic of poor safety in the system.

$$FaultyTrace \subseteq traces(SnakeFactory, Snake, MoveController)$$

(An implementation of the SDL-10 translation in Java, done for the sake of curiosity, did not in fact appear to lose the signals, possibly due to a platform characteristic that resolved the race condition in a satisfactory fashion. This could not, of course, be relied upon.)

VI

Conclusion

In this thesis, we have

- given a formal semantics to a domain-specific version of SDL, developed by Ericsson for the purposes of software component generation;
- shown how more abstract characterisations of asynchronous or buffered systems can be produced using a process calculus, based upon the CSP formalism;
- demonstrated the applicability of the semantics and the calculus by means of a case study.

We have defined also a framework, as a collection of related mechanisms, for describing buffered systems. This framework may be used—as shown in Figure I.1 in the opening chapter—to support the flow of software development activity. Within this framework, the semantics was a halfway house between informal description and full SDL, formal enough to allow mathematical reasoning, but without the block, process, and signal declarations needed for final implementation.

The disadvantage facing the state-based semantics is that it represents an additional notation, an unwelcome proposition in terms of industrial deployment. The calculus CBS faces the same disadvantage, but does offer greater flexibility and abstraction. The implementation of appropriate tool support could conceivably unlock the benefits of the notation in an industrial context, were this tool support to include at least the ability to generate SDL code.

An obvious direction for further study would be the definition of a single, common semantics for SDL-10 and CBS, extended perhaps to incorporate the existing dynamic semantics of SDL. This would provide a basis for validating the generation tool proposed above, for checking that the semantics of the generated SDL is consistent with that of the source model. It may even be possible to generate the compiler aspect of the tool directly from the semantics, as in [20].

The pragmatic considerations in using one language and translating it into another are significant: it is important that the translation should—as far as possible—allow for traceability, of concepts, of functionality, of structure, and even of syntax. The tool would have to provide disassembly as well as compilation so that any low level changes could be reflected back in the high level models (or revealed as problematic, and thus not carried out).

The tool support might be drawn in part from the existing support for model-driven development in UML and related notations. We might define a refinement relation from CBS to SDL, and thence to UML, in order to exploit that support in producing a Java implementation. The case study demonstrated that the semantics could be used to support a process of manual implementation in Java, with the relationships between model and code remaining relatively accessible.

However, the subtleties of behavioural description with UML mean that the idea of user familiarity is less persuasive than it might be at a programming level. UML has a small set of features,

but a number of ways in which these may be interpreted. Java has a large feature set, across the various packages, although each feature has a single, deterministic interpretation. Familiarity with the syntax of UML is thus less useful than familiarity with the syntax of Java, where knowing that a particular method exists is enough to make a significant contribution to effective usage, given that the user can then quickly obtain and understand a description of the intended functionality.

The dynamic semantics of UML are neither abstract nor well understood: the user is referred, effectively, to the behaviour of a specific implementation mechanism or tool platform. The limitations of SDL, as a domain specific language, are also a strength: there is less to explain, less to define, and less that can be interpreted in a variety of ways. The constraints upon the language, and the assumptions about the implementation domain or strategy, once formalised, act as a basis for automatic model or code generation.

A recommendation [14] has been made for translation from SDL to UML, using stereotypes for systems, blocks, processes, and signals. However, the mapping of processes is complicated by the fact that the behavioural notation in UML is based on state charts, in which the emphasis is upon defining regions of the state space. In SDL applications, the emphasis is upon the transitions, and it is the information about input, output, and parameters that is of the greatest interest [1]: state chart notations leave the connections between events implicit, or as ‘semantic variation points’ in the UML standard. The semantics defined in this thesis could help to address this problem, acting as a basis for formalising the definition of an SDL-to-UML translation, and for giving a semantics to an SDL profile within UML.

Furthermore, there is a lesson to be learnt in terms of tool support. Ericsson had assumed that the existing Telelogic Tau SDL toolsets would produce an accurate analysis of SDL-10 models, and that the differences in the underlying semantics would be insignificant. The semantics defined here demonstrated that this was not the case: the behaviour of the SDL-10 implementations was different enough for Ericsson to introduce guidance into their development processes. Extending the single, common semantics not only to SDL-10, SDL, and CBS, but also to the resulting profile of UML, and to the Java framework implementation, would provide a basis for end-to-end, reliable tool support, sufficient to guarantee that the behaviour of a *Java* implementation is consistent with the agreed semantics of the original CBS or SDL model.

VI.1 Related work

The use of shorthand version of SDL in an industrial context was suggested also in [18], the motivation being to develop simple models whose correctness can be established by inspection. The equivalent of CBS here is a language called Simple Specification Language (SSL). SSL lacks the formal basis of CBS, and hence does not support reasoning and analysis. It does, however, address a wider range of features within SDL than are considered here.

The Abstract State Machines (ASM) formalism has been used to model the SDL variant SDL-2000 [10], essentially a superset of the language considered here, with the addition of object-oriented features [11]. Concrete graphical and concrete textual forms are defined [12] as well as an abstract grammar notation; transformation rules are used to describe the relationships between the abstract grammar and most of the concrete grammar. The static semantics is then defined using predicates to characterise a syntactically-correct SDL specification as an abstract syntax tree.

The dynamic semantics [13] defines the set of computations that are associated with a specification; it applies only to well-formed abstract syntax. An abstract state machine (in ASM) is defined as an equivalent to our machine class, and user, and data and state are compartmentalised via interfaces. The ASM semantics is, in effect, an implementation or an interpreter for the language: comprehensive, but difficult to use as a basis for understanding or (manual) analysis. Indeed, a com-

piller is provided that is based upon this formal semantics [20], and has reportedly led to several revisions to the language definition, as well as to the semantics itself.

Other authors have adopted a similar approach to translation into UML: that is, via abstract classes and specialisation. [18] defines a subset of UML using class diagrams, with the intention of providing a platform for the interpretation of a domain-specific language called *Architectural TRIO*, based upon higher-order temporal logic. The UML class structures mirror the semantics of the domain-specific notation, just as ours mirror the semantics of SDL-10.

The translation of SDL into UML has also been considered in [19], where a meta-modelling approach was adopted. This allows for a more general framework than the one considered in this dissertation, one that would allow for the translation of any subset of SDL, or SDL-like language, that conformed to the SDL meta-model used. It requires an additional level of instantiation for application—and also for comprehension and traceability. The work in [19] could sensibly be extended with a formal, behavioural semantics, although none has been exhibited.

The game provided an indication, through the treatment of signals, of platform-dependent properties. In an industrial environment, directives would be required as part of a meta-language for the compiler. [16] has shown how SDL could be extended with constructs for targeting embedded systems. These enable more efficient use of resources, such as the memory available to the CPU, and explain how to map to implementation features such as centrally-assigned variables. Further development of applications along the lines of this case study would require such constructs for the automatic generation of working systems.

Bibliography

- [1] M. Bjorkander. Graphical Programming Using UML and SDL. *IEEE Computer Practices*, 33(12), December 2000.
- [2] Ericsson Telecom, Stockholm. *AXE-10 Survey, Ericsson Reference, EN/LZT 101 1513 R2A*, 1998.
- [3] Ericsson Telecom, Stockholm. *SDL10 Language Reference Manual, Ericsson Reference SMO98-XPR-63 Ver: 1.2*, 1998.
- [4] C. Grealish. Formal Semantics of an Industrially applicable variant of the Specification and Description Language. *IEE Proceedings Software*, 148(4), 2001.
- [5] Colin Grealish. SDL into CSP, MSc Dissertation, Computing Laboratory, Oxford University, Ericsson Reference XL/EF-97:189 Uen. Master's thesis, Oxford University, 1997.
- [6] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [7] C.A.R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.
- [8] International Organization for Standardization. *Information technology - Programming languages, their environments and system software interfaces - Vienna Development Method - Specification Language - Part 1: Base language*, September 2006.
- [9] ITU General Secretariat, Geneva. *ITU-T Recommendation Z.100: CCITT Specification and Description Language (SDL), 03/93*, 1993.
- [10] ITU General Secretariat, Geneva. *ITU-T Recommendation Z.100: Specification and Description Language, SDL, 2000*, 1999.
- [11] ITU General Secretariat, Geneva. *ITU-T Recommendation Z.100: Annex F1: SDL Formal Definition: General Overview*, 2000.
- [12] ITU General Secretariat, Geneva. *ITU-T Recommendation Z.100: Annex F2: SDL Formal Definition: Static Semantics*, 2000.
- [13] ITU General Secretariat, Geneva. *ITU-T Recommendation Z.100: Annex F3: SDL Formal Definition: Dynamic Semantics*, 2000.
- [14] ITU General Secretariat, Geneva. *Recommendation Z.109, SDL Combined with UML*, 2000.
- [15] Ellsberg J., Hogref D., and Sarma A. *Formal Object-oriented Language for Communicating Systems*. Prentice Hall, 1997.
- [16] Christoforos Kavadias, Bernard Perrin, Vangelis Kollias, and Michael Loupis. Enhanced SDL Subset for the Design and Implementation of Java-Enabled Embedded Signalling Systems. Technical Report SDL/03, TELETEL SA, Centre Suisse d'Electronique et de Microtechnique and SOLINET GmbH Solutions for Innovative Networks, 2003.

- [17] Object Management Group. *Unified Modeling Language (OMG UML), Infrastructure, V2.1.2*, November 2007.
- [18] Matteo Pradella, Matteo Rossi, and Dino Mandrioli. A UML-Compatible Formal Language for System Architecture Description. Technical report, Politecnico di Milano, Italy, 2005.
- [19] Andreas Prinz, Markus Scheidgen, and Merete S.Tveit. A Model-Based Standard for SDL. Technical Report SDL/07, Humboldt University, Germany, 2007.
- [20] Andreas Prinz and Martin v. Loewis. Generating a Compiler for SDL from the Formal Language Definition. Technical Report ISBN: 3-540-40539-9, Humboldt-University, Germany, 2003.
- [21] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1988.
- [22] Kenneth J. Turner. *Using Formal Description Techniques*. John Wiley and Sons, 1993.
- [23] History of SDL. <http://www.sdl-forum.org/sdl2000present/sld002.htm>, October 2008.
- [24] Mobile Information Device Profile (MIDP). <http://java.sun.com/javame/index.jsp>, April 2008.
- [25] The Java ME Platform. <http://java.sun.com/javame/index.jsp>, April 2008.

Appendix A:Java Implementation

SDL Environment Class

```
package SDL;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public interface Environment {

    public void transferSignal (Signal s);

}
```

SDL Machine Class

```
package SDL;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;

public class Machine {

    protected static SDLsystem systemname;

    public static Vector processes = new Vector();

    public static Vector buffer = new Vector();
    //This buffer is added. to represent a single buffer

    protected static synchronized void addProcess(Process process){
        processes.addElement(process);
    }

    protected static synchronized void removeProcess(Process process){
        processes.removeElement(process);
    }

    public static synchronized void addtobuffer (Signal signal, Process process){
        if (process == null)
            systemname.Env.transferSignal(signal);
        else
            process.buffer.addElement(signal);
    };

    public static synchronized void cedecontrol() {

        if (!processes.isEmpty()) {
            Process p;

```

```

int i = -1;
while (true) {

    if (!processes.isEmpty()) {
        try { p = (Process)processes.elementAt(++i); }
        catch (ArrayIndexOutOfBoundsException ex) {break;}
        if (!p.Started) {
            p.Started = true;
            p.Sender = null;
            p.Start();
        }
        else
        {
            if (p.duration == 1) {
                p.duration = 0;
                addtobuffer (p.timeSignal, p);
            }
            else if (p.duration > 0){
                --p.duration;
            }
            Signal signal = p.currentstate.Nextsignal(p);
            if (!(signal == null)) {
                p.Sender = signal.Sender;
                signal.enter(p.currentstate);
            }
        }
    }
}
}
}
}

```

SDL Process Class

```

package SDL;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;

public abstract class Process {

    protected State currentstate;
    public Vector buffer = Machine.buffer;
    // This line has been changed
    protected Process Offspring, Sender, Parent;
    protected Process Self = this;
    protected static Process Env = null;
    protected boolean Started = false;
    protected int duration = 0;
    protected Signal timeSignal;

    public Process() {
        buffer = new Vector();
    };

    protected void Create(Process process) {
        Machine.addProcess (process);
        process.Parent = this;
        Offspring = process;
    }
}

```

```

};

protected abstract void Start();

protected void Stop() {
    Machine.removeProcess(this);
};

protected void Transition(State state) {
    currentstate = state;
    Machine.cedecontrol();
}

protected void Send (Signal signal, Process process) {
    signal.Sender = this;
    if (process == null)
        Machine.systemname.Env.transferSignal(signal);
    else
        process.buffer.addElement(signal);
};

protected void Send (Signal signal, String processname) {
    signal.Sender = this;
    Machine.systemname.getProcessForName(processname).buffer.addElement(signal);
};

protected void set (int d, Signal s) {
    duration = d;
    timeSignal = s;
}

}

```

SDL System Class

```

package SDL;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class SDLsystem {

    protected static String[] UndynamicProcesses;
    protected Process[] Processlist;

    public Environment Env;

    private String trimprocess (String s) {
        int i = s.lastIndexOf('.');
        return s.substring(0,i);
    }

    public SDLsystem (String[] list, Environment e) {
        UndynamicProcesses = list;
        Env = e;
        Processlist = new Process[UndynamicProcesses.length];
        for (int i = 0; i < Processlist.length; i++)
            try {
                Processlist[i] =

```

```

        (Process)Class.forName((trimprocess(e.getClass().getName()))
        + "." + UndynamicProcesses[i]).newInstance();
Machine.addProcess(Processlist[i]);
    }
    catch (Exception ex) { }
    Machine.systemname = this;
}

public Process getProcessForName(String p) {
    int i;
    for (i = 0; i < Processlist.length; i++) {
        if (UndynamicProcesses[i].equals(p))
            break;
    }
    return Processlist[i];
}
}

```

SDL Signal Class

```

package SDL;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public abstract class Signal {

    public Process Sender;

    public void enter(State state) {
    }

}

```

SDL State Class

```

package SDL;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public abstract class State {

    protected String[] Signals;

    protected Signal Nextsignal(Process Currentprocess) {
        boolean saved = true;
        int index = 0;
        if (!(Signals == null)) {
            for (int i = 0; saved & i < Currentprocess.buffer.size(); i++) {
                for (int j = 0; j < Signals.length; j++) {
                    if (!Currentprocess.buffer.elementAt(i).getClass().getName().endsWith(Signals[j])) {
                        saved = false;
                        index = i;
                        break;
                    }
                }
            }
        }
    }
}

```

```

        else saved = false;
        if (!saved) {
            try {
                Signal s = (Signal)Currentprocess.buffer.elementAt(index);
                Currentprocess.buffer.removeElementAt(index);
                return s;
            }
            catch (ArrayIndexOutOfBoundsException ex) { return null;}
        }
        return null;
    };

    public void Save (String[] s) {
        Signals = s;
    };
}

```

Supersnake Createsnake Class

```

package Supersnake.Signals;

import SDL.*;

public class Createsnake extends Signal {

    private int x, y;

    public Createsnake (int X, int Y) {
        x = X; y = Y;
    };

    public void enter(State state) {
        SnakeState s = (SnakeState)state;
        s.Createsnake(x,y);
    }
}

```

Supersnake Died Class

```

package Supersnake.Signals;

import SDL.*;

public class Died extends Signal {

    public void enter(State state) {
        SnakeState s = (SnakeState)state;
        s.Died();
    }
}

```

Supersnake Left Class

```

package Supersnake.Signals;

import SDL.*;

public class Left extends Signal {
    public void enter(State state) {
        SnakeState s = (SnakeState)state;
        s.Left();
    }
}

```

```
    }
}
```

Supersnake Move Class

```
package Supersnake.Signals;

import SDL.*;

public class Move extends Signal {

    public int w,x,y,z;

    public Move (int W, int X, int Y, int Z) {
        w = W;
        x = X;
        y = Y;
        z = Z;
    }

    public void enter(State state) {
        SnakeState s = (SnakeState)state;
        s.Move(w,x,y,z);
    }
}
```

Supersnake NotOK Class

```
package Supersnake.Signals;

import SDL.*;

public class NotOK extends Signal {

    public void enter(State state) {
        SnakeState s = (SnakeState)state;
        s.NotOK();
    }
}
```

Supersnake OK Class

```
package Supersnake.Signals;

import SDL.*;

public class OK extends Signal {

    public void enter(State state) {
        SnakeState s = (SnakeState)state;
        s.OK();
    }
}
```

Supersnake Right Class

```
package Supersnake.Signals;
```

```
import SDL.*;

public class Right extends Signal {

    public void enter(State state) {
        SnakeState s = (SnakeState)state;
        s.Right();
    }
}
```

Supersnake Snakecreated Class

```
package Supersnake.Signals;

import SDL.*;

public class Snakecreated extends Signal {

    public Process process;

    public Snakecreated (Process p) {
        process = p;
    };

    public void enter(State state) {
        SnakeState s = (SnakeState)state;
        s.Snakecreated(process);
    }
}
```

Supersnake SnakeState Class

```
package Supersnake.Signals;

import SDL.*;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public abstract class SnakeState extends State {

    public void Createsnake (int x, int y) {
    };

    public void Snakecreated (Process p) {
    };

    public void Left() {
    };

    public void Right() {
    };

    public void OK() {
    };

    public void NotOK() {
    };
}
```

```

};

public void Died()    {
};

public void Winner(int i)    {
};

public void Timeout() {
};

public void Move(int w, int x, int y, int z) {
};

public void UseElement(int w, int x, int y, int z) {
};
}

```

Supersnake Timeout Class

```

package Supersnake.Signals;

import SDL.*;

public class Timeout extends Signal {

    public void enter(State state) {
        SnakeState s = (SnakeState)state;
        s.Timeout();
    }
}

```

Supersnake UseElement Class

```

package Supersnake.Signals;

import SDL.*;

public class UseElement extends Signal {

    public int w,x,y,z;

    public UseElement (int W, int X, int Y, int Z) {
        w = W;
        x = X;
        y = Y;
        z = Z;
    }

    public void enter(State state) {
        SnakeState s = (SnakeState)state;
        s.UseElement (w,x,y,z);
    }
}

```

Supersnake Winner Class

```

package Supersnake.Signals;

import SDL.*;

```

```

public class Winner extends Signal {

    public int player;

    public Winner (int p) {
        player = p;
    }

    public void enter(State state) {
        SnakeState s = (SnakeState)state;
        s.Winner(player);
    }
}

```

Supersnake Movecontroller Class

```

package Supersnake;

import SDL.*;
import Supersnake.Signals.*;
import javax.microedition.lcdui.*;

public class Movecontroller extends Process {

    private static int anchorx, anchory, x, y;
    boolean[][] screen = new boolean[90][60];

    public void Start() {
        for (int i = 0; i < x; i++)
            for (int j = 1; j < y; j++)
                screen[i][j] = false;
        Transition (new Activated());
    }

    class Activated extends SnakeState {
        public void Use(int W, int X, int Y, int Z) {
            anchorx = W;
            anchory = X;
            x = Y;
            y = Z;

            try {
                if (!screen[x][y]) {
                    screen[x][y] = true;
                    Send (new OK(), Sender);
                }
                else
                    Send (new NotOK(), Sender);
            } catch (java.lang.ArrayIndexOutOfBoundsException ie) {
                Send (new NotOK(), Sender);
            }
        }
    }
}

```

Supersnake Snake Class

```
package Supersnake;

import SDL.*;
import Supersnake.Signals.*;
import javax.microedition.lcdui.*;

public class Snake extends Process {

    private static String Forward = "Forward";
    private static String Left = "Left";
    private static String Right = "Right";
    private static String Up = "Up";
    private static String Down = "Down";
    private static String Movecontroller = "Movecontroller";
    private static int Interval = 10;

    String Direction = Right;

    private int x,y, anchorx, anchory;

    public Snake(int X, int Y) {
        anchorx = X;
        anchory = Y;
        x = X;
        y = Y;
    }

    public void Start() {
        if (x == 0)
            Direction = Right;
        else if (y == 0)
            Direction = Down;
        else if (x < y)
            Direction = Up;
        else Direction = Left;
        set(Interval,new Timeout());
        Transition(new Command());
    }

    class Command extends SnakeState {

        public void Left() {
            if (Direction == Right)
                { Direction = Up; }
            else if (Direction == Left)
                { Direction = Down; }
            else if (Direction == Down)
                { Direction = Right; }
            else {Direction = Left; }
        }

        public void Right() {
            if (Direction == Right)
                { Direction = Down; }
            else if (Direction == Left)
                { Direction = Up; }
            else if (Direction == Down)
                { Direction = Left; }
            else{ Direction = Right; }
        }
    }
}
```

```

        public void Timeout() {
            if (Direction == Right)
                x++;
            if (Direction == Left)
                x--;
            if (Direction == Down)
                y++;
            if (Direction == Up)
                y--;
            Send(new UseElement(anchorx, anchory, x, y), Movecontroller);
            Transition (new Check());
        }
    }

    class Check extends SnakeState {

        public Check() {
            Save(new String[]{Forward, Left, Right} );
        }

        public void OK() {
            Send(new Move(anchorx, anchory, x, y), Env);
            anchorx = x;
            anchory = y;
            set(Interval, new Timeout());
            Transition (new Command());
        }

        public void NotOK() {
            Send(new Died(), Parent);
            Stop();
        }
    }
}

```

Supersnake Snakefactory Class

```

package Supersnake;

import SDL.*;
import Supersnake.Signals.*;

import java.util.*;

public class Snakefactory extends Process {

    public Process[] SnakeList = new Process[4];
    public Process SnakeCreator;
    public int i = -1;

    public void Start() {
        Transition(new Idle());
    }

    class Idle extends SnakeState {

```

```

    public void Createsnake(int x, int y) {
        Create(new Snake(x,y));
        SnakeList[++i] = Offspring;
        SnakeCreator = Sender;
        Send((new Snakecreated(Offspring)),Sender);
    }

    public void Died() {
        for (int j = 0; j < 4; j++) {
            if (Sender == SnakeList[j]) {

                SnakeList[j] = null;
                if (--i <= 0) {
                    for (int k = 0; k < 4; k++) {
                        if (SnakeList[k] != null) {

                            Send((new Winner(k)),SnakeCreator);
                            i = 5;
                        }
                    }
                    if (i !=5)
                        Send((new Winner(1)),SnakeCreator);
                }
            }
        }
    }
}

```

Supersnake Snakescreen Class

```

package Supersnake;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.lang.*;
import java.util.*;

import SDL.*;
import Supersnake.Signals.*;

public class Snakescreen extends Canvas implements Runnable {

    private Process player1, player2, player3, player4;
    public static Vector pending = new Vector();
    private boolean forceRedraw = true;
    private boolean gameOver = false;
    private boolean moved = false;
    private boolean started = false;
    private SDLsystem theSystem;
    private Process theFactory;
    private int w,x,y,z;
    private int score = 0;
    private int level = 7;
    private int players;
    private int width;
    private int height;
    private int winner;

```

```

static int CellWidth;
static int CellHeight;
private MIDlet game;
private static final int START_POS = 3;
private static final int SCORE_CHAR_HEIGHT;
private static final int SCORE_CHAR_WIDTH;
private static final int SCORE_HEIGHT;
private static final int DEFAULT_WAIT = 400;
private static final byte MAX_LEVELS = 10;
private static final int FOOD_COLOUR = 0x0000ff00;
private static final int TEXT_COLOUR = 0x00ff0000;
private static final int ERASE_COLOUR = 0x00ffffff;
private static final int DRAW_COLOUR = 0x00000000;
public static final int CELL_SIZE = 5;

    static {
        Font defaultFont = Font.getDefaultFont();
        SCORE_CHAR_WIDTH = defaultFont.charWidth('S');
        SCORE_CHAR_HEIGHT = defaultFont.getHeight();
        SCORE_HEIGHT = SCORE_CHAR_HEIGHT * 2;
    }

protected Vector buffer;

private void Send (Signal signal, Process process) {
    signal.Sender = null;
    process.buffer.addElement(signal);
}

public Snakescreen(int p) {
    players = p;
    buffer = new Vector();
    String[] list = new String[]{"Snakefactory",
        "Movecontroller"};
    theSystem = new SDLsystem(list, this);
    theFactory = theSystem.getProcessForName("Snakefactory");
    width = round(getWidth());
    height = round(getHeight()-SCORE_HEIGHT);
}

private int round(int val) {
    int delta = (val-(START_POS*2)) % CELL_SIZE;
    return (val - delta);
}

public void createsnake(Process process) {
    moved = true;
    if (player1 == null) {
        player1 = process;}
    else if (player2 == null) {
        player2 = process;}
    else if (player3 == null) {
        player3 = process;}
    else if (player4 == null) {
        player4 = process;}
}

public synchronized void paint (Graphics g) {

    if (forceRedraw) {
        forceRedraw = false;

```

```

        g.setColor(ERASE_COLOUR);
        g.fillRect(0, 0, width,
            (height + SCORE_HEIGHT + CELL_SIZE));

        g.setColor(DRAW_COLOUR);
        g.drawRect(1, 1, (width - START_POS), (height - START_POS));

        g.translate(START_POS, START_POS);

    } else {
        g.translate(START_POS, START_POS);
    }

    if (gameOver) {
        Font overFont = g.getFont();
        int fontH = overFont.getHeight();
        int fontW = overFont.stringWidth("Player 1 wins!");
        g.setColor(ERASE_COLOUR);
        g.fillRect((width-fontW)/2 - 1, (height-fontH)/2,
            fontW + 2, fontH);
        g.setColor(TEXT_COLOUR);
        g.setFont(overFont);
        if (players == 1)
            g.drawString("You loose!", (width-fontW)/2, (height-fontH)/2,
                g.TOP|g.LEFT);
        else
            g.drawString("Player " + winner + " wins!", (width-fontW)/2, (height-fontH)/2,
                g.TOP|g.LEFT);
    } else if (moved){
        g.setColor(TEXT_COLOUR);
        while (pending.size() !=0) {
            Move m = (Move)pending.firstElement();
            g.drawLine(m.w, m.x, m.y, m.z);
            pending.removeElement(m);
        }
    }

    if (players != 0 & !started) {

        started = true;

        g.drawString("1: 1 & 4", START_POS, height,g.TOP|g.LEFT);

        if (players >= 2)
            g.drawString("2: 6 & 9",
                (width - (SCORE_CHAR_WIDTH * 2)),
                height, g.TOP|g.RIGHT);

        if (players >= 3)
            g.drawString("3: 3 & 2", START_POS, (height + SCORE_CHAR_HEIGHT), g.TOP|g.LEFT);

        if (players == 4)
            g.drawString("4: 7 & 8",
                (width - (SCORE_CHAR_WIDTH * 2)),
                (height + SCORE_CHAR_HEIGHT),
                g.TOP|g.RIGHT);
    }

    g.translate(-START_POS, -START_POS);
}

public void transferSignal (Signal signal) {

```

```

        if (signal.getClass().getName().equals("Supersnake.Signals.Snakecreated")) {
            Snakecreated s = (Snakecreated)signal;
            createsnake(s.process);
        }
        else if (signal.getClass().getName().equals("Supersnake.Signals.Move")) {
            moved = true;
            Move m = (Move)signal;
            pending.addElement(m);
            repaint();
        }
        else if (signal.getClass().getName().equals("Supersnake.Signals.Winner")) {
            Winner m = (Winner)signal;
            winner = ++m.player;
            gameOver = true;
            repaint();
        }
    }

    public void startgame() {
        Send (new Createsnake(0,((height-6)/2)), theFactory);
        if (players >= 2)
            Send (new Createsnake((width-6),((height-6)/2)), theFactory);
        if (players >= 3)
            Send (new Createsnake(((width-6)/2),(height-6)), theFactory);
        if (players == 4)
            Send (new Createsnake(((width-6)/2),0), theFactory);
    }

    public synchronized void keyPressed(int key) {
        switch (key) {
            case KEY_NUM1 : Machine.addtobuffer (new Left(), player1) ; break;
            case KEY_NUM4 : Machine.addtobuffer (new Right(), player1) ; break;
            case KEY_NUM9 : Machine.addtobuffer (new Left(), player2) ; break;
            case KEY_NUM6 : Machine.addtobuffer (new Right(), player2) ; break;
            case KEY_NUM3 : Machine.addtobuffer (new Left(), player3) ; break;
            case KEY_NUM2 : Machine.addtobuffer (new Right(), player3) ; break;
            case KEY_NUM7 : Machine.addtobuffer (new Left(), player4) ; break;
            case KEY_NUM8 : Machine.addtobuffer (new Right(), player4) ; break;
        }
    }

    public void run() {
        while (true) {
            try {
                Machine.cedecontrol();
                Thread.sleep(30);
            } catch (java.lang.InterruptedException ie) {
            }
        }
    }
}

```

Supersnake Supersnake Class

```
package Supersnake;
```

```

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

import SDL.*;

public class Supersnake extends MIDlet implements CommandListener {

    private Snakescreen theGame;

    private int players = 2;

    private Command exitCmd = new Command("Exit", Command.SCREEN, 3);

    private Command startCmd = new Command("Start", Command.SCREEN, 1);

    private Command OKCmd = new Command("OK", Command.OK, 1);

    public Supersnake() {
    }

    protected void destroyApp(boolean unconditional) {
        Display.getDisplay(this).setCurrent(null);
    }

    protected void pauseApp() {
    }

    protected void startApp() {

        Item[] levelItem = {
            new ChoiceGroup("Number of Players", Choice.EXCLUSIVE,
                new String[]{"1 Player", "2 Player", "3 Player", "4 Player"}, null));
        Form f = new Form("Players", levelItem);
        f.addCommand(OKCmd);
        f.setCommandListener(this);
        Display.getDisplay(this).setCurrent(f);
    }

    public void commandAction(Command c, Displayable d) {
        if (c == exitCmd) {

            destroyApp(false);
            notifyDestroyed();

        } else if (c == OKCmd) {
            Form f = (Form)d;
            ChoiceGroup g = (ChoiceGroup)f.get(0);
            players = g.getSelectedIndex() + 1;
            theGame = new Snakescreen(players);
            theGame.addCommand(exitCmd);
            theGame.addCommand(startCmd);
            theGame.setCommandListener(this);

            Display.getDisplay(this).setCurrent(theGame);

            try {
                Thread myThread = new Thread(theGame);
                myThread.start();
            }
        }
    }
}

```

```

        } catch (Error e) {
            destroyApp(false);
            notifyDestroyed();
        }

    } else if (c == startCmd) {
        theGame.removeCommand(startCmd);
        theGame.startgame();
        Display.getDisplay(this).setCurrent(theGame);
    }
}
}

```

SDL-10 Machine Class

```

package SDL;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;

public class Machine {

    protected static SDLsystem systemname;

    public static Vector processes = new Vector();

    public static Vector buffer = new Vector();
    //This buffer is added. to represent a single buffer

    protected static synchronized void addProcess(Process process){
        processes.addElement(process);
    }

    protected static synchronized void removeProcess(Process process){
        processes.removeElement(process);
    }

    public static synchronized void addtobuffer (Signal signal, Process process){
        if (process == null)
            systemname.Env.transferSignal(signal);
        else
            process.buffer.addElement(signal);
    };

    public static synchronized void cedecontrol() {

        if (!processes.isEmpty()) {
            Process p;
            int i = -1;
            while (true) {

                if (!processes.isEmpty()) {
                    try { p = (Process)processes.elementAt(++i); }
                    catch (ArrayIndexOutOfBoundsException ex) {break;}
                    if (!p.Started) {
                        p.Started = true;
                        p.Sender = null;
                        p.Start();
                    }
                }
            }
        }
    }
}

```

```

else
{
    if (p.duration == 1) {
        p.duration = 0;
        addtobuffer (p.timeSignal, p);
    }
    else if (p.duration > 0){
        --p.duration;
    }
    Signal signal = p.currentstate.Nextsignal(p);
    if (!(signal == null)) {
        p.Sender = signal.Sender;
        signal.enter(p.currentstate);
    }
}

}

}

}

}

```

SDL-10 Process Class

```
package SDL;

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import java.util.*;

public abstract class Process {

    protected State currentstate;
    public Vector buffer = Machine.buffer;
    // This line has been changed
    protected Process Offspring, Sender, Parent;
    protected Process Self = this;
    protected static Process Env = null;
    protected boolean Started = false;
    protected int duration = 0;
    protected Signal timeSignal;

    public Process() {
        buffer = new Vector();
    };

    protected void Create(Process process) {
        Machine.addProcess (process);
        process.Parent = this;
        Offspring = process;
    };

    protected abstract void Start();

    protected void Stop() {
        Machine.removeProcess(this);
    };

    protected void Transition(State state) {
        currentstate = state;
    };
}
```

```

        Machine.cedecontrol();
    }

    protected void Send (Signal signal, Process process) {
        signal.Sender = this;
        if (process == null)
            Machine.systemname.Env.transferSignal(signal);
        else
            process.buffer.addElement(signal);
    };

    protected void Send (Signal signal, String processname) {
        signal.Sender = this;
        Machine.systemname.getProcessForName(processname).buffer.addElement(signal);
    };

    protected void set (int d, Signal s) {
        duration = d;
        timeSignal = s;
    }
}

```