A Graphical SDL Editor
Produced by the
Meta-Tool LOGGIE

by

Anneli Avatare
A Graphical SDL Editor
Produced by the
Meta-Tool LOGGIE

Anneli Avatare
anneli@sics.se

Abstract
We present a prototype implementation of a graphical editor for a subset of SDL (Specification and Description Language). SDL is a CCITT recommended standard for specifying and describing telecom systems. The editor is defined by the meta-tool LOGGIE, a tool for generating interactive language-oriented graphical editors. A short introduction to LOGGIE is given and the SDL application is described including a presentation of the chosen subset of the SDL syntax.

1. Introduction
LOGGIE [LOGGIE89], a Language Oriented Generator of Graphic Interactive Environments, is a meta-tool for generating language-oriented graphical editors. LOGGIE uses attribute grammars for describing language semantics and graphic presentations. LOGGIE has been used to design graphical environments for CCS [LOGGIE89] and G-LOTOS [G-LOTOS89].

SDL [SDL1,SDL2,Rochström85] applications can be found in the telephone applications area and within the data communication protocols area. If one can be precise and unambiguous at specification of a system then it will give benefits in following development work.

SDL specifications describe a whole system. A system contains a set of blocks which are connected to each other and to the environment by channels. Within a block there are one or more processes. Processes consists of different states and communicate with each other by signals.

The motivation for this work is to show the possibility to map the graphical syntax of SDL to LOGGIEs meta-notation. Our SDL editor is not complete, only a subset of the graphical grammar has been implemented.
In section 2 a short description of SDL is given and in section 3 the chosen subset of the SDL syntax is presented. Section 4 describes LOGGIE's meta-notation and in section 5 the LOGGIE specification of the SDL application is described. Section 6 presents the editor from a user point of view. It can be a good idea to read section 6 first to get an introduction of the editor and then start from the beginning.

2. SDL-Specification and Description Language
SDL has been designed having in mind the specification of behaviour of telecommunications switching systems, but can also be used to model other applications. SDL is well suited to all systems whose behaviour can be modelled by extended finite state machines. Examples can be found in the telephone applications area and within the data communication protocols area.

SDL has two different forms: one is called SDL/GR (SDL Graphic Representation) and is based on a set of graphical symbols. The other is called SDL/PR (SDL Phrase Representation) and is based on programlike statements. Both represent the same SDL concepts. SDL/PR and SDL/GR share a common subset.

It is not essential that an SDL specification should be executable by a machine, more important is the capability of transferring information from one human to another.

3. Definition of the subset of the SDL/GR syntax
This section presents the subset of the SDL/GR syntax which we support in our editor and explains the different graphical symbols in an SDL specification. A formal definition of this subset is given in the Appendix.

The graphical language has the advantage of clearly showing the structure of a system and it makes it easier for humans to visualize the specification. The textual phrase representation is best suited for machine use.
The graphical syntax is specified in Backus-Naur Form, extended with the following metasymbols:

`contains` is an infix operator, indicate that its right-hand argument should be placed within its left-hand argument.

`is connected to` is an infix operator, mean that its right-hand argument is connected to its left-hand argument.

`set` is a postfix operator, operating on the immediately preceding syntactic elements within curly brackets, and indicate an unordered set of items.

`is associated with` is an infix operator, indicate that its right-hand side argument is logically associated with its left-hand argument.

`is followed by` is an infix operator, mean that its right-hand argument follows its left-hand argument.

An SDL specification has a hierarchical structure, it starts with an overview of the system and the components in the system describe the system more in detail.

All the terminals in the grammar (see the Appendix) which include the word `symbol` are graphical symbols, e.g. `<text symbol>` (see figure 3.1).

![<text>](image)

Figure 3.1: A text symbol.

3.1 System
An SDL `system` contains everything the specification is trying to define. Each system is composed of a number of `blocks` connected together by `channels`. Each block in the system is independent from every other block. The system communicates with the environment through channels.

The syntax for system diagrams is defined in the Appendix, section 1. Figure 3.2 shows a system diagram. It is a frame symbol, enclosing all the other symbols; a system heading "SYSTEM System1"; a text symbol and a block interaction area with two graphical block references (Block1 and Block2) and a channel c.
3.2 Blocks

*Blocks* are the next level in the system specification. Within a block, processes can communicate with each other with signals. The only way of communication between processes in two different blocks is by sending signals that are transported by the channels between the blocks.

Within a block it is possible to define *communication paths* between the processes or between the processes and the environment of the block. The communication paths are called *signal routes*, see section 3.5. Blocks serve as a transport media for signals exchanged with the environment or between processes contained in the blocks.

The syntax for block diagrams is defined in the Appendix, section 2. Figure 3.3 shows a block diagram: a frame symbol which contains a block heading "BLOCK Block1", a text symbol, a process interaction area with two process references (P1 and P2) and a signal route r between them.

3.3 Channels

Channels transport signals between blocks of the system or between blocks and the environment. A channel may connect one block to another block or one block to the environment in one or both directions. Channels can be partitioned, i.e. it is possible to formally specify the behaviour of each channel. A list of signals is associated to a channel. A signal list symbol (see figure 3.4) defines the signal types which are allowed to be conveyed by the associated channel. The syntax is specified in the Appendix, section 2.
3.4 Processes

Processes are contained in blocks. The block definitions serve as structuring modules for an SDL specification.

A process is an extended finite state machine which defines the dynamic behaviour of a system. Processes wait for signals. For each type of signal that the process can receive a specific action is specified. A process contains a number of different states to perform the actions when a signal is received. After all the actions associated with a signal have occurred, the next state is entered and the process waits for a new signal.

Processes can be created when the system is created or by a create request from another process. The process body represents the actual graph of the finite state machine: a sequence of symbols that are connected to each other by directed arcs. A process always begins with a START symbol followed by a set of actions. A process terminates with a STOP symbol. The interpretation of a process starts when the process is created.

Some of the symbols within a process diagram are explained below:

**Create request:** A create request symbol represents the creation of another process instance of the type specified in the symbol. If the process requires actual parameters they must be given. It contains the name of the process type which is created and actual parameters if any.

![Create request symbol]

**Decision:** A decision is an action within a transition which asks a question. After that the process proceeds to one of the two or more paths following the decision depending on the answer from the question. A decision symbol contains a question and is associated with two or more actions.

![Decision symbol]
Input: An input symbol is attached to a state. If the signal within the input symbol arrives while the process is waiting in the state the transition which follows the input should be interpreted. It contains signals that can be received by the process.

Nextstate: A nextstate symbol represents a terminator of a transition. It specifies the state the process will assume when terminating the transition. It contains the state name.

Output: An output symbol represents the sending of a signal from one process to another or to itself. The destination process in the output statement contains the specification of signals, actual parameters and an optional destination.

Procedure call: A procedure call may occur wherever a task is allowed in either a process or a procedure graph (see the description of the procedure below). When a procedure is called, a procedure environment is created and the procedure begins to be interpreted and stops when the RETURN is reached. The procedure does not have its own input queue, but uses the input queue of the process that called it. A procedure call symbol contains a request of an interpretation of a separate defined part.
Procedure reference: A procedure reference symbol is a procedure symbol containing a procedure name representing a local procedure. It is a reference to a procedure definition.

Save: The save symbol allows the signal to be delayed until one or more other signals have arrived. A signal in a save symbol remains in the queue in the same sequential position. The symbol contains signals that need to be saved for future processing.

Set and Reset: In the SDL model, timers are meta-processes that are able to send signals to the process upon request. The SET operation requests a timeout to occur at specified time and the RESET operation cancels the specified timeout.

Start: A process diagram begins with a start symbol.

State: A state is a point in the process where no actions are being performed but where the input queue is monitoring for the arrival of incoming signals. When a signal arrives the process leaves the state and performs a specific sequence of actions. A state symbol contains the state name and the symbol is connected to input symbols and save symbols.
Stop: A stop symbol denotes immediate halt of the process.

Task: A task symbol is used in a transition to represent operations on variables or to represent special operations to be performed. A task symbol contains a list of statements or informal text.

The syntax for processes are defined in the Appendix, section 3. Figure 3.5 shows a process diagram: a frame symbol which contains a process heading "PROCESS P1 FPAR i integer" and some states and transitions. The diagram is not complete, only the beginning is shown.

3.5 Signal routes
Signal routes express communication paths similarly to the channels but they cannot be partitioned. They are used at block level. The signal routes represent communication between processes in a block or between processes and the block environment. When a signal route is connected from a process and to a channel at the environment it means that signals are sent via the channel to another block and its processes. The syntax is specified in the Appendix, section 3.
3.6 Procedures

SDL Procedures are similar to procedures in ordinary programming languages such as Pascal. Procedures are used to structure the processes into several levels of details and to allow often used or complicated sequences to be predefined. A specification structured into levels is easier to understand.

A procedure definition may only be contained in a process or in a procedure. A procedure is visible only to the process or procedure in which it is defined.

A procedure body is very similar to the process body but the start symbol and the return symbol are different (see figure 3.6). See also the description of the symbols in a process in section 3.4.

![Figure 3.6: A start symbol to the left and a return symbol to the right.](image)

The syntax for procedures is defined in the Appendix, section 4. Procedure diagrams are similar to process diagrams.

4. LOGGIE

LOGGIE uses attribute grammars for describing language semantics and graphic presentations. An attribute grammar [Knuth68] is an extension of a context-free grammar. To every grammatical symbol a finite set of attribute equations are attached. Attribute grammars provide a powerful and practically useful technique to describe programming language semantics and to implement programming tools.

4.1. Designing an editor

This section describes LOGGIE's meta-notation and how an editor is designed. A LOGGIE grammar consists of the following parts: an abstract syntax, attributes with semantic equations, garlands and pictures.

LOGGIE is layered in order to separate derivation and structure presentation. The Derivation Layer (DL) handles the design of grammars and the derivation of programs of such grammars. The Structure Presentation Layer (SPL) handles the presentation and the interaction between the user and derived programs.
4.1.1 Abstract syntax
The abstract syntax of the grammar is a context-free grammar which is a set of terminal and non-terminal symbols and a set of BNF productions. To each non-terminal symbol a syntax operator is associated for defining how the children to the node are derived (see figure 5.3). The following operators can be used: construction (.), list (\*), non empty list (+) and bag ([\*]), (e.g. see figure 5.3). The productions are defined in graphical EBNF, i.e. nodes and syntax operators which are connected to each other by links is a directed graph. Each non-leaf in the graph is a non-terminal symbol and each leaf is a terminal symbol. In figure 5.3 the SYSTEM-TEXT-AREA is a terminal and the BLOCK-INTERACTION-AREA is a non-terminal. The construction-operator of the SYSTEM-DIAGRAM derives one term of each right hand node and the bag-operator of the BLOCK-INTERACTION-AREA derives a mixed list of terms of the right hand nodes.

4.1.2 Garlands
Garlands connect nodes in syntax trees. Attribute relations can be defined along garlands. A garland specification has a source grammar node and a set of destination nodes. In figure 4.1 two garland specifications have been added (TO and FROM). The garland specification of CHANNEL-DEFINITION-AREA are represented by channels (a line with an arrow). In this example CHANNEL-DEFINITION-AREA9 is the source node, BLOCK2 and BLOCK3 are destination nodes.

Figure 4.1: A derived syntax graph with garlands.

4.1.3 Attributes
There are different types of attributes. A synthesized attribute is an attribute whose attribute arguments may be attributes from the associated derivation node, or attributes of the children of the associated node. An inherited attribute is an attribute whose arguments may only be attributes of the associated node or attributes of the parent or siblings of the associated node. A local attribute has no attribute arguments, its value is evaluated at creation time, and is not affected by changes of other attribute values. The value of local attributes can
be changed by a user. *Edit-semantic* attributes are the subset of attributes that define the presentation and interaction of a derived structure. They are used to specify the graphic interaction.

Some edit-semantic attributes are:

**PICTURE** defines picture specifications that a derivation node is associated with.

**ATTRIBUTES** defines those attributes that are to be displayed in a pictures.

**CONNECTION** specifies how garlands are mapped onto connectors (see 4.1.4).

**MENUS** defines the contents in a menu. Pictures are associated with menus.

### 4.1.4 Pictures

SPL handles the picture specification which is an internal format with different fields (see below). All derivation nodes can be associated with a graphic representation, a picture. A *picture* is described as two graphical data types: an *open figure* and a *folded figure*. An open figure shows the graphical substructures of the node. An open figure is just like an open window. As opposed to an open figure a folded figure does not show the sub-structures of the node. Folded figures are similar to icons in widow systems.

The figures are displayed on the screen as *views* of certain basic types:

- **Box** a rectangular area that can take any size.
- **Icon** a bitmap.
- **Connector** a polyline that can attach to other views.
- **String** a string of characters.
- **Formatted text** pretty-printed text.

Some of the *fields* which describe pictures are:

- **Image** how the picture is presented on the screen.
- **Placement** where the image can be placed in relation to the parent picture.
- **In- and Out-spots** where connectors can be attached.
Ornaments can be added to the picture, for example lines, text etc. Some of the associated attributes to the derivation nodes can be presented as ornaments on the image.

Input definitions describe the interaction with the picture.

Some graphical constraints can be placed on views. In the SDL application the following constraints have been used:

inside a view must always be inside its parent view

on the border a view must always be on the border of its parent.

5. The SDL application

This section describes how to map the SDL/GR syntax to LOGGIE's meta notation. The SDL/GR syntax defined in section 3 is mapped to the abstract grammar. The implementation of the subset of SDL/GR must be suited to the syntax of LOGGIE.

5.1. The abstract grammar

The specification of the abstract grammar:

\[
\begin{align*}
\text{SYSTEM-DIAGRAM} & \quad ::= \quad \text{BLOCK-INTERACTION-AREA} \\
& \quad \quad \quad \quad \text{SYSTEM-TEXT-AREA} \\
\text{BLOCK-INTERACTION-AREA} & \quad ::= \quad \text{BLOCK}* \quad \text{CHANNEL-DEFINITION-AREA}* \\
& \quad \quad \quad \quad \text{SIGNAL-LIST}* \quad \text{PORT}*
\end{align*}
\]

\[
\begin{align*}
\text{BLOCK} & \quad ::= \quad \text{PROCESS-INTERACTION-AREA} \\
& \quad \quad \quad \quad \text{SYSTEM-TEXT-AREA}
\end{align*}
\]

\[
\begin{align*}
\text{PROCESS-INTERACTION-AREA} & \quad ::= \quad \text{PROCESS-DIAGRAM}* \quad \text{SIGNAL-LIST}* \\
& \quad \quad \quad \quad \text{PORT}* \quad \text{CREATE-LINE-AREA}* \\
& \quad \quad \quad \quad \text{SIGNAL-ROUTE-DEFINITION-AREA}*
\end{align*}
\]

\[
\begin{align*}
\text{PROCESS-DIAGRAM} & \quad ::= \quad \text{START-AREA}* \quad \text{TASK-AREA}* \\
& \quad \quad \quad \quad \text{OUTPUT-AREA}* \quad \text{INPUT-AREA}*
\end{align*}
\]

\[
\begin{align*}
& \quad \quad \quad \quad \text{STATE-AREA}* \quad \text{RESET-AREA}* \\
& \quad \quad \quad \quad \text{SET-AREA}* \quad \text{PROCEDURE-CALL-AREA}* \\
& \quad \quad \quad \quad \text{CREATE-REQUEST-AREA}* \\
& \quad \quad \quad \quad \text{DECISION-AREA}* \\
& \quad \quad \quad \quad \text{PROCEDURE-DIAGRAM}* \\
& \quad \quad \quad \quad \text{CONNECTOR-ARROW}* \\
& \quad \quad \quad \quad \text{MULTI-CONNECTOR-ARROW}* \\
& \quad \quad \quad \quad \text{MULTI-CONNECTOR-LINE}* \\
& \quad \quad \quad \quad \text{STOP}* \quad \text{PROCESS-TEXT-AREA}*
\end{align*}
\]
All the symbols in the productions except the BLOCK-INTERACTION-AREA and the PROCESS-INTERACTION-AREA are presented on the screen and are associated with picture attributes, (see picture definition in section 4.1.4). BLOCK has two different display forms, an icon (closed figure), see figure 5.1 and a window (open figure), see figure 6.4. The substructure inside the block is shown in the open window, see figure 6.5.

![E1]

Figure 5.1: A block icon.

The PROCESS and PROCEDURE-DIAGRAM have also two display forms like the block, see figure 6.7 and 6.9. The graphical symbols for the START-AREA, TASK-AREA, OUTPUT-AREA etc. are shown in section 3.4. CHANNEL-DEFINITION-AREA are associated with a picture, which is like the channel-symbol-one in the SDL/GR syntax (see figure 6.2). For example, the create-request picture definition is a box with three ornaments. The local attribute for the text inside the symbol and the two lines are presented as ornaments (see figure 5.2)

![create-body]

Figure 5.2: Create-request symbol.

There can only be one START-AREA in a PROCESS-DIAGRAM and in a PROCEDURE-DIAGRAM even if the abstract grammar specifies that there can be several START-AREAS. There is a restriction on the START-AREA, it can not be added or deleted.

In figure 5.3, a part of the SDL grammar is displayed in graphical form. The stars around the node means that the sub-tree is not shown. The construction-operator, (.), of the SYSTEM-DIAGRAM derives one term of each right hand node, (BLOCK-INTERACTION-AREA and SYSTEM-TEXT-AREA) and the
bag-operator, (l*), of the BLOCK-INTERACTION-AREA derives a mixed list of terms of the right hand nodes.

![Diagram](image)

**Figure 5.3:** A part of the abstract grammar in graph form.

In figure 5.4 a derived syntax graph for the abstract grammar in figure 5.3 is shown.

![Diagram](image)

**Figure 5.4:** A derived syntax tree.

### 5.2. Interaction with the user

The interaction with the pictures is defined with the input-definitions field of the pictures (see section 4.1.4). These fields specify which operations that can be performed on the node and how the operations are accessed, for example via pop-up menus, fix menus or function calls.

All the derivation nodes that are displayed on the screen are associated with an attribute called *button-items* which contains a specification for a pop-up menu. This menu appears when the middle mouse button is pressed at a
graphical symbol representing the derivation node. As an example when the user presses the middle mouse button in the create-request symbol the menu in figure 5.5 appears.

![Figure 5.5: Create-request symbol menu.](image)

This menu contains some commands that can be applied to the create-request symbol.

- **remove**: deletes the create-symbol from the process diagram.
- **connect**: connects the create-symbol to another symbol.
- **edit**: edits the create-body in a text editor.

The user can change the size of the pictures interactively. The picture can be changed so all the text or only a part of the text inside is shown. The text inside the pictures can be edited in a text editor.

When a picture is selected with the right mouse button a default menu appears. Two examples are shown in figure 5.6.

![Figure 5.6: Right mouse button default menus.](image)

If a picture is selected with the left mouse button, the picture can be moved interactively. All the graphical symbols have this behaviour associated to the left button. When a symbol which is connected to another symbol is moved, the connector is stretched out like a rubber-band.

### 5.3. Connector

The derivation node CONNECTOR-ARROWS, in figure 5.7, is a connector from one picture to another (e.g. figure 5.8). The following grammar nodes are displayed as connectors: "channel definition area", "signal route definition area", "connector line" and "connector arrow". The garland FROM has been
connected to derivation node START-AREA5 and the garland TO has been connected to derivation node OUTPUT-AREA2. The connector in fig 5.8 depicts the garland connections of figure 5.7.

![Diagram](image)

**Figure 5.7**: A part of a derived syntax graph with garlands.

![Diagram](image)

**Figure 5.8**: Two connected symbols.

### 5.4. Multi-connector

A multi-connector looks like a fork. It is a connector from one picture to many, many to many or many to one (as shown in figure 5.10). The following grammar nodes are displayed as multi-connectors: "multi connector arrow" (a multi connector with an arrow) and "multi connector line" (without an arrow).

In figure 5.9 the MULTI-CONNECTOR-LINE has a CONNECTION attribute, which contains two lists of garland identifiers (FROM) (TO1 TO2 TO3) indicating how the multi-connector shall be connected. The garland FROM has been connected to derivation node STATE-AREA3, the garland TO1 has been connected to derivation node INPUT-AREA2 and so on. The multi-connector in figure 5.10 depicts the garland connections of figure 5.9.
Figure 5.9: A part of a derived syntax graph with garlands.

Figure 5.10: A multi-connector.

The button-items menu at the multi-connector has a command "connect", which add a new symbol to the multi-connector. The user selects a symbol among the allowed ones, which are highlighted.

5.5 Yacc specification
The text in the graphical symbols are defined as local attributes at the derivation node and they are displayed in association with the pictures. A parser can be defined which is used when the value is entered. For each local attribute, an editor type can be specified. The text is defined in SDL/PR syntax (see the Appendix, section 5). Only a subset of the syntax can be entered as text in the graphical symbols.

LOGGIE can interact with YACC (Yet Another Compiler-Compiler), the famous parser generator. A Yacc specification is specified together with a lexical analyzer, Lex, which generate a parser. The SDL/PR syntax in each symbol is mapped to BNF productions in a Yacc specification.
The syntax is only checked. There are no semantics control between the text in the symbols.

Every time the attribute is changed the Yacc parser is invoked to check the syntax. If the syntax is incorrect the user gets a message like: "Yacc syntax error: "text string" Try again!" - and the text can be edited again. No help is given to find the text error.

6. The SDL Editor

This section will show simple examples on how the SDL-editor can be used. Some of the examples are fetched from [SDL1] and [SDL2]. When the SDL-editor is started the window in figure 6.1 is opened. Initially it contains only a text-symbol.

![System window](image)

Figure 6.1: A system window.

The menu attached to the window in figure 6.1 shows the different operations that can be performed on a system. The menu can also be displayed by clicking inside the system frame with the middle mouse button.

The system heading, the keyword **SYSTEM** followed by the identifier, is displayed in the upper left corner in the system frame. The user can select the command "Edit system id" in the menu. A string editor appears and a new identifier can be input. The user can also point directly on the <id> symbol and enter the identifier.

The blocks are presented as icons when they are added and they can be connected to each other via channels. Blocks can also be connected to the environment (frame). The user selects the command "Add channel to frame" or "Add channel from frame" and selects the block target. A small box is added to the frame so that the connection point can be moved. If the command "Add
channel" is chosen, it is possible to connect a block to a connection point on the border or to another block. The channel C1 in figure 6.2 is connected to the environment.

A signal list picture is always added together with a channel. The signal list defines the signal types which are allowed to be conveyed by the associated channel. The user places the signal list picture close to the channel (see figure 6.2).

![Figure 6.2: An example of a system specification.](image)

When blocks are selected with the middle button a menu appears (see figure 6.3). If the command "open" is choosen, a window is opened and the substructure of the block is shown, allowing the user to edit the block structure.

![Figure 6.3: A block menu.](image)

The block heading, the keyword BLOCK followed by the identifier is displayed in the upper left corner in the block frame (see figure 6.4). The user can select the command "Edit block id" in the menu. A string editor appears and a new block name is typed. The user can also point directly on the <id> symbol and type the identifier.
The menu for the block is shown in figure 6.4. In figure 6.5 some processes have been added to the block. These processes can be connected to each other via signal routes. Processes can also be connected to the environment, the border. The user selects the command "Add a channel to frame" or "Add a channel from frame" and selects a process which is going to be connected to the frame. A little connection point is also added to the frame as described above. The result is shown in figure 6.5. The signal routes R1 and R2 are leading from the environment. The channel C1 is connected to R1 and R2. The connection point with the identifier C1 indicates that the channel C1, from the environment, is connected to the signal routes R1 and R2. A signal list is also added together with a signal route (same as for a channel, see above).

Another type of connection is shown in figure 6.6. It is an example of a block with create line symbols, a dashed line between processes. A create line symbol specifies that the process at the originating end will create the process type at the terminating end, i.e. the process father creates the processes son1 and son2.
Figure 6.6: An example of a block diagram with create line symbols.

The process has a pop-up menu like the block (see figure 6.7). The "open" command opens the process icon. In the upper left corner the process heading, the keyword PROCESS followed by the identifier and the keyword FPAR followed by the parameters are shown (see figure 6.9).

Figure 6.7: Process menu.

When the user selects the command "Edit parameters". A text editor appears and the parameters can be entered. The user can also point directly at the symbol <parameters> and enter the parameters in a text editor. The syntax for the parameters are checked by a parser. The process name can be edited in the same way as the parameters with the command "Edit id". The menu in figure 6.8 shows the symbols that can be added.

Figure 6.8: Processesymbols.
The symbols can be connected to each other by transitions. All the symbols have a pop-up menu, like the one in figure 5.5. Each symbol knows to which other symbols it can be connected. When the user selects the "connect" command the allowed symbols are highlighted and a desired symbol is chosen to be connected. In figure 5.8 the start-symbol is connected to the output-symbol.

The text in all symbols in the process can be edited with the command "Edit". The text is entered with a text editor and parsed. If the syntax is incorrect the user gets a message like: - "Yacc syntax error: "text string" Try again!" - and the text can be edited again. No help is given to find the text error.

Figure 6.9 shows a process diagram with a transition of symbols which are connected to each other. In the process diagram there must be one start symbol and at least one stop symbol.

Figure 6.10 shows a process body with a reference to a procedure. In the upper left corner a procedure reference is placed, "TermP". "TermP" is invoked in the process body. This procedure reference can be opened with the command "Open" in the pop-up menu. The procedure body is shown in a window, like the process diagram, and the user can edit the procedure.

**Figure 6.9:** A process diagram.
The static semantic control is limited. There are no semantic controls between the text in different graphical symbols. For example, there is no control that a procedure call in a diagram has a procedure reference (see figure 6.10). There are some controls when a user is editing a process or procedure diagram. If the diagram is not complete a little "ERROR" button is shown (see figure 6.11). For example, a process diagram is incomplete if there is no stop symbol to terminate the process and a procedure diagram must contain a return symbol. A diagram is also incomplete if there are symbols which are not connected to each other.

If a procedure is incomplete, an error chain begins. The process which invokes the procedure and the block which includes the process is wrong. So at the end the whole system is incomplete.
Figure 6.11: An incorrect process diagram.

7. Conclusions and future works

The report briefly describes the Specification and Description Language, SDL, and the subset of SDL that has been implemented in LOGGIE. Furthermore a short description about LOGGIE and the implementation of the SDL editor is presented.

This work shows that it is possible to map the SDL/GR syntax to LOGGIE's meta-notation. The current editor is only a prototype and cannot be used to define full SDL specifications.

The static semantic control is limited in the process and procedure diagrams. For example, an error is not propagated upwards in the structure and a process error is not shown at block level. The static semantics syntax is more complex than the graphical syntax.

There are some limitations in the graphical support of LOGGIE. A connector cannot be connected to another connector and a connector cannot easily be associated to another connector in different levels in a specification. For example, if a block reference is open inside a system frame, the channels in the system are not coupled to the signal routes inside the block at a common point. A multi-connector can only connect a predefined number of objects. In this implementation state and decision symbols can be connected to up to ten other symbols.
To make the SDL editor useful, the full SDL/GR syntax must be implemented. Some examples that could be further developed are: The structural concepts in SDL ([SDL1], section 3); some part in section 4 [SDL1], additional concepts in SDL, for example Macro definition ([SDL1], section 4.2) and Service definition ([SDL1], section 4.10). The limitations discussed above must also be studied and solved.
References


Appendix

The subset of the SDL/GR syntax

A strict subset of the original syntax has been chosen. The whole grammar, both SDL/GR and SDL/PR syntax, are presented in [SDL1]. All the terminals in the grammar which include the word symbol are graphical symbols.

Section 1.

The syntax for system diagram:

\[
\text{system diagram} ::= \text{frame symbol} \text{ contains} \\
\quad \text{( system heading)} \\
\quad \text{system text area} \\
\quad \text{block interaction area)}
\]

\[
\text{system text area} ::= \text{text symbol} \text{ contains} \\
\quad \{\text{signal definition}\} \\
\quad \{\text{signal list definitions}\}*
\]

\text{system heading}, \text{signal definition} and \text{signal list definitions} are defined in the SDL/PR syntax. \text{block interaction area} is defined in section 2.

Section 2.

The syntax for block diagram

\[
\text{block interaction area} ::= \{\text{block area}\} \\
\quad \{\text{channel definition area}\}+
\]

\[
\text{block area} ::= \text{graphical block reference} \\
\quad \text{block diagram}
\]

\[
\text{graphical block reference} ::= \text{block symbol} \text{ contains} \\
\quad \text{block name}
\]

\[
\text{block diagram} ::= \text{frame symbol} \text{ contains} \\
\quad \{\text{block heading}\} \\
\quad \{\text{block text area}\} \\
\quad \{\text{process interaction area}\}
\]

\[
\text{block text area} ::= \text{system text area}
\]

\[
\text{channel definition area} ::= \text{channel symbol} \text{ contains} \\
\quad \text{is associated with} \\
\quad \{\text{channel name}\} \\
\quad \{\text{[\text{channel identifier}\} \\
\quad \{\text{block identifier}\}]} \\
\quad \text{signal list area} \\
\quad \{\text{signal list area}\} \text{ set} \\
\quad \text{is connected to} \\
\quad \{\text{block area}\}
\]

-29-
<block area>
    | <frame symbol>>set

<signal list area> ::= <signal list symbol>
                        contains <signal list>

The <block heading> and <signal list> are defined in section 5. <process interaction area> is defined in section 3. <system text area> is defined in section 1.

Section 3.
The syntax for process diagram:

<process interaction area> ::= { <process area>
    | <create line area>
    | <signal route
definition area> }+

<process area> ::= <graphical process reference>
    | <process diagram>

<graphical process reference> ::= <process symbol> contains
                                {<process name>}

<create line area> ::= <create line symbol>
                        is connected to
                                [<process area>
                                <process area>]

<signal route definition area> ::= <signal symbol>
                                    is associated with
                                        [<signal route name>
                                        [[<channel identifier>]]
                                        <signal list area>
                                        [<signal list area>] } set
                                    is connected to
                                        [<process area>
                                        [<process area>
                                        [<frame symbol>>] } set

<process diagram> ::= <frame symbol> contains
                          {<process heading>
                            { [<process text area]>}*
                            {<graphical procedure
                            reference>}*
                            {<procedure diagram>}*
                            {<process graph area>} set }*
                            [is associated with
                            [<signal route identifier> ]+]

<process text area> ::= <text symbol> contains
                          {<signal definition>
                          | <signal list definition>
                          | <variable definition> }*
::= <start area>

<start area> ::= <start symbol> is followed by
transition area

<transition area> ::= [<transition string area>]
| state area
| nextstate area
| decision area
| stop symbol
| return symbol

<transition string area> ::= <task area>
| output area
| create request area
| procedure call area
| set area
| reset area

| is followed by
transition area

<next state area> ::= state symbol contains
nextstate body

<decision area> ::= decision symbol contains
question is followed by
<outlet1> { <outlet1> | <outlet2> } <outlet1>* set

<outlet1> ::= flow line symbol
is associated with answer
is followed by
transition area

<outlet2> ::= flow line symbol
is associated with ELSE
is followed by
transition area

<task area> ::= task symbol contains
<task body>

<output area> ::= output symbol contains<output body>

<create request area> ::= create request symbol
contains<create body>

<procedure call area> ::= procedure call symbol
contains<procedure call body>

<set area> ::= task symbol contains
<set>

<reset area> ::= task symbol contains
<reset>
<state area> ::= <state symbol> contains
       <state list>
is connected to
       {<input association area>
         | <save association area>}#

<input association area> ::= <solid association symbol>
is connected to <input area>

<save association area> ::= <solid association symbol>
is connected to <save area>

<input area> ::= <input symbol> contains
       <input list>

<save area> ::= <save symbol> contains
       <save list>

<process heading>, <signal definition>, <signal list definition>, <variable
definition>, <nextstate body>, <question>, <answer>, <task body>,
<output body>, <create body>, <procedure call body>, <set>, <reset>,
<state list>, <input list> and <save list> are defined in section 5.

Section 4.
The syntax for the procedure diagram:

<procedure diagram> ::= <frame symbol> contains
       {<procedure heading>
        {<procedure text area>
         |<graphical procedure reference>
         |<procedure diagram>*
        } set
       }

<procedure text area> ::= <text symbol> contains
       {<variable definition>}*

<graphical procedure reference> ::= <procedure symbol> contains
       <procedure name>

<procedure graph area> ::= <procedure start area>

<procedure start area> ::= <procedure start symbol> is followed by
       <transition area>

<variable definition> is defined in section 5 and <transition area> is defined
in section 3.
The subset of the SDL/PR syntax

Section 5.

<system heading> ::= 'SYSTEM' <system name>

<block heading> ::= 'BLOCK' <block name>
                    |<block identifier>

<create body> ::= <process identifier>
                    |<actual parameters>

<actual parameters> ::= '(' <expression list> ')' 

<expression list> ::= [<expression>] <expression list>

<expression> ::= <operand0> 
                | <sub expression> '->' <operand0>

<sub expression> ::= <expression>

<operand0> ::= <operand1> 
             | <sub operand0> {'OR'|'XOR'} <operand1>

<sub operand0> ::= <operand0>

<sub operand0> ::= <operand2> 
                    | <sub operand1> 'AND' <operand2>

<sub operand1> ::= <operand1>

<operand2> ::= <operand3> 
             | <sub operand2> 
                    {'='|'<='|'>='|'>|'<|'<='|'IN'}

<sub operand2> ::= <operand2>

<operand3> ::= <operand4> 
             | <sub operand3> {'+'|'-'|'/'} <operand4>

<sub operand3> ::= <operand3>

<operand4> ::= <operand5> 
             | <sub operand4> {'*'|'/'|'MOD'|'REM'} <operand5>

<sub operand4> ::= <operand4>

<operand5> ::= ['NOT'|'-'] <primary>

<primary> ::= <identifier> 
            | <identifier> <expression list> 
            | <conditional expression> 
            | '(' <expression> ')' 

<conditional expression> ::= 'IF' <expression> 'THEN' <expression> 'ELSE' <expression> 'FI'

<input list> ::= <stimuli identifier> 
                { <stimuli identifier> }*
<stimuli identifier> ::= \([<\text{signal identifier}>\]
\)(['([<\text{variable identifier}>]
\)\([<\text{variable identifier}>]\)])\)*')
\| \(<\text{timer identifier}>\)

<output body> ::= \(<\text{signal identifier}>\)
\([<\text{actual parameters}>]\)
\)([,\(<\text{signal identifier}>\)
\)([<\text{actual parameters}>])\)*
\)(['TO' \(<\text{pid expression}>\)])

<pid expression> ::= 'SELF'
\| 'PARENT'
\| 'OFFSPRING'
\| 'SENDER'
\| \(<\text{expression}>\)

<procedure call body> ::= \(<\text{procedure identifier}>\)
\([<\text{actual parameters}>]\)

<variable definition> ::= 'DCL'\| 'REVEALED'\| 'EXPORTED'
\| 'REVEALED EXPORTED'
\| 'REVEALED EXPORTED'
\(<\text{variable name}>\)
\)([,\(<\text{variable name}>\)]\)* \(<\text{sort}>\)';'

<signal definition> ::= 'SIGNAL' \(<\text{signal name}>\)
\([<\text{sort list}>]\)
\)([,\(<\text{signal name}>\)\([<\text{sort list}>]\)])\)*

<sort list> ::= '('\(<\text{sort}>\)([,\(<\text{sort}>\)])*')'

<signal list definition> ::= 'SIGNALLIST' \(<\text{signal list name}>\)
\('=' \(<\text{signal list}>\)

<signal list> ::= \(<\text{signal item}>\)
\)([,\(<\text{signal item}>\)]\)*

<signal item> ::= \(<\text{signal identifier}>\)

<view definition> ::= 'VIEWED' \(<\text{variable identifier}>\)
\)([,\(<\text{variable identifier}>\)]\)* \(<\text{sort}>\)
\)([,\(<\text{variable identifier}>\)\([<\text{sort}>]\)])\)*

<timer definition> ::= 'TIMER' \(<\text{timer name}>\)
\([<\text{sort list}>]\)
\)([,\(<\text{timer name}>\)\([<\text{sort list}>]\)])\)*

<reset> ::= 'RESET' '(' \(<\text{reset statement}>\)
\)([,\(<\text{reset statement}>\)]\)* ')'

<reset statement> ::= \(<\text{timer identifier}>\)
\)('\(<\text{expression list}>\)')'

<set> ::= 'SET' \(<\text{set statement}>\)
\)([,\(<\text{set statement}>\)]\)*

<set statement> ::= '('\(<\text{expression}>\),\(<\text{timer identifier}>\)
\)('\(<\text{expression list}>\)')')'
<save list> ::= <signal list>
<state list> ::= {<state name>{, <state name>}}*
<next state body> ::= <state name>
<task body> ::= {<assignment statement>
[, <assignment statement>]}*
<assignment statement> ::= <variable identifier> ':='
<expression>
<procedure heading> ::= 'PROCEDURE' { <procedure name>
                  |<procedure identifier>}
<procedure formal parameters> ::= 'FPAR' <formal variable parameter>
                               [,<formal variable parameter>]*
<formal variable parameter> ::= ['IN/OUT' | 'IN']
                              <variable name>
                              [,<variable name>]*<sort>
                              [,<variable name>[, <variable name>]*
                              <sort>]*
<process heading> ::= 'PROCESS'
<process name><process identifier>
<process parameters> ::= 'FPAR' <variable name>
<variable name>
[,<variable name>]*<sort>
[,<variable name>[, <variable name>]*
<sort>]*
<question> ::= <question expression>
|<informal text>
<question expression> ::= '(' <expression> ')'
<informal text> ::= <character string>
<answer> ::= '('<range condition>[<informal text>]')'
<range condition> ::= {<closed range><open range>
[,<closed range><open range>]*
<closed range> ::= <constant> ':' <constant>
<open range> ::= <constant>
|{'='|'M'='|'<'|'>'}| '<'|'>='| '='|'>')' <constant>
<identifier> ::= [<qualifier]><name>
<qualifier> ::= <path item>{'/'<path item>}*
<path item> ::= <scope unit class> <name>
<scope unit class> ::= 'SYSTEM' | 'BLOCK' |
                   'PROCESS' | 'PROCEDURE'