



SDL Suite Getting Started

Getting Started

Introduction	xiv
1. Introduction to Languages and Notations.....	1
Benefits of a Specification Language.....	2
General about the SDL Language	3
Modularity	3
Object Oriented Design	3
Graphical and Textual Notations	4
Application Areas.....	4
More about SDL.....	4
Theoretical Model	4
Structure	5
Communication	6
Behavior	7
Data.....	7
Type Concept	8
The Message Sequence Chart Language	9
History	9
Plain MSC	9
High-Level MSC	10
Graphical and Textual Notations	11
Application Areas.....	11
Object Model Notation.....	12
Class	12
Relations and Multiplicity	13
Objects	14
State Chart Notation	15
State	15
Transition	15
Start and Termination Symbol	16
Substates.....	16
ASN.1 – Abstract Syntax Notation One.....	17
The TTCN Notation	17
TTCN – Tree and Tabular Combined Notation	18
Tool Support.....	18
References	19
2. Introduction to the SDL Suite.....	21
About the SDL Suite.....	22

IBM Rational	22
The SDL Suite	22
Overview of the SDL Suite	23
Architecture	23
Starting the SDL Suite tools	23
Batch Facilities	25
Licensing Mechanism	25
Common Tools	26
The SDL Suite Graphical Tools	27
Other SDL Suite Tools and Back-End Facilities	28
Information Management	32
SDL Diagrams	32
MSC Diagrams	33
High-Level MSC Diagrams	33
Textual SDL and MSC Formats	33
Object Model Diagrams	34
State Chart Diagrams	34
Text Documents	34
The System File	34
The Link File	35
Control Unit Files	35
Source Management	36
Target Management	36
PCs and Workstations	37
User Interface	37
Supported UNIX Systems	37
3. Tutorial: The Editors and the Analyzer	39
Purpose of This Tutorial	40
The Demon Game	41
Behavior of the Demon Game	41
Starting the SDL Suite	42
Some Preparatory Work	42
Starting the SDL Suite	43
The Organizer Window	44
Preferences	46
What You Will Learn	46
What Are Preferences for?	46
Displaying and Changing Preferences	46
Help Preferences	47
Setting the Default Printer	48

Setting the Drawing Area Size	49
Saving the Preferences	49
Creating an SDL Structure	51
What You Will Learn	51
Customizing the Organizer Chapters	51
Creating a System Diagram	54
Saving the Newly Created System Diagram	66
Saving the Diagram Structure	70
More About Saving	71
Printing the System Diagram	72
What You Will Learn	72
How to Print	72
Checking the System Diagram	74
What You Will Learn	74
Running the Analyzer	74
Looking for Analysis Errors	76
Correcting Analysis Errors	77
Creating a New Block Diagram	79
What You Will Learn	79
Creating a Block Diagram from the Organizer	79
Editing the Block Diagram	83
Working with Multiple Diagrams	88
Working with Multiple Windows	89
Resulting Organizer View	91
Checking the Syntax of the Block Diagram	91
Creating a Block Diagram From a Copy	91
What You Will Learn	91
Creating the Block DemonBlock	92
Creating a Process Diagram	96
Editing the Process Demon	96
Editing the Process Game	103
Editing the Process Main	106
More About the Organizer	107
What You Will Learn	107
Tree View	108
Expand / Collapse	108
Rearranging Diagrams	109
Diagram Pages	110
Printing the System	110
Analyzing the Complete System	112

What You Will Learn	112
Enabling Semantic Analysis	112
Managing Message Sequence Charts	114
What You Will Learn	114
Inserting an MSC into the Organizer	114
Editing an MSC	118
Using the Index Viewer	124
What You Will Learn	124
Starting the Index Viewer	124
Finding a Definition	125
Finding References	127
So Far	128
Appendix A: The Definition of the SDL-88 DemonGame	129
Appendix B: The MSC for the DemonGame	134
4. Tutorial: The SDL Simulator	135
Purpose of This Tutorial	136
Generating and Starting a Simulator	137
What You Will Learn	137
Generating the Simulator	137
Starting the Simulator	139
Executing Transition by Transition	141
What You Will Learn	141
Executing the Start Transitions	141
Sending Signals from the Environment	143
Viewing the Internal Status	148
What You Will Learn	148
Restarting the Simulator	148
Viewing Process and Signal Queues	149
Viewing Variables and Process Instances	151
Other Viewing Options	153
Dynamic Errors	153
What You Will Learn	153
Finding a Dynamic Error	153
Using Different Trace Values	155
What You Will Learn	155
Setting Trace Values	155
Executing Symbol by Symbol	157
Hiding Uninteresting Transitions	158
Looking at the External Behavior	159

What You Will Learn	159
Setting Trace and Signal Logging	159
Adding Buttons for Common Commands	160
Playing the Game	161
Examining the Signal Log File	162
Using Breakpoints	163
What You Will Learn	163
Setting Up the System	163
Setting a Symbol Breakpoint	164
Setting a Transition Breakpoint	165
Changing the System	168
What You Will Learn	168
Some Preparations	169
Creating a Process	170
Changing the State of Timers	172
Generating Message Sequence Charts	173
What You Will Learn	173
Initializing the MSC Trace	173
Tracing the Execution in the MSC	175
Trace-Back to SDL Diagrams	179
Ending the MSC Trace	180
The Coverage Viewer	181
What You Will Learn	181
Starting the Coverage Viewer	181
Using the Coverage Viewer	181
Augmenting the Coverage	184
Looking at Coverage Details	185
Exiting the Simulator UI	186
So Far.....	186
5. Tutorial: The SDL Explorer	187
Purpose of This Tutorial	188
Generating and Starting an SDL Explorer	189
What You Will Learn	189
Quick Start of an SDL Explorer	190
Basics of an SDL Explorer	192
Navigating in a Behavior Tree	193
What You Will Learn	193
Setting Up the Exploration	193
Using the Navigator	194
More Tracing and Viewing Possibilities	200

What You Will Learn	200
Using the View Commands	200
Using MSC Trace	201
Going to a State Using Path Commands	202
Validating an SDL System	203
What You Will Learn	203
Performing a Bit State Exploration	203
Examining Reports	205
Exploring a Larger State Space	207
Restricting the State Space	210
Checking the Validation Coverage	212
Going to a State Using User-Defined Rules	213
Performing a Random Walk	215
Verifying a Message Sequence Chart	216
What You Will Learn	216
Verifying a System Level MSC	216
Exiting the SDL Explorer UI	220
Using Test Values	221
What You Will Learn	221
Using the Automatic Test Value Generation	221
Changing the Test Values Manually	224
Exiting the SDL Explorer	225
So Far	225
6. Tutorial: Applying SDL-92 to the DemonGame	227
Purpose of This Tutorial	228
Applying SDL-92 to the DemonGame	229
Some Preparatory Work	229
Creating a Process Type from a Process	233
What You Will Learn	233
Changing into a Process Type	233
Inserting Gates and Virtual Transitions	236
The Organizer Structure	239
Redefining the Properties of a Process Type	240
What You Will Learn	240
The Process Type JackpotGame	240
Changes to the Block GameBlock	242
Changes to Process Main and System DemonGame	243
Simulating the JackpotGame	244
Adding Properties to a Process Type	246

What You Will Learn	246
The Process Type DoubleGame	246
Simulating the DoubleGame	249
Combining the Properties of Two Process Types	250
What You Will Learn	250
Working with the Type Viewer	250
How to Work-Around the Lack for Multiple Inheritance	252
Using Packages and Block Types	255
What You Will Learn	255
Package – a Reusable Component	255
Creating a Package	256
Using a Package	259
Reusing Packages	260
What You will Learn	260
The Package AdvancedFeatures	261
Block Type AdvancedGameBlock	262
Redefined Process Type Main	263
Creating the System AdvancedDemonGame	264
Conclusion	265
More Exercises...	265
Appendix: Diagrams for the DemonGame Using Packages	266
7. Cmicro Targeting Tutorial	269
Prerequisites / Abbreviations Used	270
Introduction	271
General	271
Integrations	271
Target Tester Communication	271
Prerequisites to the Example	272
The Pager System	272
Delivered Files	273
Targeting	274
Preparations - File Structure	274
Using the Targeting Expert	274
Step 1: Select the Desired Component	275
Step 2: Select the Type of Integration	277
Step 3: Configure the Build Process	287
Step 4: Make the Component	288
Use of the SDL Target Tester	289
Differences between SDL Simulator and SDL Target Tester	289

Restrictions in this Tutorial	289
Testing the Pager System	290
Run Target EXE without Tester	294
8. Tutorial: Using ASN.1 Data Types	297
Introduction	298
Implementation of ASN.1	298
Abstract Syntax	299
Transfer Syntax	299
Creating the Abstract Syntax	300
Adding ASN.1 Modules to your Project	300
Importing ASN.1 Modules	303
Assigning Values to the Data Types	305
Creating the Transfer Syntax	307
Introduction	307
Generating Template Files - the Organizer	311
Editing the Generated Files - the Organizer	313
Generating Template Files - Targeting Expert	318
Editing the Generated Files - Targeting Expert	319
Compiling Your Application	321
Using the edited files - Organizer	321
Using the edited files - Targeting Expert	322
Running Your Application	323
Appendix A	324
9. Tutorial: Using SDL-2000 features	331
Purpose of this Tutorial	332
Introduction	332
Support in the SDL Suite	332
Why SDL-2000?	332
Graphical Design of Data Types	333
Class Symbols	333
Association and Aggregation Lines	333
Creating an SDL Structure	336
Working with Class Symbols	336
Working with Lines	337
Moving Symbols and Lines	338
Saving the Diagram	338
Editing a Diagram	339
Case Sensitivity	340

Limitations	340
Edit the Diagram.	340
Viewing a Class	342
Limitations	342
View the Definition	342
Textual Algorithms.	343
Textual Algorithms.	343
Operators without Parameters and Operators without Return Results.	343
Limitations	344
An Example of Using Class Symbols.	344
10. Tutorial: Threaded Integration	347
Introduction	348
Prerequisites	348
Description of Example System.	349
The SDL System.	349
The Target Application.	349
Preparations	352
Copy the Example System	352
Open the System.	352
Drawing a Deployment Diagram	353
What You Will Learn.	353
Starting the Deployment Editor	353
Deploying an SDL System	353
Using the Targeting Expert.	358
What You Will Learn.	358
Starting the Targeting Expert	358
Selecting Target Platform.	358
Configuring C Code Generation.	360
Compiling and Linking.	363
The Target System	366
Running the System	367
What You Will Learn.	367
An Overview of the System	367
Using the System	368

IBM Rational SDL Suite 6.3

Getting Started

This edition applies to IBM Rational SDL Suite 6.3 and IBM Rational TTCN Suite 6.3 and to all subsequent releases and modifications until otherwise indicated in new editions.

Copyright Notice

© Copyright IBM Corporation 1993, 2009.

US Government Users Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send written license inquiries to the following:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send written inquiries to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions. Therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation
1 Rogers Street
Cambridge, Massachusetts 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Additional legal notices are described in the `legal_information.html` file that is included in your software installation.

Copyright License

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_.

Trademarks

See <http://www.ibm.com/legal/copytrade.html>.

IBM, the IBM logo, and `ibm.com` are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.html.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

Introduction

About this Manual

This volume, [Getting Started](#), contains a guide that will assist you in getting familiar with the SDL Suite environment.

We start by giving an introduction to the SDL Suite and the supported languages. Then follows a number of chapters with hands-on tutorials on how to use the tools included in the SDL Suite, including a first look at object-oriented analysis and design. These tutorials use the same simple example (DemonGame), and can be followed without any previous knowledge of the SDL Suite.

The remaining chapters contain a number of more advanced tutorials on how to use ASN.1 data types and Cmicro targeting in practice.

All tutorials build on the knowledge gained in the previous tutorials, so you should practice them in the order they are included.

Documentation Overview

A general description of the documentation can be found in [“Documentation” on page viii in the Release Guide](#).

Typographic Conventions

The typographic conventions that are used in the documentation are described in [“Typographic Conventions” on page x in the Release Guide](#).

How to Contact Customer Support

Detailed contact information for IBM Rational Customer Support can be found in [“How to Contact Customer Support” on page iv in the Release Guide](#).

Introduction to Languages and Notations

This chapter begins with a brief introduction to SDL; the language, its history, its main concepts and application areas.

Next follows an introduction to the MSC language (including High level MSC), the Object Model notation and State Chart notation, and the ASN.1 notation.

For the sake of completeness, we have included a brief introduction to the TTCN notation. More information can be found in [chapter 1, *Introduction to Languages and Notations, in the Getting Started*](#).

After reading this chapter, you may want to deepen your knowledge of SDL. In [chapter 1, *Object Oriented Design Using SDL, in the Methodology Guidelines*](#), you will find information about how to take advantage of the SDL-92 language in an SDL Suite environment.

Also, a list of recommended literature dealing with various language topics is enclosed at the end of this chapter; see [“References” on page 19](#).

Benefits of a Specification Language

It is widely accepted that the key to successfully developing a system is to produce a thorough system specification and design. This task requires a suitable specification language, satisfying the following needs:

- A well-defined set of concepts.
- Unambiguous, clear, precise, and concise specifications.
- A basis for verifying specifications with respect to completeness and correctness.
- A basis for determining whether or not an implementation conforms to the specifications.
- A basis for determining the consistency of specifications relative to each other.
- Use of computer-based tools to create, maintain, verify, simulate and validate specifications.
- Computer support for generating applications without the need of the traditional coding phase.

The SDL Suite fulfills all the demands outlined in the list above.

General about the SDL Language

SDL (Specification and Description Language) is a standard language for specifying and describing systems¹. It has been developed and standardized by ITU-T in the recommendation Z.100.

The development of SDL started in 1972 after a period of research work. The first version of the language was issued in 1976 and it has been followed by new versions every fourth year. The latest versions expanded the language considerably, and today SDL is a “complete” language in all senses.

In the SDL Suite, there is full support of SDL including some of the SDL-96 concepts. For more information about the SDL support in the SDL Suite, see [“Compatibility with ITU SDL” on page 2 in chapter 1, Compatibility Notes, in the Release Guide.](#)

Modularity

An SDL specification/design (a system) consists of a number of interconnected modules (blocks). A block can recursively be divided into more blocks forming a hierarchy of blocks. The channels define the communication paths through which the blocks communicate with each other or with the environment. Each channel usually contains an unbounded FIFO queue that contain the signals that are transported on the channel. The behavior of the leaf blocks is described by one or more communicating processes. The processes are described by extended finite state machines.

Object Oriented Design

SDL furthermore supports object-oriented design by a type² concept that allows specialization and inheritance to be used for most of the SDL concepts, like blocks, processes, data types, etc. The obvious advantage is the possibility to design compact systems and to reuse components which in turn reduces the required effort to maintain a system.

-
1. No distinction is made in SDL between the terms “specification” and “description”, although they generally have different meanings in SDL applications.
 2. SDL has adopted the term *type* which corresponds to the term *class* used in many of the OO notations and programming languages.

Graphical and Textual Notations

SDL gives a choice of two equivalent syntactic forms; a Graphical Representation (SDL/GR) and a textual Phrasal Representation (SDL/PR). The SDL Suite supports both notations.

Application Areas

Currently, SDL is mainly known within the telecommunication industry, but it also has broader areas of application and is now gaining acceptance within the real-time software industry. The application areas may be summarized as follows:

- Type of system described by SDL: Real-time, interactive, distributed.
- Type of information provided by SDL: Behavior and structure.
- Level of abstraction supported by SDL: From system overview to functional detail.

More about SDL

Theoretical Model

The basic theoretical model of an SDL system consists of a set of extended finite state machines (FSM) that run in parallel. These machines are independent of each other and communicate with discrete signals.

An SDL system consists of the following components:

- [Structure](#)
 - hierarchical decomposition with system, block, process, and procedure as the main building blocks
- [Communication](#)
 - asynchronous signals with optional signal parameters
 - remote procedure calls for synchronous communication
- [Behavior](#)
 - processes
- [Data](#)
 - abstract data types that can be inherited, generalized and specialized
 - ASN.1 data types according to Z.105
- [Type Concept](#)
 - describing type hierarchies with inheritance, generalization and specialization

Structure

[Figure 1](#) shows the four main hierarchical levels in SDL: system, block, process, and procedure.

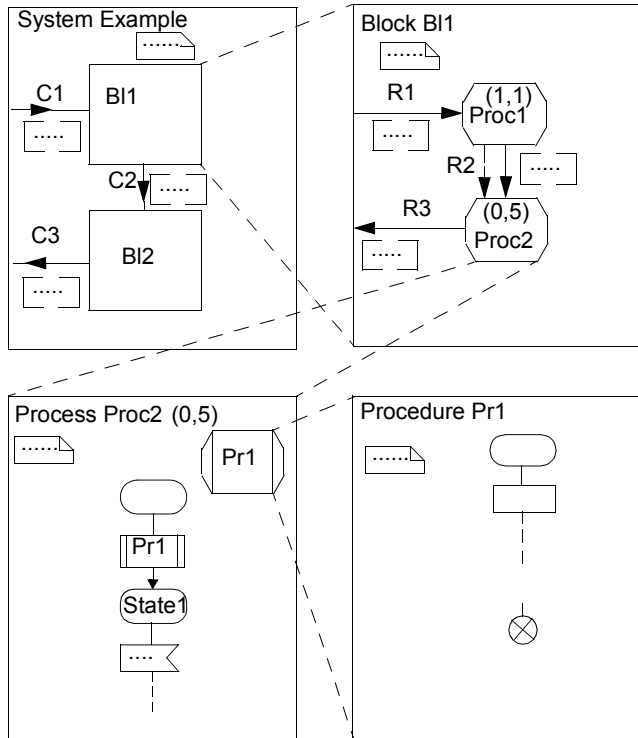


Figure 1: The architectural view of an SDL system

In addition, there is a service concept that can be used within processes. Procedures can be used in both processes and services.

Communication

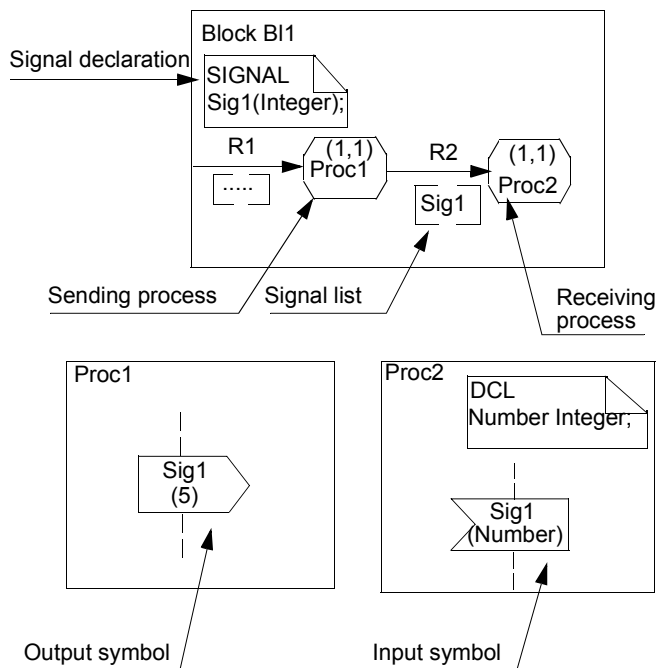


Figure 2: Sending signals between two processes

In SDL, there is no global data. This approach requires that information between processes, or between processes and the environment, must be sent with signals and optional signal parameters. Signals are sent asynchronously, that is, the sending process continues executing without waiting for an acknowledgment from the receiving process.

Synchronous communication is possible via a shorthand, remote procedure call. This shorthand is transformed to signal sending with an extra signal for the acknowledgment.

Behavior

The dynamic behavior in an SDL system is described in the processes. The system/block hierarchy is only a static description of the system structure. Processes in SDL can be created at system start, or created and terminated dynamically at runtime. More than one instance of a process can exist. Each instance has a unique process identifier (PID). This makes it possible to send signals to individual instances of a process. The concept of processes and process instances that work autonomously and concurrently makes SDL a true real-time language.

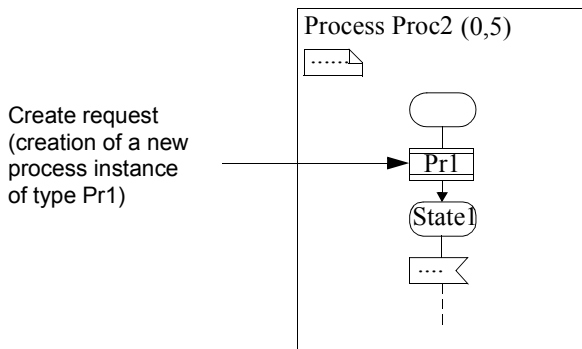


Figure 3: Creation of a new process instance at runtime

Data

The abstract data types concept used within SDL is very well suited to a specification language. An abstract data type is a data type with no specified data structure. Instead, it specifies a set of values, a set of operations allowed on the data type and a set of equations that the operations must fulfil. This approach makes it very simple to map an SDL data type to data types used in other high-level languages.

Alternatively, ASN.1 types can be used in SDL. This is useful when specifying or implementing telecommunication applications that make use of ASN.1. ITU-T Recommendation Z.105 defines how ASN.1 is used in combination with SDL. For more information on ASN.1, see [“ASN.1 – Abstract Syntax Notation One” on page 17](#).

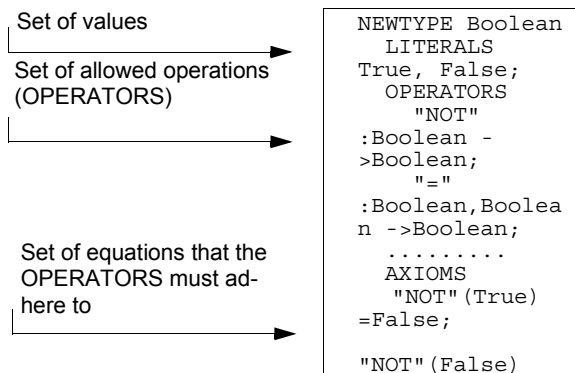


Figure 4: Abstract Data Type example

Type Concept

The object-oriented concepts of SDL give you powerful tools for structuring and reuse. The concept is based on type definitions. All structural building blocks can be typed. Type definitions can be placed anywhere in the system, and also in packages outside the system.

One of the major benefits of using an object oriented language is the possibility to create new objects by adding new properties to existing objects, or to redefine properties of existing objects. This is what is commonly referred to as specialization.

In SDL, specialization of types can be accomplished in two ways:

- A subtype may add properties not defined in the supertype. One may, for example, add new transitions to a process type, add new processes to a block type, etc.
- A subtype may redefine virtual types and virtual transitions defined in the supertype. It is possible to redefine the contents of a transition in a process type, to redefine the contents/structure of a block type, etc.

The Message Sequence Chart Language

History

During the last years, ITU has made a considerable effort in standardizing a formal language which defines Message Sequence Charts (MSC). In the summer of 1992, a first version of the MSC recommendation Z.120 was published.

As defined in the recommendation Z.120, the MSC language offers a powerful complement to SDL in describing the dynamic behavior of an SDL system. Its graphical representation is well suited for presenting a complex dynamic behavior in a clear and unambiguous way which is easy to understand.

There is an extended version of the MSC standard, called MSC'96, as defined in the current Z.120. In the SDL Suite, there is support for the most important MSC'96 extensions. See [“Compatibility with ITU MSC” on page 5 in chapter 1, Compatibility Notes, in the Release Guide](#) for more information.

Plain MSC

An MSC describes one or more traces from one node to another node of an abstract communication tree generated from an SDL specification.

Basically, the information interchange is carried out by sending *messages* from one *instance* to another (see [Figure 5](#)). In an SDL specification, those messages would coincide with the signals which are sent from one process and consumed in another process. The instances would correspond to any part of the specification (an SDL system, a block or a process).

An MSC can reference another MSC using an *MSC reference* symbol. MSC references can for example be used to have one MSC describing an initialization sequence and then reference this MSC from a number of other MSCs.

The reference symbol may not only refer to an MSC but can also contain MSC reference expressions that reference more than one MSC. This construct gives a very compact MSC representation and it also provides an excellent means for reusability of certain MSCs.

By using *inline operator expressions*, several MSC scenarios can be composed in a single diagram. The same structures of events can be expressed as with MSC reference symbols.

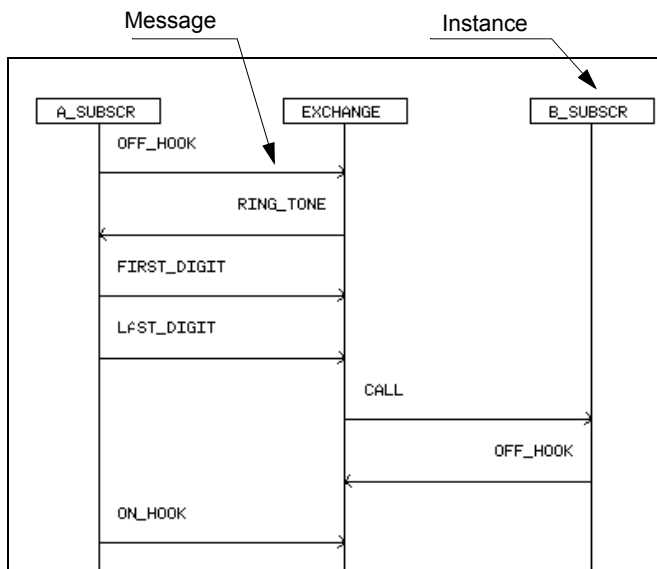


Figure 5: An example of a simple Message Sequence Chart

High-Level MSC

A high-level MSC (HMSC) provides a means to graphically define how a set of MSCs can be combined. Contrary to plain MSCs, instances and messages are not shown within an HMSC, but it focus completely on the composition aspects. HMSCs can be hierarchically structured, i.e. it is possible to refine HMSCs by other HMSCs. The power of the MSC language is considerably improved with the new concepts introduced with HMSCs. It is e.g. much easier to specify a main scenario together with all accompanying exceptions.

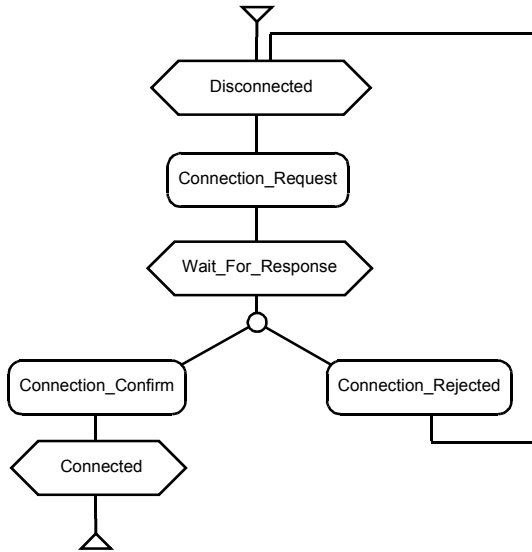


Figure 6: Example of an HMSC

Graphical and Textual Notations

The MSC language supports two notations which are equivalent. Besides the graphical notation (MSC-GR), a textual notation (MSC-PR) is standardized since the autumn of 1994.

Application Areas

Among the various application areas, we have selected the following:

- Producing documents with the purpose of defining the requirements of a system.
- Facilitating the design phase, by identifying and documenting a multitude of dynamic cases before starting designing with SDL.
- Presenting the execution of a simulation as a graphical output which is easy to understand and which can later on be verified against a reference. Message Sequence Charts can be verified against an SDL system using the SDL Suite.
- Presenting the execution trace of an SDL system during an interactive simulation and generation of reports.

Object Model Notation

The object model notation used in the SDL Suite is an adaptation of the notations used in OMT (Object Modeling Technique) and UML (Unified Modeling Language). The OMT/UML notation is a commonly accepted graphical notation that is used for drawing diagrams that describe objects and the relations between them.

Class

The most important concept in an object model is the class definition. A class is a description of a group of similar objects that share the properties defined by the class. The properties of a class are described with attributes and operations. The object model notation for a class is exemplified in [Figure 7](#), where the second class definition also shows how to define attributes and operations.

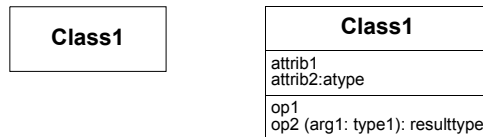


Figure 7: A collapsed class symbol and a class symbol with attributes and operations

Classes may inherit attributes and operations from other classes, known as specialization and generalization. The object model notation for this is shown in [Figure 8](#).

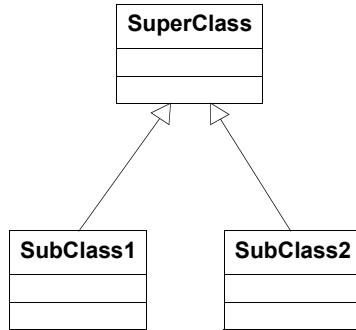


Figure 8: inheritance between classes

Relations and Multiplicity

Classes may be physically or logically related to each other. This is shown in the object model by means of associations as shown in [Figure 9](#). An association may have a name and/or the endpoints of the association may be labeled by the role of this endpoint.

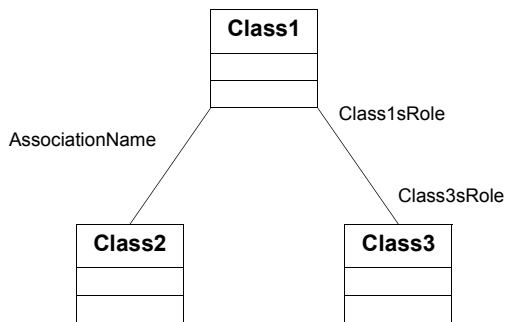


Figure 9: Associations between classes

Aggregation is special kind of association, indicating a “consists of” relation. It has its own notation as shown in [Figure 10](#).

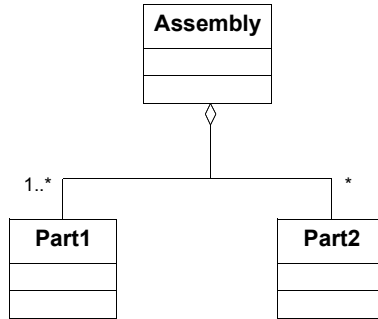


Figure 10: Aggregation

The endpoints of associations and aggregations may have a multiplicity according to the following:

- No multiplicity (exactly one)
- ***** (zero or more)
- **0, 1** (zero or one)
- **1..*** (one or more)
- **1..3, 6, 10..*** (several intervals: 1, 2, 3, 6, 10 or more)

Objects

Besides class definitions, object models may also contain objects (instances) and their relations. The relation that exists between objects are links, which corresponds to associations for classes. The object symbol has one field containing the object name and a reference to the class (“name:class”), and an attribute field where constant or default values can be assigned to the object attributes. See [Figure 11](#).

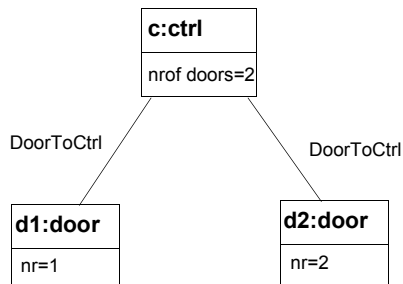


Figure 11: Objects related by links

State Chart Notation

The state chart notation used in the SDL Suite is a subset of the notations used in OMT (Object Modeling Technique) and UML (Unified Modeling Language).

A state chart model is suitable to use together with class and object models. The descriptions of the behavior of a class in a class diagram is collected into a state chart which describes the dynamic view of the model by means of *states* and *transitions* between states.

State

A state symbol describes the name of the class, state variables, and internal activities. Internal activities are taking place upon entering the state, while in the state and when exiting the state. Activities are described by specifying events and associated actions. [Figure 12](#) shows a collapsed state and a state with events.

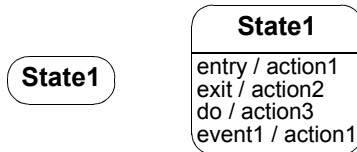


Figure 12: A collapsed state symbol and a state symbol with events

Transition

A transition symbol is an arrow which typically connects two state symbols. A transition is triggered by an event together with a condition, and a transition then executes an action; see [Figure 13](#).

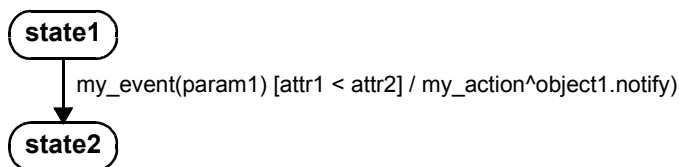


Figure 13: The transition from state1 to state2 is triggered by the event *my_event* and the condition that *attr1* is less than *attr2*

Start and Termination Symbol

The start symbol denotes the starting point of a state machine described by a state chart and the termination symbol denotes the point of termination of a state machine. Figure 14 shows a simple state machine, describing the behavior of a door, including start and termination symbols.

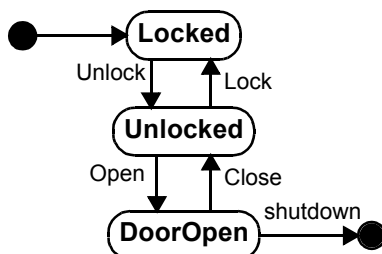


Figure 14: A simple state chart with a start symbol and a termination symbol

Substates

States may be refined into nested diagrams of sub-states, or hierarchical states. The state represents a simplification of more complex behavior expressed in the nested diagram.

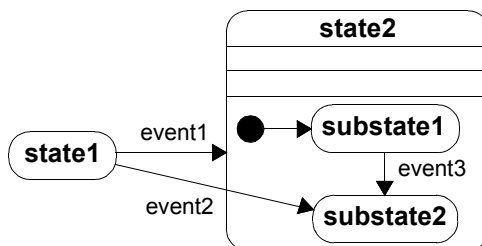


Figure 15: A state with substates

ASN.1 – Abstract Syntax Notation One

ASN.1 (ITU-T Recommendation X.680-683) is a generic notation standardized by ISO and ITU for the specification of data types and values. The general idea behind ASN.1 is to describe data type information independent of the transfer format.

The original use of ASN.1 has been the information description in high-level protocols like FTAM, CMIP, MHS, DS, VT, etc. Today it is also frequently used in numerous other telecommunication protocols and applications.

ASN.1 data types and values can be defined in modules that can be used in TTCN and in SDL. This makes it possible to use the data types of an application both in the SDL specification and in the TTCN test suite, which assures consistency between the information transferred in the system specification and the test specification.

The TTCN Notation

As the use of standards within the world of Information Technology and Telecommunications has increased tremendously during the last decade, so has the need for methods and tools that support the verification and validation of both the standards and their actual implementations.

This need has been addressed by ISO and ITU in the “Framework and Methodology for Conformance Testing of Implementations of OSI and ITU Protocols”. The framework has for some time had the status of an international standard as ISO/IEC 9646 (or X.290).

The standard introduces the concept of abstract test suites (consisting of abstract test cases). This is a description of a set of tests that should be executed for a system. The tests should be described using a black-box model, i.e. only control and observation using the available interfaces.

The abstract tests are to be described using a formal language rather than using informal natural language. As part of the standard the language TTCN is defined in order to describe the abstract tests.

TTCN – Tree and Tabular Combined Notation

With TTCN a test suite is specified. This is a collection of various test cases together with all the declarations and components it needs.

Each test case is described as an event tree. In this tree behaviors like “First we send A, then either B or C will be received, if it was B we will send D...” are described. The new version of TTCN allows several event trees to be running concurrently.

TTCN is abstract in the sense of being independent of the actual test systems. This means that a test suite in TTCN for one application (protocol, system...) can be used in any test environment for that application.

The use of TTCN has increased tremendously during the last years. This has been augmented by the significant amount of TTCN test suites released by various standardization bodies. TTCN is however not only used in standardization work. The language is very suitable for all kinds of functional testing for communicating systems. This has led to a wide usage also within the industry.

The specifications of the messages being sent and received can be defined using either the built-in form of TTCN or by using ASN.1.

Tool Support

IBM Rational has been a firm supporter of SDL for a long time. We cooperate with ITU in the on-going work of improving the language and with ETSI in using SDL for defining protocol standards. We initiate and participate in international research programs on how to use the language in different application areas (such as the European Community programs RACE, ESPRIT and EUREKA). Our experience and know-how in these areas is put to practice when we develop software engineering tools that support the languages.

A tool for a specification language must be able to create, maintain, and analyze a specification. It is also fundamental that the tool can simulate, validate and generate application code to other high level languages.

The SDL Suite can do all of this.

References

- [1] ITU Recommendation Z.100:
Specification and Description Language (SDL)
1994, ITU, General Secretariat - Sales Section,
Places des Nations, CH-1211 Geneva 20
- [2] Annex A, B, C1, C2 D, E, F1, F2 and F3 to Z.100, as above
- [3] ITU Recommendation Z.120:
Message Sequence Charts (MSC)
1992, ITU General Secretariat - Sales Section
Place des Nations, CH-1211 Geneva 20
- [4] Jan Ellsberger, Dieter Hogrefe, Amardeo Sarma:
SDL – Formal Object-oriented Language for Communicating Systems.
Prentice Hall Europe (1997)
ISBN 0-13-632886-5
- [5] A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, J.R.W. Smith:
Systems Engineering Using SDL-92.
Elsevier (1994)
ISBN 0-444-89872-7
- [6] Ferenc Belina, Dieter Hogrefe, Amardeo Sarma:
SDL with Applications from Protocol Specification.
Prentice Hall International (UK) Ltd. (1991)
ISBN 0-13-785890-6
- [7] Belina, Hogrefe:
The CCITT Specification and Description Language SDL
Computer Networks and ISDN System.
North-Holland, Amsterdam (1988/1989)
- [8] Bræk, Gorman, Haugen, Melby, Møller-Pedersen, Sanders:
TIme - The Integrated Method
SINTEF 1998
<http://www.sintef.no/time>
- [9] Færgemand, Marques (editors):
SDL 89: The language at work.
Proceedings of the Fourth SDL Forum,
North Holland, Amsterdam (1989)

[10] Færgemand, Reed (editors):
SDL 91: Evolving Methods.
Proceedings of the Fifth SDL Forum,
North-Holland, Amsterdam (1991)

[11] Færgemand, Sarma (editors):
SDL 93: Using Objects.
Proceedings of the Sixth SDL Forum,
North-Holland, Amsterdam (1993)

[12] Haugen, Møller-Pedersen:
Tutorial on object-oriented SDL.
SISU Project Report 91002
Norwegian Computer Center
PO Box 114, N-0314 Oslo 3, Norway

[13] Behcet Sarikaya:
Principles of Protocol Engineering and Conformance Testing.
Simon & Schuster International (1992)

[14] Sarraco, Smith, Reed:
Telecommunications system engineering using SDL.
North-Holland, Amsterdam (1989)

[15] K.J. Turner (editor):
Using Formal Description Techniques -
An Introduction to Estelle, LOTOS and SDL.
John Wiley & Sons (1992)

[16] ITU Recommendation X.680-683
Abstract Syntax Notation One (ASN.1)
1994, ITU, General Secretariat- Sales Section,
Places des Nations, CH-1211 Geneva 20

[17] ITU Recommendation Z.105
SDL Combined with ASN.1 (SDL/ASN.1)
1995, ITU, General Secretariat- Sales Section,
Places des Nations, CH-1211 Geneva 20

Introduction to the SDL Suite

This chapter contains a brief introduction to the SDL Suite and to the functionality of its components.

After reading this chapter, you may want to familiarize yourself with the SDL Suite: starting the tools, proceeding with a small example, or reading about what is new in this release in comparison to earlier versions. You are then recommended to study the following chapters:

- For a beginner's tutorial on the SDL Suite: [chapter 3, *Tutorial: The Editors and the Analyzer*](#), in this volume.
- For news compared to previous versions: [chapter 2, *Release Notes, in the Release Guide*](#).

About the SDL Suite

IBM Rational

The SDL Suite is developed and marketed by IBM Rational. Our company has been a firm supporter of the SDL, MSC and UML languages for a long time. We cooperate with ITU and OMG in the ongoing work of improving the languages and with ETSI in defining international standards in the field of communication protocols.

We initiate and participate in international research programs on how to use the languages in different application areas (such as the European Community programs RACE, ESPRIT and EUREKA, as well as the Swedish national IT program).

Our experience and know-how in these areas is put to practice when we develop software engineering tools that support the languages.

The SDL Suite

Tools for design and specification languages must be able to create, maintain, and verify a specification with respect to the language syntax and semantics. It is also fundamental that the tools can simulate, validate and generate application code to other high level languages.

To be able to perform a complete development cycle, the tool should support early analysis phases, and the move from object-oriented analysis to SDL design.

The tool should be able to export and import information from other SDL tools. Major documentation standards or de-facto standards should be supported.

The tool should provide an intuitive and consistent graphical user interface which reduces learning time and makes it easy to work with the tool. Besides the graphical user interface, a batch facility should allow to process a large amount of information without user interaction.

A powerful, context-sensitive Help facility should be provided, freeing you from time-consuming browsing through user documentation in search for the topic of interest.

The SDL Suite can do all of this, and much more.

Overview of the SDL Suite

Architecture

The SDL Suite is a member of a wider tool family, which also includes the TTCN Suite and Logiscope.

The SDL Suite consist of a number of separate tools that process information. The tools are integrated using a *selective broadcast integration mechanism*, making it possible to design a highly integrated system from separate tools. This approach also makes it possible to add new tools without creating any conflicts with the existing tools. In addition, the integration between two separate tools can be easily enhanced, and tools can communicate with each other over a network.

The interface that ties the tools and editors is called [*The PostMaster*](#). The parts of the Postmaster interface that are of interest for the users, are documented so that you can read about how to integrate your own tools with SDL Suite.

Starting the SDL Suite tools

The SDL Suite components are normally started by using the *sdt* start script in the bin directory of the installation. This script can take a number of options:

- `sdt -reuse`
If there is a running SDL Suite session, then the Organizer from that session is displayed. If there is no running SDL Suite session, then a new session is started.
- `sdt <system file>`
Starts the SDL Suite components and loads the specified system file.
- `sdt <diagram file>`
Starts the SDL Suite components and loads the specified diagram file in an editor.
- `sdt <archive file> [<unpack directory>]`
Starts the SDL Suite components, unpacks the specified archive file, and loads any system file found in the archive file. The archive file is unpacked in the specified unpack directory. If no directory is

specified, then the Organizer will display a dialog, asking for a directory to unpack in.

- **On UNIX**, `sdt -fg`

Normally, the `sdt` start script immediately gives the user a new command line prompt for other commands. When `sdt -fg` or is used, the start script will wait until the SDL Suite is closed down before giving the user a new command line prompt.

- `sdt -noclients`

This variant starts the Postmaster without starting any clients/applications such as the Organizer. Later, Postmaster clients can be started and attached to the postmaster. To attach to an existing postmaster, start a client with `-post`. Read more about this in [“Run-Time Considerations” on page 515 in chapter 10, *The PostMaster*](#).

- `sdt -grdiff [-notmoved] v1.ssy v2.ssy [[-original o.ssy] -mergeto r.ssy]`

Starts the SDL Suite components and immediately invokes the *Compare Diagrams* operation (see [“Compare Diagrams” on page 2029 in chapter 43, *Using the SDL Editor*](#) and [“Compare Diagrams” on page 1692 in chapter 39, *Using Diagram Editors*](#)), comparing the SDL diagram files `v1.ssy` and `v2.ssy`. The command can also be used to compare HMSC or MSC diagrams. The use of the `-notmoved` option corresponds to setting the option [Ignore moved or resized objects](#) to off. (Note that if you use the `-n` option on UNIX it will be processed by the X window system) Normally this means that moved and resized symbols will not be detected as being different. When the operation is finished, the SDL Suite is closed down. If the `-mergeto` option is used, the Merge Diagrams operation is used instead. (See [“Merge Diagrams” on page 2030 in chapter 43, *Using the SDL Editor*](#) and [“Merge Diagrams” on page 1692 in chapter 39, *Using Diagram Editors*](#)) and the merge result diagram is saved in `r.ssy`. If the `-original` option is used as well, then the merge operation tries to auto-merge differences using the `o.ssy` file as a guide for deciding how to merge `v1.ssy` and `v2.ssy`. `o.ssy` should point out an original diagram file version that both `v1.ssy` and `v2.ssy` are derived from. When using `-grdiff` to start a merge operation from another tool (such as a configuration management tool), combine `-grdiff` with `-fg` to have the other tool wait with processing the merge result file until after the merge operation is finished.

- `sdt -sdtldiff v1.sdt v2.sdt [-mergeto r.sdt]`
Starts the SDL Suite components and immediately invokes the *Compare System* operation (see [“Compare System” on page 74 in chapter 2, The Organizer](#), comparing the system files v1.sdt and v2.sdt. When the operation is finished, the SDL Suite is closed down. If the *-mergeto* option is used, the user will be prompted to save the merge result when the operation is finished, and r.sdt is used as a proposed file name.

Batch Facilities

The [Batch Facilities](#) are commands that you type from the OS prompt. These facilities take advantage of the Postmaster and pass messages to the tools, ordering the individual tools to process information as required. The batch facilities support the following operations:

- Printing (to file or to printer)
- Analyzing (typically syntactic and semantic check of an SDL system)
- Making (for instance building an application for target environment)
- Comparing SDL, MSC or HMSC diagrams (with a textual report)

Licensing Mechanism

The software license server controls the licensing of the tools included in SDL Suite. This is performed through a floating license mechanism based on a third party software, FLEXnet Publisher™. The current license numbers along with a key are stored on a text file, which is distributed at installation of the software. This provides a flexible way of upgrading licences and adding new license agreements, as well as allowing you to keep track of the actual usage of the tools you have purchased.

FLEXnet supports multiple tools (even from different tool manufacturers) sharing the same license server, so IBM Rational should not cause any problems when installing it into your computer environment.

For increased flexibility in the use of licenses and to prevent started but unused tools from holding licenses, an optional timeout feature can be

enabled. This will automatically release licenses when a user has been idle during a selectable interval.

For more information on license mechanisms, see [chapter 6, *Understanding IBM Rational Licenses, in the Installation Guide*](#).

Common Tools

The SDL Suite shares a set of tools common to SDL Suite and TTCN Suite:

- [The Organizer](#) features a graphical view of all diagrams and documents making up a system. This may include SDL hierarchies, Message Sequence Charts, Object Model diagrams, State Charts, High-level MSCs, TTCN documents and text documents. The view may be freely organized into chapters and modules according to your preference.

Furthermore, the Organizer manages the other tools, taking advantage of their respective functionality when needed and thus providing the feeling of a truly integrated tool set.

- [The Link Manager](#) and [The Entity Dictionary](#) maintains and visualizes [Implinks and Endpoints](#). Implinks (short for implementation links) are used to trace the implementation and design decisions for concepts and objects between different phases in the development process. The link endpoints are texts and objects created in the editors described below. The Link Manager and the Entity Dictionary can be started from all of the editors.
- [The Preference Manager](#) allows you to set up or change the behavior of the tools by customizing the values of preference parameters. It is possible to specify whether a customized behavior should be project-wide or even company-wide, or if an individual user should be allowed to customize some behavior.
- When you are [Printing Documents and Diagrams](#), there are various options that allow you to customize the printouts so that they fit in your documentation environment. Except for when you print TTCN documents, it is possible to generate, PostScript, encapsulated PostScript, FrameMaker™, Interleaf™ and web files. **In Windows**, you can also print to any printer you have set up in Microsoft Windows.

- *The on-line help* provides access to help on tools, windows, dialogs and commands. The on-line help is in HTML-format, featuring hypertext links and navigation support. The PostScript files that were used when printing this manual are also enclosed in the distribution. You are free to produce additional hard-printed copies of the manual pages that are of interest.
- [*The PostMaster*](#) implements the integration mechanism that ties the tools together. The public parts of the PostMaster interface are documented, allowing you to integrate your own tools with SDL Suite and TTCN Suite tools and the information they manage.

The SDL Suite Graphical Tools

The SDL Suite comprises the following graphical tools:

- *The diagram editors* are used for creating, editing and printing Object Model, State Charts, Message Sequence Charts and High-level Message Sequence Charts diagrams.

The OM Editor uses the full graphical notations of OMT/UML. It keeps track of all class and object definitions with the same name in a scope of OM diagrams, and supports the merging of these definitions for maintaining a combined view of a class. The SC and HMSC editors work in a similar way.

The MSC Editor uses the graphical notation defined in the standard Z.120. Also, it can serve as a powerful graphical trace tool when you simulate and validate a system specified in SDL. MSCs can also be verified for consistency with an SDL system when you use the SDL Explorer.

- *The SDL Editor* is used for creating, editing and printing specifications and descriptions using the graphical SDL notation defined in the standard Z.100. The SDL Editor also performs various syntax checks at editing time.
 - Advanced functions include a context-sensitive grammar help and signal dictionary. When you edit an SDL diagram, the signal dictionary is automatically updated to contain all SDL sig-

nals that you add to a system and provides immediate access to them.

- The SDL Editor can also display [Overview Diagrams](#) of the SDL system, where the diagrams are displayed in a nested fashion.
- [The SDL Type Viewer](#) visualizes the impact of the inheritance and specialization mechanisms in your SDL-92 system. The Type Viewer produces a graphical tree that is of great assistance to understand and take full advantage of the SDL types¹ that you have defined in an SDL system.
- [The SDL Index Viewer](#) presents listings of definitions and cross-references in a clear and easy-to-understand graphical notation. The Index Viewer is provided with filtering and navigation functions, with a trace-back to the source SDL or MSC diagrams.
- [The SDL Coverage Viewer](#) is a test coverage and profiling tool that displays the results of a simulation or validation as a graphical transition or symbol tree. The tool can present an overview of the system, coverage or a detailed view on a part of the system.

Other SDL Suite Tools and Back-End Facilities

The following additional tools and facilities are available:

- [The Text Editor](#) is used for creating, editing and printing ASCII text documents. The text documents can be textual requirements, use cases and other textual documentation used in the development process. The Text Editor can also be used for writing ASN.1 or C code to be linked to the SDL system.
- [The ADT Library](#) (library of Abstract Data Types) features a number of general ADTs that provide the basic services that are often needed when you design an SDL system. The ADT library is distributed in source code so you can tailor the ADTs to fit your specific requirements, if needed.
- [The SDL Analyzer](#) performs several functions. It performs syntactic and semantic analysis of your SDL descriptions, generates error reports and warnings in appropriate cases, and has the ability to pro-

1. The SDL term *type* corresponds to the term *class*, used in many OO notations

Overview of the SDL Suite

duce information about definitions and cross references in an SDL system. The Analyzer also converts SDL information from the Graphical Representation (SDL/GR) to the textual Phrase Representation (SDL/PR). The reverse conversion is also possible, allowing you for instance to import PR files from other tools supporting SDL.

- [*The Cadvanced/Cbasic SDL to C Compiler*](#) transforms your SDL system into a number of C source files that are compiled and linked with an SDL Suite run-time library. The C code can be used for a number of purposes, depending on what libraries are available in your configuration (see below). The SDL to C Compiler is available in a *Cbasic* (for simulation and validation purposes), and a *Cadvanced* (for building any kind of application) version.
- [*The SDL Simulator*](#) library allows you to make an executable program, a *simulator*, which helps you to understand and debug the behavior of a system specification. The simulator can be controlled from a graphical user interface (SimUI).

You can choose to focus on the external view of a system specification, where you are interested in the signal interface, or on the internal behavior of a system specification. The execution of a simulator can be traced in a graphical mode in the source SDL diagrams and can be logged graphically in terms of Message Sequence Charts. Target simulation is also supported.

- [*The SDL Explorer*](#) library allows you to make an *SDL Explorer*, an advanced “self-exploring” simulator that may be used for finding errors and inconsistencies in an SDL system and for verifying that a system is consistent with a Message Sequence Chart. The Explorer can be controlled from a graphical user interface (ExpUI).
- [*The Performance Library*](#) allows you to create a performance model of your SDL system that you run on your host computer. The library is optimized with respect to performance, so that a large amount of statistical data can be produced during a reasonable execution time.
- The Cadvanced SDL to C Compiler can be used for [*Building an Application*](#) for both host and target environments. Predefined *Application libraries* are available for specific host environments. [*The Master Library*](#) is the SDL Suite run-time library in source code format, which can be customized to fit different needs and operating systems.

[*Integration with Operating Systems*](#) supports most of the commercially available real-time operating systems. You can also build applications where the runtime library schedules the system and sets the real-time pace.

Overview of the SDL Suite

- [*The Cmicro SDL to C Compiler*](#) is designed to meet the needs of small to mid-range microcomputer controlled applications. It translates an SDL system into optimized and compact C code with highly reduced memory requirements. The Cmicro SDL to C Compiler is part of the *Cmicro Package* which in addition consists of the Cmicro Library and the SDL Target Tester. The Cmicro Package can be ordered separately.
 - [*The Cmicro Library*](#) is the “virtual SDL machine” needed to build an executable from the generated Cmicro Code.
 - [*The SDL Target Tester*](#) allows testing and debugging of the generated SDL system while it is running on a target. A prerequisite is a communications link to a host system. The SDL Target Tester is an optional part of the Cmicro Package.
- [*TTCN Test Suite Generation*](#) is provided by two features: TTCN Link and Autolink. They provide a means to check the consistency between an SDL system managed by the SDL Suite and a test specification, expressed in TTCN¹ managed by the TTCN Suite. TTCN Link generates the declarations of the test specification automatically. In the TTCN Suite, there is direct access to the SDL system specification and you can interactively build test cases. Autolink is a feature of the SDL Explorer, and can generate entire test suites from an SDL specification.

1. TTCN stands for Tree and Tabular Combined Notation. It is an ISO standard that is used to describe a test specification.

Information Management

In order to properly use SDL Suite, you need to understand the basics for how the information is organized.

SDL Diagrams

The SDL Suite primarily handles SDL information in the graphical representation, SDL/GR. The major advantage that follows this approach is that you are free to apply any graphical style guide to your diagrams since the SDL Suite lets you position symbols and shape lines the way you like.

Each SDL diagram consists of a number of diagram pages. An SDL diagram page may contain references to other SDL diagrams. This allows you to build a hierarchical structure which adheres to the SDL syntax rules. See [Figure 16](#).

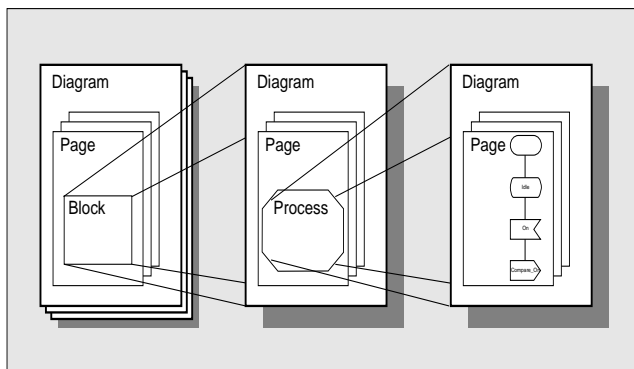


Figure 16: Organization of SDL information

Each diagram stored on its own individual file. An SDL structure is built up from a number of these SDL files. These files are logically tied together by the SDL Suite components, in order to constitute a coherent SDL structure. This process is managed by the Organizer. The SDL Suite can manage several separate SDL structures at the same time.

You may also include SDL/PR files into an SDL/GR structure. Transformation of SDL/GR to SDL/PR and vice versa is supported.

MSC Diagrams

Message Sequence Charts are mainly handled in the graphical representation, MSC/GR.

In contrast to SDL diagrams, MSCs are not paginated and cannot build a hierarchical structure. However, an MSC diagram can reference other MSCs (or HMSCs, see below), but without implying any structure between them.

MSC diagrams may be managed as entities of their own. The Organizer also supports including MSCs in an SDL structure using the concept of *associated documents*.

SDL Suite allows reading and writing of MSCs expressed in the textual form, MSC/PR. Both the *instance-oriented* and *event-oriented* forms are supported, according to the recommendation.

High-Level MSC Diagrams

In contrast to “plain” MSCs, High-level MSCs (HMSCs) are paginated, but they cannot build a hierarchical structure visible in the Organizer. However, an HMSC diagram can reference other HMSCs or MSCs, but without implying any structure between them.

Textual SDL and MSC Formats

SDL Suite has the ability to read and write SDL and MSC textual files, SDL/PR and MSC/PR. The primary purpose is to enable importing and exporting of SDL and MSC information, rather than to provide an alternative storage format, since the layout and exact appearance of a diagram is lost when stored in PR format and read back again.

SDL Suite also use the PR formats as temporary storage formats when processing information.

SDL/GR diagrams can be converted to and from CIF (Common Interchange Format) files, by using CIF converters supplied with the SDL Suite. CIF is an extension to SDL/PR that also stores the graphical layout information. However, CIF files cannot be managed directly by the SDL Suite.

Object Model Diagrams

Object Model (OM) diagrams may be managed as entities of their own, or grouped together using the concept of *modules* in the Organizer.

OM diagrams are paginated, but does not contain any structure information. However, a *scope* concept is used to allow the same object class to be defined in more than one OM page or diagram. All diagrams and pages within the scope are considered when the complete definition of a class is needed. OM diagram belong to the same scope if they are managed in the same *module* in the Organizer.

State Chart Diagrams

State Chart (SC) diagrams are paginated, but does not contain any structure information, nor any references to other diagrams. They are always managed as entities of their own.

Text Documents

SDL Suite and TTCN Suite handles text documents in the form of plain ASCII files. SDL Suite and TTCN Suite uses the file extension of text files to determine the type of text file. SDL Suite and TTCN Suite recognizes text files as C header, ASN.1 or as plain text files.

Text documents are managed as entities of their own, but C header and ASN.1 specifications can be linked to the rest of the system by using the concept of *dependency links*. In this way, these text documents can be analyzed and translated to SDL/PR format.

A text document can also be a *build script*, containing commands to control the analyze and code generation process in detail.

The System File

Once SDL Suite is up and running, you may work on individual documents, regarding them as individual objects of their own. However, this requires that you keep track of each individual file.

When the amount of documents increases, this process tends to become rather complicated, in particular when introducing inheritance and specialization between SDL diagrams, and dependency links between different document types.

To cope with this problem and as a means to ensure the consistency of a document structure, the *system file* is introduced. The system file is managed by the Organizer.

Document Structure

The system file holds the information about the SDL structure and all other documents included in a system. It also keeps track of the file bindings, i.e. what file a particular document is stored on, and the dependency links between the documents. When working on your documents, the Organizer keeps track of the changes you apply and updates the system file accordingly.

A graphical approach is used, in order to display the contents of the system file in the Organizer. Documents may be freely organized into *chapters* and *modules* (within chapters) to keep related documents together.

Connections between documents in different chapters, modules and SDL structures can be made in the form of *associations* and *dependency links*.

Options

In addition to the properties mentioned above, the system file may store information about what *options* you have set up for the document structure that is managed by the system file. Typically, *analysis* and *code generation* options are stored in the system file.

The Link File

The Link Manager keeps track of all link endpoints and implementation links in the system. This link database is stored on a separate link file, which is referenced from the system file.

Control Unit Files

Control unit files facilitate multiuser support when you work with an SDL system. They contain structure information for a subset of a document system and are suitable for configuration management (revision control). If control unit files exist, they are referenced from the system file.

Source Management

Since SDL Suite operates on files, you may use any revision handling system for checking out work copies of your SDL diagram files to your work directory (this task needs to be performed outside the SDL Suite).

The SDL Suite may also be configured to manage multiple versions of your source documents, by binding a document to any suitable file in your file system.

The SDL Suite provides mechanisms for an easy rebinding of documents. These file binding mechanisms allow you to keep track of multiple versions of your source documents with a minimum of effort.

Target Management

Virtually all of the output information that is produced with the SDL Suite consists of files, most of them use a text-based format (for instance SDL/PR files and C files).

You may specify default locations for files that are generated. Also, you may specify the level of granularity, allowing you to generate multiple files or one file only.

Furthermore, the SDL Suite features an *SDL-Make* mechanism that minimizes the turnaround time, by computing the passes the tool must run in response to a modification of a source diagram.

PCs and Workstations

User Interface

On UNIX workstations, the SDL Suite is implemented as X Window applications, using the Motif widget set. On PCs, the SDL Suite is designed as Microsoft Windows applications (Windows 2000 and XP). At present, some features are not available on the PC platform.

Since the SDL Suite is supported on different systems, there may be slight differences in the appearance of the tools between environments. However, the functionality is identical as long as the underlying system and the OS allow it.

All SDL Suite and TTCN Suite graphical applications follow the same style guide, described in [chapter 1, *User Interface and Basic Operations, in the User's Manual*](#).

Supported UNIX Systems

Full compatibility between the SDL Suite on PCs and the SDL Suite on UNIX workstations ensures that future upgrading of your computers towards workstations is possible, and allows heterogeneous network solutions with, for instance, PCs connected to a UNIX based file server.

On workstation environments, the following architectures and operating systems are supported:

- Sun SPARCstation (Solaris)

For more information about the supported platforms, see [chapter 1, *Platforms and Products, in the Installation Guide*](#).

Tutorial: The Editors and the Analyzer

The SDL Suite products are used for designing and specifying systems, in particular real-time systems. The SDL Suite supports the Specification and Description Language (SDL) as recommended by ITU (the Z.100 recommendation). The SDL Suite also supports the definition of Message Sequence Charts (MSCs), as well as parts of the Unified Modeling Language (UML) notation. For full support of the UML language you should use IBM Rational Tau/Developer.

This tutorial assumes that you are already familiar with SDL and have some brief notions about Message Sequence Charts.

We will demonstrate, by using a simple SDL system as example, the basic editing and analysis functionality that is available. You will practice various “hands-on” exercises that will get you more familiar with the SDL Editor and the MSC Editor, as well as the SDL Analyzer.

In order to learn how to use these tools, read through this entire chapter. As you read, you should perform the exercises on your computer system as they are described.

Purpose of This Tutorial

The purpose of this tutorial is to make you familiar with the user interface and the essential editing functionality in the SDL Suite. This tutorial is designed as a guided tour through the SDL Suite, where a number of hands-on exercises should be performed on your computer as you read this chapter.

We have on purpose selected a simple example that should be easy to understand. It is assumed that you have a basic knowledge about SDL — this chapter is **not** a tutorial on SDL.

This tutorial addresses primarily persons with no or little experience of the SDL Suite.

Once you have completed the exercises in this tutorial, you may want to continue with the tutorials that are presented in:

- [*chapter 4, Tutorial: The SDL Simulator*](#),
- [*chapter 5, Tutorial: The SDL Explorer*](#), and
- [*chapter 6, Tutorial: Applying SDL-92 to the DemonGame*](#).

Note: Platform differences

It is possible to run the tutorials on UNIX as well as on Windows platforms. Should there be any differences between the platforms, this is indicated in the text with the markers “on UNIX”, “Windows only”, etc. This is also indicated in the platform-specific screen shots.

When such platform indicators are found, please pay attention only to the instructions and screen shots that are valid for your platform.

The Demon Game

The example that has been chosen in this tutorial is a simplified version of the “Demon game”, which is a well known example in the SDL community, since it is, among other things, used as example in the SDL recommendation.

The SDL definition of the Demon game may be found in SDL/GR form later in this chapter (see [“Appendix A: The Definition of the SDL-88 DemonGame” on page 129](#)). The definition of the behavior of the Demon game is probably not the simplest way of describing the game, but it has been selected since it is good for demonstrating the facilities of simulation and validation.

Behavior of the Demon Game

Seen from the environment, the behavior of the system is as follows. The system accepts four different types of signals, Newgame, Endgame, Probe, and Result, where the first two signals are used to start and end a game. Only one game at a time can be played, that is, Newgame signals will be ignored when a game is in progress and Endgame will be ignored if there is no game in progress.

The game in itself is very simple. A “demon,” which in the system is represented by the process Demon, changes the status of the system every now and then between winning and losing. This is represented by the states Winning and Losing in the process Game. The user is to guess when the status is winning. If the user probes (outputs the signal Probe), when the status is winning, he wins one point. If the user probes when the status is losing he loses one point. The system responds to a Probe signal by either a Win or a Lose signal. To see the current score the user can issue a Result signal, which will be answered by a Score signal containing an integer parameter giving the current score.

Starting the SDL Suite

Some Preparatory Work

This tutorial assumes that the SDL Suite has been installed correctly, according to the instructions in the [Installation Guide](#).

Note: Installation directory

On UNIX, the installation directory is pointed out by the environment variable `$telelogic`. If this variable is not set in your UNIX environment, you should ask your system manager or the person responsible for the SDL Suite environment at your site for instructions on how to set this variable correctly.

In Windows, the installation directory is assumed to be `C:\IBM\Rational\SDL_TTCN_Suite6.3` throughout this tutorial. If you cannot find this directory on your PC, you should ask your system manager or the person responsible for the SDL Suite environment at your site for the correct path to the installation directory.

The directory `$telelogic/sdt/examples/demongame` (**on UNIX**), or `C:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\examples\demon-game` (**in Windows**), is created during installation and is the directory that contains the complete example, that you may look at whenever you feel insecure or want to “shortcut” an exercise.

In order **not** to modify these completed example files, you should create a dedicated directory for the purpose of this tutorial.

On UNIX, follow this instruction:

1. Create a new subdirectory in your home directory:

```
mkdir ~/demongame
```

In the remainder of this tutorial, we will assume this name for your personal tutorial directory.

In Windows, follow these instructions:

1. Create a local directory

```
C:\IBM\Rational\SDL_TTCN_Suite6.3\work\demongame on
```

your PC. In the remainder of this tutorial, we will assume this name for your personal tutorial directory.

Note: Do not use space characters in Windows

In **Windows**, SDL Suite does not support file or directory names that contain space characters. Make sure you do not use such names.

Starting the SDL Suite

On **UNIX**, to start the SDL Suite environment:

1. Change directory to your `demongame` directory:

```
cd ~/demongame
```

2. Now, type:

```
sdt
```

Note:

If the command `sdt` is not found, you first have to set up your `$path` variable correctly. Consult your system manager or the person that is responsible for the SDL Suite environment at your site.

In **Windows**, to start the SDL Suite in a manner suitable for this tutorial you should create and use a shortcut icon:

1. Locate the executable SDL Suite file `sdt.exe` (or just `sdt`) in `C:\IBM\Rational\SDL_TTCN_Suite6.3\bin\wini386`. (See the note [“Installation directory” on page 42](#) if you cannot find this directory.)
2. Create a shortcut icon to this file on the Windows desktop.
3. From the new icon’s popup menu, select *Properties*. In the dialog, select the *Shortcut* tab at the top.
4. In the *Start in* field, enter the path to your new directory, i.e. `C:\IBM\Rational\SDL_TTCN_Suite6.3\work\demongame`
5. Click *OK* to close the dialog.
6. Double-click the shortcut icon.

The Organizer Window

When you have started the SDL Suite, the *Organizer window* is displayed (see [Figure 17](#) and [Figure 18](#)). The Organizer is the main tool from which you have access to the tools in SDL Suite.

The Organizer also displays the *Welcome* window, where you may read the licensing agreement for SDL Suite and TTCN Suite. The window is always placed on top of the Organizer window and disappears as soon as you perform any action in the Organizer (you may also click the *Continue* button).

You are now ready to start working.

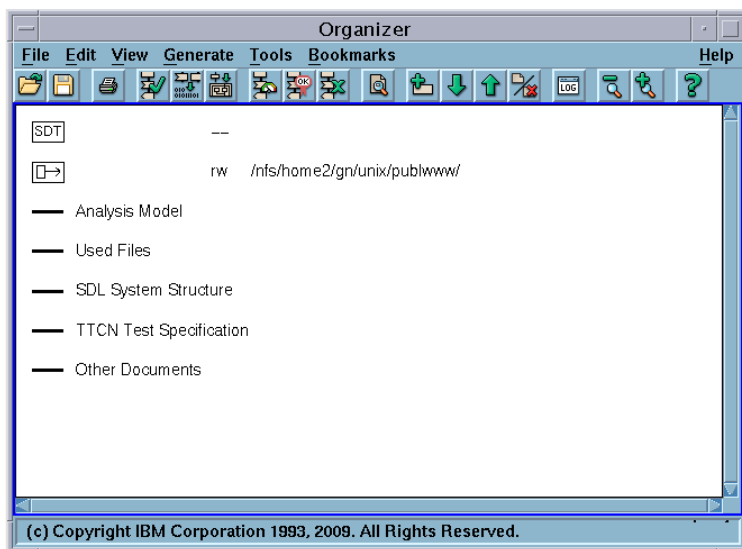


Figure 17: The Organizer window (on UNIX)

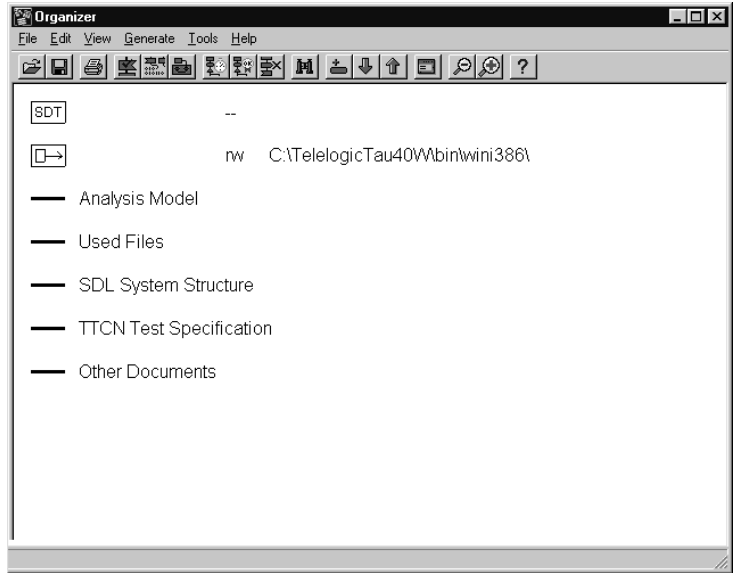


Figure 18: The Organizer window (in Windows)

Note: Screen shots

As you can see, screen shots of the Organizer window are shown for each platform, UNIX and Windows. From now on, screen shots will only be shown for one of the platforms, provided they contain the same information for both platforms. This means that **the layout and appearance of screen shots may differ** slightly from what you see on your computer screen.

Only if a screen shot differs in an important aspect between the platforms will two separate screen shots be shown.

Preferences

What You Will Learn

- To set up and save preferences
- The basics of the graphical user interface in the SDL Suite, operated from the mouse and the keyboard. You will learn to:
 - Use graphical lists
 - Use pull-down menus
 - Use pop up menus
 - Use quick buttons
 - Use the status bar
 - Use option menus
 - Use text fields
 - Use slide bars
 - Use keyboard accelerators

What Are Preferences for?

Before starting creating your first SDL diagram, you should set up some preferences to match your computer environment. These preferences affect the default behavior of the SDL Suite tools and should be adjusted to convenient values in order to have SDL Suite function properly (most options may be set as preferences). When SDL Suite is installed, the factory settings are used as preference settings. Your system manager may have already prepared the environment for you; good advice is to check this anyway.

At least the following should be checked:

- The help preferences
- The printer preferences
- The drawing area size
- The platform mode.

Displaying and Changing Preferences

To view and possibly change the preferences:

1. From the Organizer's *Tools* menu, select the *Preference Manager* command. The Preference Manager window is displayed:

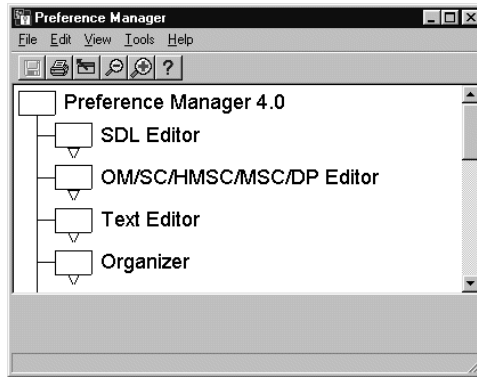


Figure 19: The Preference Manager window

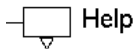
Your next task will be to check and, if required, modify a few preferences.

Help Preferences

The SDL Suite supports a context-sensitive online help facility that you may use at any moment to request help on a window, on a command, on a dialog etc.

On **UNIX** the online help is in HTML-format and you may use Firefox, Netscape Navigator or Internet Explorer as help viewer. You can change the help viewer by setting a preference.

To set up the Help preferences:



1. Locate the icon titled *Help* and double-click it.
 - You can also right-click the icon and select *Expand* from the pop up menu.

This will expand the list structure below and make the Help preferences visible.



2. Locate the preference *HelpViewer*.

To the right of the preference icon you can see the current value, the currently saved value and an explanatory text.



HelpViewer

3. Change the preference by selecting the *HelpViewer* icon and then selecting a new viewer in the option menu at the bottom of the window. (You may need to ask your system manager if you do not know what viewer to use). The icon will turn gray, which means that the preference has been changed and needs to be saved.

4. Depending on your choice of viewer, you should now check that the command used when starting the help viewer is correct, according to your computer environment:
 - Locate and select the icon titled , *NetscapeCommand* or *InternetExplorerCommand*. The current value is shown to the right. If it is not correct, change the text in the text field at the bottom of the window. (You may need to ask your system manager about the correct value.)

 A small icon representing the Netscape browser, with the word "netscape" written below it.

5. Collapse the *Help* icon by double-clicking it.

You have now learned how to work with graphical lists. Graphical lists are used extensively throughout the tools; they may hold as many levels as required (the Preference Manager uses three levels of indentation, as seen on the screen).

Some tools also support a vertical tree as an alternative to a graphical list. The functionality is identical, only the presentation differs. You will acquaintance yourself with a graphical tree later in this tutorial.

Setting the Default Printer

In this tutorial, you will learn how to print diagrams. Before you start printing, you should check and, if needed, set up your print preferences in accordance to your computer environment.

To set up the Print preferences:

1. Locate the *Print* icon. Expand it.
2. Locate the *PrinterCommand* preference. Adjust it to an adequate value (if required, ask your system manager). You may specify any suitable operating system command, for instance sending the resulting printouts to a printer queue (the command `lp`) or previewing a PostScript file in a pre-viewer such as Ghostview¹.

1. Ghostview: A user interface for ghostscript. 1992 Timothy O. Theisen.

3. Locate the *PaperFormat* preference. The SDL Suite supports a number of predefined paper sizes on the option menu (A4, A3, US Letter and US Legal). You may also specify an arbitrary *UserDefined* value, in which case you also need to specify the preferences *UserDefinedWidth* and *UserDefinedHeight*; these values are expressed in millimeters.
4. Adjust, if required, the preferences *MarginUpper*, *MarginLower*, *MarginLeft* and *MarginRight*. These preferences govern how much space in millimeters will be reserved for the margins on the printed pages; you may use this space for including headers and footers in your printouts.
 - To adjust these preferences, select them and drag the slider for a coarse adjustment. Terminate by clicking left or right of the slide bar or on the arrows to adjust in smaller steps.
5. Adjust, if required, the preference *Landscape* to on or off (this preference specifies the orientation, landscape or portrait).

42



Setting the Drawing Area Size

When editing SDL diagrams, the pages are assigned a predefined size. You should specify the default size to match the size of the printer pages and the printer margins that you defined in the previous exercise.

1. Locate the *SDL Editor* icon (at the very top), expand it and inspect the preferences *PageWidth* and *PageHeight*.
2. If required, adjust these preferences to suitable values.

Saving the Preferences

You should now save your preference settings for future sessions.

1. Select the *Save* command from the *File* menu.



- Alternatively, you may click the quick button for *Save*. Quick buttons are located in a *tool bar* which may be found immediately beneath the menu bar. Quick buttons are mouse accelerators for frequent commands and are available in all SDL Suite and TTCN Suite tools, not only in the Preference Manager.
- You may “preview” the functionality that a quick button provides by pointing on the quick button; the status bar (situated at

the bottom of the window) displays an explanatory text. If you let the mouse pointer rest on the quick button, a short “tool tip” text is also displayed just below the button.

- Another possibility is to type the keyboard accelerator for the *Save* command, by pressing `<Ctrl+S>`. This is indicated immediately to the right of the menu choice *Save*.
2. You will receive a warning that the preferences you have changed will not take effect until the individual tools are restarted (exited and started again). Just click *OK* to acknowledge this.

Your preferences are now saved on file for the current (and for future) sessions.

3. Close the Preferences window by selecting the *Exit* command from the *File* menu.

This concludes your Preference session. You may of course at any moment go back to the Preference Manager and adjust other preferences.

Creating an SDL Structure

You are now ready to create your first SDL diagrams.

What You Will Learn

- To customize the Organizer chapters
- To create an SDL structure
- To add a system root node
- To create a system diagram
- To add a page
- To edit a system diagram
- To save a diagram on file
- To save a diagram structure on a system file
- To work with dialogs (modal and modeless)
- To work with tree structures

Customizing the Organizer Chapters

When the SDL Suite is started, the Organizer displays two icons that symbolizes the system file and the source directory for your diagrams. The system file will be explained later. The source directory is where the SDL Suite components will look for existing diagrams, and save newly created diagrams. (The source directory can of course be changed.)

The source directory should already be set to the directory that you started the SDL Suite from (`~/demongame` **(on UNIX)**, or `C:\IBM\Rational\SDL_TTCN_Suite6.3\work\demongame` **(in Windows)**).

By default, the Organizer also shows 5 areas in its window:

- *Analysis Model*
- *Used Files*
- *SDL System Structure*
- *TTCN Test Specification*
- *Other Documents*.

These areas are known as *chapters*. You may use the chapters to hold a number of diagrams and documents; the actual use is a matter of personal taste and the default is to be regarded as a suggestion. As you will design a rather simple system, we suggest that you start by removing the chapters *Analysis Model*, *Used Files* and *TTCN Test Specification*.

To remove the chapters:

1. Select the chapter *Analysis Model*.
2. Select the menu choice *Remove* from the *Edit* menu. (You may also press the button on the keyboard.)
 - A dialog opens – confirm the removal by clicking the *Remove* button in the dialog.

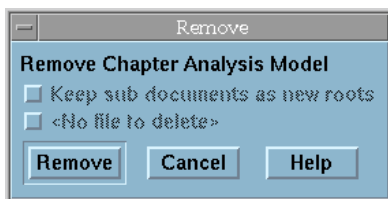


Figure 20: Confirming to remove a chapter

3. Repeat the steps above for the chapters *Used Files* and *TTCN Test Specification*.

You may also rename the remaining two chapters:

1. Select the chapter *SDL System Structure*.
2. Select the *Edit* menu choice from the *Edit* menu. (You may also double-click the chapter.)
3. In the dialog that opens, make sure the option *Edit chapter symbol* is selected and click the *Edit* button.

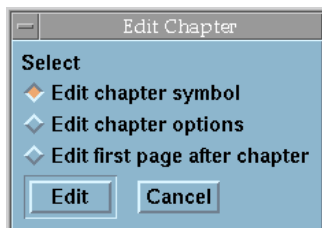


Figure 21: Editing the chapter symbol

Creating an SDL Structure

4. Change the document name in the opened *Edit* dialog; for instance to **My first SDL system**. Do **not** change the document type indicated by the *Organizer* button and the option menu value *Chapter*.

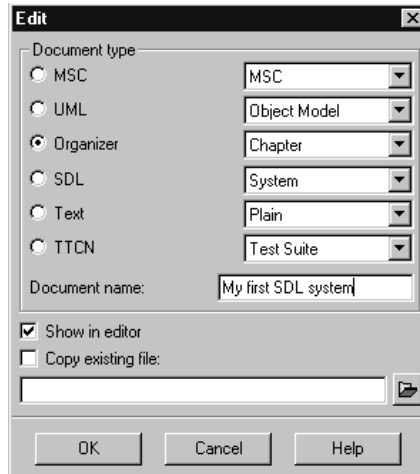


Figure 22: Naming the chapter

- **On UNIX**, you may note that the cursor changes to the shape of a question mark as soon as it points on the parent *Organizer* window. This convention has been adopted to indicate that a dialog must be closed before any other operation is allowed to take place in the tool. Dialogs that need to be answered before proceeding further are called *modal* dialogs.
5. **Turn off** (uncheck) the option *Show in editor*.
 6. Terminate by clicking the *OK* button.
 - If you like, also rename the *Other Documents* chapter. This chapter will be used later in this tutorial to hold diagrams that are not part of the SDL system but that you will want to keep track of.

Creating a System Diagram

Adding a Root Node

You will now create an SDL system, working in a top-down fashion:

1. Make sure the chapter *My first SDL system* is selected.
2. Select the *Add New* command from the *Edit* menu. The *Add New* dialog opens, prompting you to specify the name and type of diagram to add.

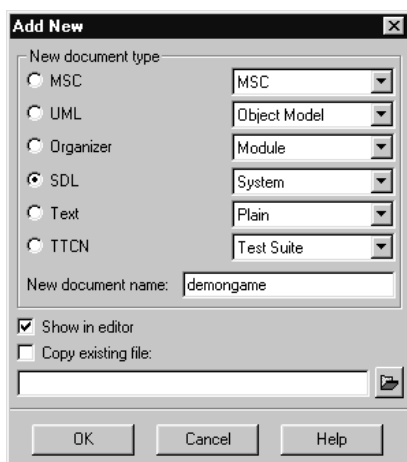


Figure 23: Adding a new diagram

3. Specify the *New document type* as *SDL*, and specify the SDL diagram type as *System*, as depicted above.
 - If the SDL diagram type shows something else than *System* and thus needs to be changed, click the option menu to adjust it.
4. Specify the *New document name* as **DEMONGAME** (the default name, *Untitled*, disappears).
 - You may need to point and possibly click with the cursor on the text field to set the focus on it.
5. Make sure the *Show in editor* button is turned **off**.

6. Click the *OK* button.
 - The dialog disappears and the Organizer window is updated with a root node — system DemonGame. Note that the diagram is identified as being [unconnected], meaning that there is no connection to a physical file.

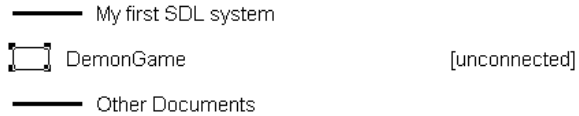


Figure 24: The new root node

Creating the System Diagram

You have so far created an Organizer *diagram structure*, consisting of one *reference* to an SDL system diagram (the referred diagram does however not yet exist).

Your next task is to create the system diagram:

1. Select the DemonGame SDL system diagram icon. See [Figure 24](#).
2. From the *Edit* menu, select the menu choice *Edit*.
 - You may also press the **right** mouse button while pointing on the icon — a popup menu appears — select the sub-menu *Edit* and the menu choice *Edit*.
 - Another way to edit the diagram is to double-click the icon with the left mouse button.
3. The *Edit* dialog opens, suggesting to create a new diagram and to *Show the diagram in editor*. (The dialog is very similar to the Add New dialog you just used.) Accept the suggestion by clicking *OK*.

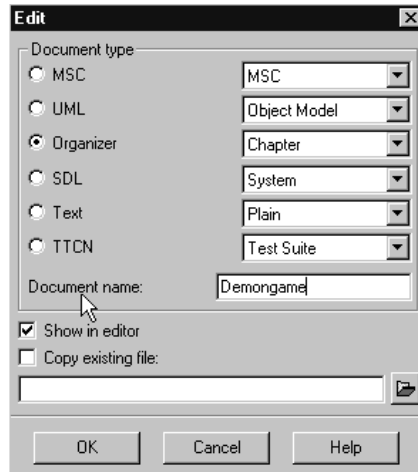


Figure 25: Prompting to create a new diagram

The SDL Suite responds by displaying the *SDL Editor window*, showing the upper left corner of page 1 of the system diagram DemonGame.

The SDL Editor is the tool you use when editing the contents of the diagrams. The SDL Editor is also used for building the diagram structure that is displayed in the Organizer window.

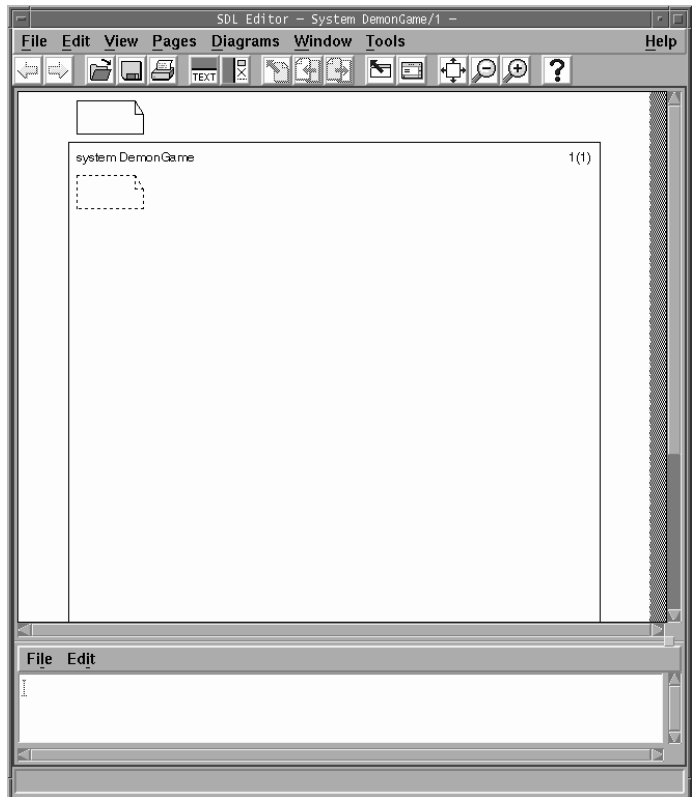


Figure 26: The SDL Editor window (on UNIX)

Your next task is to fill in the contents of the diagram. [Figure 27](#) shows the appearance of the diagram when completed and printed on paper. As you can see, the diagram consists of two block reference symbols (GameBlock and DemonBlock), a channel conveying the signals between the blocks (C3) and two channels conveying the signals to and from the environment (C1 and C2). There is also a text symbol where the signal declarations may be found.

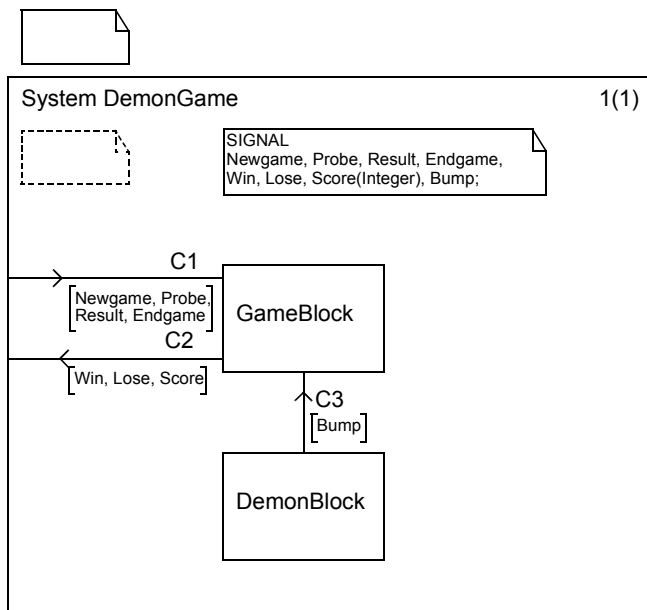


Figure 27: The system diagram

The next pages describe in detail how you proceed to add the symbols and texts to the diagram.

Customizing the SDL Editor Window

Before you start editing, you may want to resize the editor window. You may also hide and show various sub-windows using the command *Window Options* from the *View* menu.

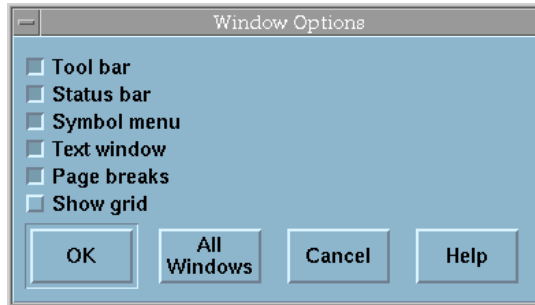
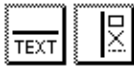
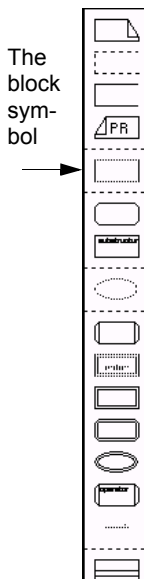


Figure 28: The window options



- You can hide and show the editor *Tool bar*, *Status bar*, printed *Page breaks* and the *grid* points.
- You may also hide and show the *text window* and the *symbol menu* by clicking on the provided quick buttons. You will however need these windows soon.

Placing Block Reference Symbols



1. Start editing the diagram by inserting the two block reference symbols (GameBlock and DemonBlock).

To place a block reference symbol:

- Click on the block symbol in the *symbol menu*. **On UNIX**, the symbol menu is located to the extreme right of the window. **In Windows**, the symbol menu is a separate window always placed on top of the SDL Editor window (if the two windows overlap).

If you are not sure what symbol to use, point to or select a symbol in the symbol menu – its type is displayed in the Status Bar at the bottom of the SDL Editor window.

- Move the mouse into the drawing area. The symbol “floats” and follows the mouse. Click to position the symbol where you want it to be. **No overlap** between symbols is allowed. (In case symbols are overlapping, an alert sound is emitted and you have to repeat the operation.)

Note: Aborting and undoing

- To abort the insertion of a symbol after you have moved the mouse into the drawing area, just press <Esc>.
- If you happen to perform a command or operation that you wish not had taken place, you should immediately select the *Undo* command from the *Edit* menu.

2. Once you have placed the symbol, type the name of the block:
GameBlock or **DemonBlock**
 - You can type in text directly at the cursor's position. The cursor position can be set by clicking on the text in the symbol. However, you cannot select (highlight) text directly in the symbol.
 - Before you have started text editing, the text cursor is not flashing. Pressing <Delete> at this stage deletes the whole selected symbol. Once text editing has started, the text cursor is flashing and pressing <Delete> only deletes a character.
 - You may see a red underlining appearing in the text, if you enter a name that has incorrect syntax according to SDL. This is the general way to indicate textual syntax errors in the SDL Editor.
 - When you edit text, you may also take advantage of the *text window*, which allows you select text by dragging. **On UNIX**, the text windows is located below the drawing area. **In Windows**, the text windows is a separate window always placed on top of the SDL Editor window (similar to the symbol menu).
 - No matter where you enter the text, the text is always displayed both in the symbol and in the text window.
 - You may also note that the Organizer diagram structure is automatically updated to reflect the insertion of the diagram reference symbol (once the symbol is de-selected).

Moving and Resizing Symbols

To move a block:

- Select and *drag* the block with the mouse to the desired location. (Remember, no overlap with other symbols is allowed).

After you place the blocks where you want them, you may resize them:

- Point to one of the symbol's corners, and drag. You must be fairly close to the corner. If this method fails, first select the symbol with a click and then repeat the procedure while pointing to a selection square.

Drawing Channels between Blocks

To draw a channel from block DemonBlock to block GameBlock:

1. Select the DemonBlock symbol. A “handle” appears.

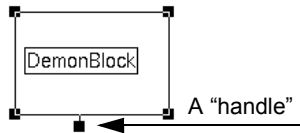


Figure 29: A block symbol's “handle”

2. Drag the handle (i.e. press the mouse button while pointing on the handle, and start moving the mouse while keeping the mouse button pressed).
3. As soon as mouse motion has begun the editor responds by drawing a line; from now on you may release the button while moving the mouse.
4. Move the mouse until it points to the GameBlock symbol. Click the mouse button; the channel is connected at both ends.
 - You may move the channel's endpoints individually by dragging them. Select the channel first if the endpoint is difficult to “hit” with the mouse.
 - You will notice a tiny selection square at the middle of some lines drawn in the SDL Editor. This can be used to create “breakpoints” on the line, thus dividing the line into different line segments. You will not use this feature in this tutorial.

The SDL Editor creates two *text attributes* associated to the channel. These text fields are displayed as *selection rectangles* which you use when entering the name of the channel and the list of the signals the channel is to convey. Initially, when the text attributes are empty, a red underlining is shown to indicate that this is not allowed according to the syntax rules of SDL.

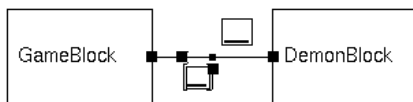


Figure 30: The channel's text attributes with red underling

The two blocks have been aligned horizontally to more easily distinguish the two text attributes.

To fill in the name of the channel C3:

- Type it directly, immediately after the channel has been drawn. (If the channel has become de-selected, select it again.)

To fill in the signal Bump into the signal list text field:

1. Click on the text field surrounded with two brackets '[]'. (Click in the space between the brackets.)
2. Type the name of the signal. Note that the brackets are adjusted automatically to fit the size of the text.
3. You may move the text attributes to new locations, if desired. Simply drag them with the mouse.

Drawing Channels to the Environment

To draw a channel from a block to the environment (e.g. C2):

1. Select the block.
2. Start by dragging the handle, and terminate by clicking on the frame symbol (the rectangle that encloses the diagram, see [Figure 32 on page 64](#)).

3. Fill in the name and the signals.
 - Note that, as you type the signal list, the red underlining changes and shows if and where the text is not syntactically correct according to SDL. As soon as you have entered the signal names separated with commas, the red underlining disappears and shows that the text now is syntactically correct!
 - The SDL Editor allows you to leave a text containing syntax errors. However, it is not possible to build an SDL system that contains syntax errors.

Drawing a Channel from the Environment

To draw a channel from the environment to a block (e.g. C1):

1. Start by drawing the channel from the block to the environment, as you just learned.
2. Make sure the channel is still selected.
3. Then, select the command *Redirect* from the *Edit* menu. Fill in the name and signals the usual way.
 - You may press <Return> to insert line breaks within the signal list, if it becomes too long.

Drawing a Text Symbol

The diagram also contains a text symbol with the required signal declarations.

1. Pick the text symbol in the symbol menu (the top symbol), insert it into the drawing area and fill in the contents as shown in [Figure 27](#). The built-in syntax check is even more evident in this case.
2. When the contents of the text symbol are changed, the editor automatically resizes the text symbol to fit the text. You may resize it by dragging the lower right corner, or toggle between its minimized and maximized sizes by double-clicking the symbol. Try this.
 - The selection squares at the other three corners of the text symbol are gray. This means that the symbol cannot be resized by dragging any of these corners.

Resizing the Text Window

If the text window is too small to bring all the text in view, you may resize it. In **Windows**, this is done in the same way as any normal window. On **UNIX**, this is done by dragging the *sash* up or down; the sash is the small square situated to the right and above the text window menu bar; the text window is a pane of the SDL Editor window.

You may drag the sash up or down to resize the text window

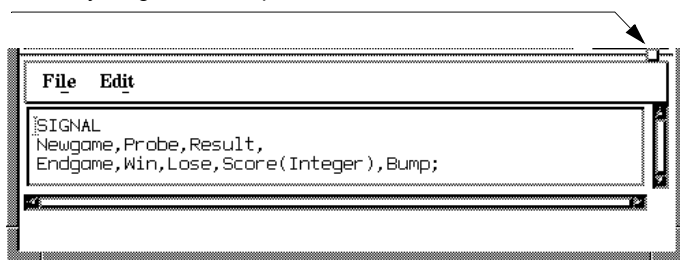


Figure 31: The SDL Editor's sash (UNIX only)

Other Items in the System Diagram

Except for SDL symbols, a diagram also contains the following:

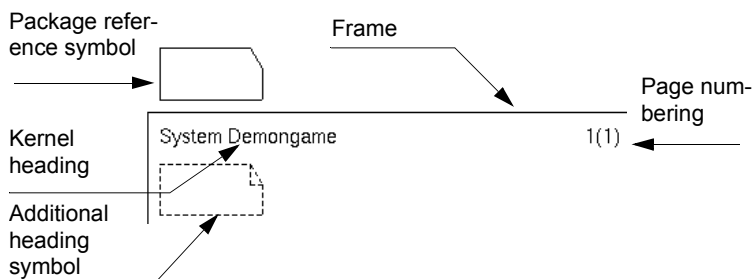


Figure 32: Other symbols

Package Reference Symbol

The package reference is used to refer to included SDL packages. This simple example does not include any packages. Just leave it empty.

The Kernel Heading

The kernel heading is automatically assigned its contents by the editor to reflect the type and the name of the diagram being edited. The kernel heading is editable, but you are not going to alter its contents in this tutorial.

Additional Heading Symbol

The additional heading symbol is not defined further according to Z.100. In the SDL Editor, it looks like a dashed text symbol. The symbol is editable and may be resized the same way as you learned for resizing text symbols, but it cannot be moved. Its intended use in the SDL Editor is, among others, to define inheritance and specialization and to specify formal parameters. You will not use this symbol in this first tutorial.

Frame

The frame surrounds the objects that are contained in your diagram. You may want to resize the frame to create a more compact diagram: simply drag any corner to do this.

Note:

The frame is not the same as the paper border!

Page Numbering

The page numbering is updated automatically, and reflects the name of the page and the total number of pages. It is not editable.

Saving the Newly Created System Diagram

In this exercise, you will learn the commands that store SDL diagrams on files.

1. You should now have two windows on the screen, the Organizer window and the SDL Editor window.
 - To find the Organizer window, you may at any time select the command *Show Organizer* from the SDL Editor's *Tools* menu, or click the *Show Organizer* quick button.
2. Before you save anything, open the Organizer's *View Options* dialog from the *View* menu.

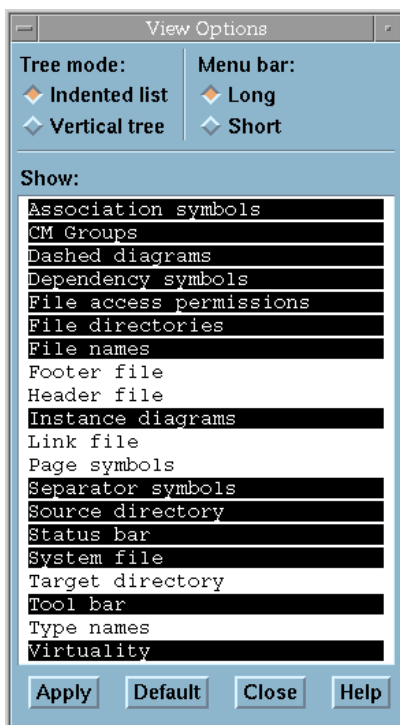


Figure 33: The Organizer's View options

Creating an SDL Structure

3. Make sure the options are in accordance to [Figure 33](#) and click the *Apply* button. This makes, among other things, the file and directory names visible in the Organizer.
 - The list in the dialog is a *multiple selection list*. When you click on an item in the list (**in Windows while holding down the <Ctrl> key**), its selected state is toggled without affecting any other item. This makes it possible to select any number of items in the list. **In Windows**, if you by mistake click on an item without using the <Ctrl> key, you may press *Default* to get back to the default settings.
 - To close the *View Options* dialog, click the *Close* button. This kind of dialog is modeless, meaning that it remains open until you decide it is longer needed and close it. You are not forced to close a modeless dialog to continue working with the tool, in opposite to modal dialogs, such as the *Add New* dialog which you used for creating a new system (see [Figure 23 on page 54](#)).
4. Look at the resulting Organizer view. The system diagram icon is drawn with a gray pattern, which shows that the diagram is modified and not saved. The name of the diagram (i.e. *DemonGame*) is shown in bold face, to indicate that the diagram is currently open in an editor. The text to the right of the icon reads [unconnected] which is a convention adopted to show that a diagram has no current binding to a file.

There are two other diagram icons, which are [unconnected]. These represent the references to the block diagrams that you added when editing the system diagram.



Figure 34: A modified, unconnected diagram (DemonGame)

5. Now, go back to the SDL Editor and save the SDL diagram by selecting the *Save* menu choice from the *File* menu.
 - To locate the SDL Editor window from the Organizer, you may double-click the icon for the system diagram again, which simply raises the SDL Editor window.
6. A file selection dialog is displayed. This is a generic dialog that opens whenever you are prompted to specify a file (to open, to save, etc.). The title of the dialog shows the nature of the operation, *Save* in this case.

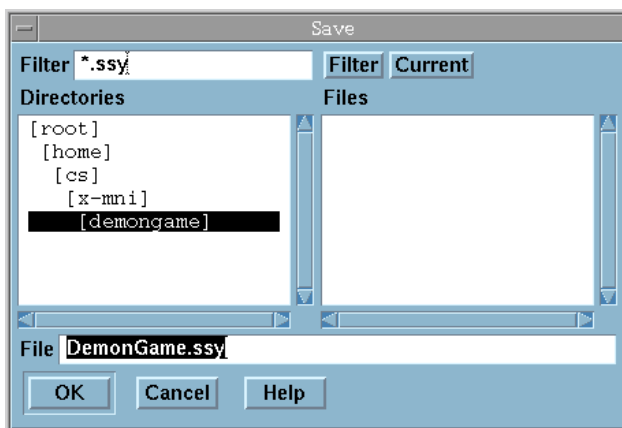


Figure 35: A file selection dialog (on UNIX)

On UNIX, this dialog works as follows:

- The *Filter* field is preset to **.ssy*, which is the default file extension for files that contain SDL system diagrams. To list other files, you have to change the contents of the *Filter* field and click the *Filter* button (but do not do this now).
- The right list shows a list of files that match the file filter. It should be empty since you have not created any diagrams yet.
- The left list shows the directory structure from the root node of the file system down to the current directory. You may double-click here in order to navigate in your directory structure (do not use this list now).

Creating an SDL Structure

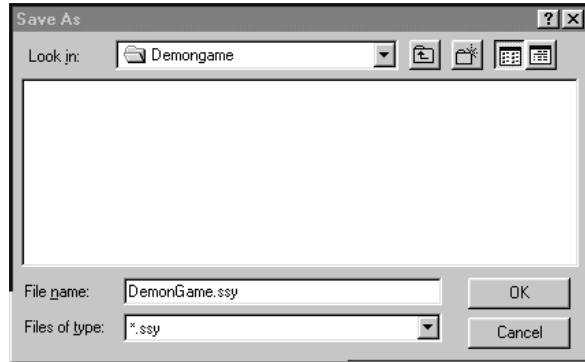


Figure 36: A file selection dialog (in Windows)

In **Windows**, this dialog works as follows:

- The *Files of type* field is preset to *.ssy, which is the default file extension for files that contain SDL system diagrams. To list other files, you have to change the contents of the *File name* field and click the *OK* button (but do not do this now).
 - The list shows a list of files that match the file filter. It should be empty since you have not created any diagrams yet.
 - The *Look in* list shows the directory structure from the root node of the file system down to the current directory. You may click here in order to navigate in your directory structure (do not use this list now).
7. The SDL Editor suggests a file name to store the diagram on: `DemonGame.ssy`. You may change to any file name; we assume however in this tutorial that you accept the suggested file name.

So, simply click the *OK* button to accept the file name. The diagram is now stored on file. If you look at the title of the SDL Editor window, you may see that the diagram has been saved on file.

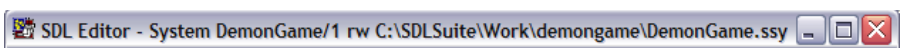


Figure 37: The SDL Editor window title

This information is also available in the Organizer structure, where the file name has changed from [unconnected] to `DemonGame.ssy`.



Figure 38: The diagram structure after saving the diagram

Saving the Diagram Structure

You have, so far, saved the system diagram. You should also save the Organizer's view options and diagram structure for future sessions. If you look at the Organizer's window title, you notice an ending asterisk. This asterisk denotes that the Organizer's view or structure information has been modified and needs to be saved.

The System File

The Organizer saves its view, along with a number of options, on a dedicated file called the system file¹. System files are used as a means to maintain the consistency of an SDL structure and provide immediate access to the diagrams that are defined in the structure.

The system file is represented by its own icon at the top of the Organizer view, a rectangle with "SDT" in it. Even though the system file has not yet been saved, the Organizer has assigned a file name for it.

1. A system file may contain information related to any kind of SDL structure, not necessarily an SDL **system**. The term system file is a general term.

Creating an SDL Structure

To save the system file:

1. Select the *Save* command from the Organizer's *File* menu. The Organizer responds by issuing the Save dialog.



Figure 39: The Organizer's Save dialog

2. The tool suggests a file name to store the information on: demongame.sdt (system files are by default assigned the extension .sdt). Accept the suggestion by clicking the *Save* button.

Once a system file has been created, the diagram structure and the Organizer options are saved for future sessions. You *Open* an existing system file from the Organizer's *File* menu.

More About Saving

For the purpose of this tutorial, you have learned how to save individual diagrams and how to save the system file. There are however other handy ways to save everything with one single command. Two of these methods are listed below.

- You may click the *Save All* button in the Organizer's Save dialog (see [Figure 39](#)).
- You may click the quick button for *Save* on the Organizer's tool bar. This button orders a global and silent save of all diagrams (no prompting will be issued unless special cases need your attention), including the diagram structure. (The SDL Editor's quick button for *Save* saves the current diagram only.)



Printing the System Diagram

What You Will Learn

- To print one SDL diagram
- To adjust print options
- To scale a printout

How to Print

You have now drawn your first SDL diagram. It may be convenient to print the diagram before proceeding with the remaining exercises. **On UNIX**, we assume that your computer environment includes a PostScript printer. If not, you may skip this exercise.

To print the diagram:

1. Raise the SDL Editor window.
2. Select the *Print* command from the *File* menu. The *Print* dialog is opened.

- You may instead click the quick button for *Print*.

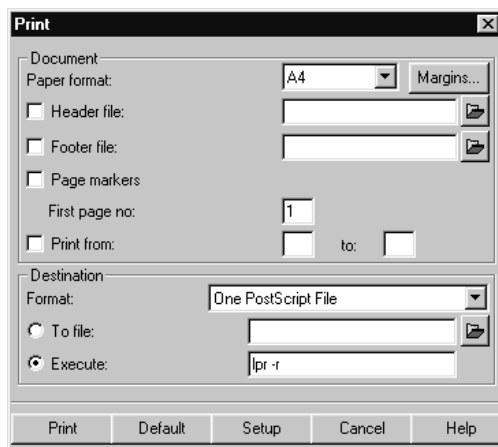


Figure 40: The Print dialog

Printing the System Diagram

3. If you have set up your preferences adequately, the dialog should be preset with the correct options. If not, you need to check and possibly modify at least the:
 - *Paper format* option menu
 - *Destination Format*: if you do not have access to a PostScript printer in **Windows**, you may select *MSW Print*.
 - *Execute* command (governed by the preference *PrinterCommand*).
4. Once the settings look OK, order the printout by clicking the *Print* button. **On UNIX**, a PostScript file will be generated on `/tmp` and the file will be piped to the *Execute* command that you have specified. **In Windows**, a print file will be generated by using the *Execute* command that you have specified to print it, or by using the default printer driver if you have selected MSW Print format.

If the file is sent to a printer queue, the printer should respond almost immediately. If you are not satisfied with the size of the resulting printout, you may scale it as follows:

- Click the *Setup* button. The *Print Setup* dialog is opened.

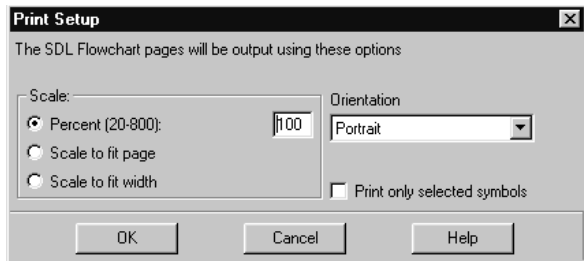


Figure 41: The *Print Setup* dialog

- Adjust the scale to the value of your choice and click *OK*.
- Order the printout once more by clicking *Print*.

Checking the System Diagram

What You Will Learn

- To invoke the Analyzer
- To set analysis options
- To work with the Organizer Log window
- To locate and correct syntax errors

Running the Analyzer

You should now check the syntax of the system diagram you created before proceeding further by creating the remaining diagrams. To do this, you will use the *Analyzer* tool, a back-end tool which is fully integrated with the Organizer.

1. Select the SDL system diagram icon in the Organizer diagram structure. Then, from the Organizer's *Generate* menu, select the *Analyze* command.
2. If you had (perhaps accidentally) modified any diagram, the Organizer first prompts you to save modified diagrams (by issuing the *Save* dialog, see [Figure 39 on page 71](#)) — in which case you should click the *Save All* button to make sure everything is OK.
3. Once the *Save* dialog is closed, the *Analyzer* dialog is opened.

Checking the System Diagram

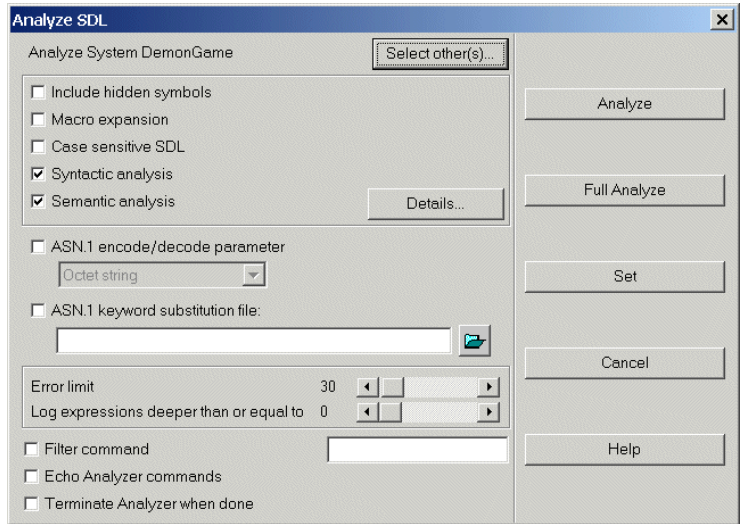


Figure 42: The Analyzer Options dialog

4. Adjust the options in accordance to [Figure 42](#), i.e.
 - *Macro expansion* off
 - *Syntactic analysis* on
 - *Semantic analysis* **off**
 - Adjust, if required, the *error limit* slide bar to a reasonable value. This parameter defines how many errors and warnings the Analyzer will report before aborting the analysis.

5. Click the *Analyze* button. The Analyzer now starts processing the input with the options as specified in the options dialog. When it is finished, the Organizer status bar should read something with the essence “Analyzer done”, possibly appended with extra information.
6. The Organizer is provided with a textual window where important information is logged. By default, the Organizer Log window is raised as soon as information classified as “warning” or “error” is reported. If the window does not appear automatically after the analysis is complete, open the window manually. (Use the *Organizer Log* command from the Organizer’s *Tools* menu to show the window, or the provided quick button).



Looking for Analysis Errors

The diagnostics that are reported by the Analyzer are appended to the Organizer Log (together with other important messages). Look at the tail of the log for the report summary, which should look something like:

```
-----  
Number of warnings: <diagram dependent>  
+ Analysis completed
```

(You may need to scroll down the Organizer Log window to bring the tail into view.)

The text “Number of warnings” or “Number of errors” shows how many syntactic warnings or errors that were detected in the diagram (if no warnings or errors were found, then these lines are missing altogether).

1. For the purpose of this exercise, you may need to introduce a syntactic **error** into the diagram. You may for instance remove one of the separating commas in the signal list of channel C1 (but make sure there is a space separating the signals).
 - Such a syntactic error will be detected already in the SDL Editor and marked with a red underline, but we will show how the error is reported by the Analyzer.
2. Save everything and repeat the analysis.

Correcting Analysis Errors

Your Organizer log should now report an error looking something like:

```
#SDTREF(SDL,/opt/home/tmi/demongame/DemonGame.ssy(1)
,131(25,50),3,8) (on UNIX)
#SDTREF(SDL,C:\IBM\Rational\SDL_TTCN_Suite6.3\work\d
emongame\DemonGame.ssy(1),131(25,50),3,8) (in
Windows)
ERROR 312 Syntax error in rule SIGNALLIST, symbol
Name found but one of the following expected:
, ; comment
Result Endgame;
?
```

How to Interpret the Error Message

Let us spend a few moments on explaining the contents of this error.

- The first part (#SDTREF . . .) is a reference to the source diagram, page, symbol, line number and finally a position within a line of text where the error was found. All references produced by the Analyzer adhere to this format in its whole or partially; the reference may in some circumstances be less precise than in the example above, depending on the Analyzer's ability to locate the exact source of error.
- The second part (ERROR 312 . . .) contains the error number and an explanatory text, telling you, in this case that a comma, a semicolon or a comment was expected.
- The last part (Result Endgame) along with the '?' character shows more specifically where the error occurred, in this case the comma should be inserted between the signals Result and Endgame.

To display the diagram and symbol where the error was found, you may use a handy facility:

1. Select, by dragging the mouse, the lines of text containing the error message.



```
#SDTREF[SDL,C:\sdt31\work\demongame\DemonGame.ssy(1),132[17,52],2,8]
ERROR 312 Syntax error in rule SIGNALLIST, symbol Name found but one of the
following expected:
, ; comment
Result Endgame;
?
```

Figure 43: Selecting the error message



2. Select the command *Show Error* from the *Tools* menu.
 - Alternatively, you may click the *Show error* quick button.
3. The symbol where the error was found is immediately selected in the SDL Editor. The more information the reference holds, the more precise the selection. Correct the error (insert the comma).
4. To correct the next error, simply click the *Show error* button again.
5. Save the diagram and repeat the analysis until the Analyzer does not report any errors. If you feel uncertain about how to interpret and correct the errors, look at the printout for the system diagram for a reference (see [Figure 27 on page 58](#)).



- You may clear the Organizer Log window at any time, for instance between subsequent passes to make it easier to read the contents of the log. Use the *Clear Log* command from the *Edit* menu for this, or the provided quick button.



- For repeated analysis passes using the same options, you can use the *Analyze* quick button in the Organizer.

You have now designed your first SDL diagram using the SDL Suite. You have also verified that the diagram is syntactically correct according to the Z.100 recommendation. Congratulations!

Creating a New Block Diagram

What You Will Learn

- To create and draw a block diagram
- To request signal dictionary support
- To work with multiple diagrams using the SDL Editor
- To open new windows on a page
- To work with multiple SDL Editor windows
- To perform syntax check on a block diagram

Creating a Block Diagram from the Organizer

In this exercise, you will create a block diagram, starting from the Organizer.

1. Locate the Organizer window and double-click the icon named GameBlock. In a similar fashion as when creating the system diagram, you will get the *Edit* dialog (see [Figure 25 on page 56](#)). Make sure the *Show in editor* option is on and click the *OK* button.

Next, the *Add Page* dialog is opened. The dialog is used to specify the type of page (process or block interaction). The page name can also be specified. Decide if you want the pages to be autonumbered. (1, 2,... N). This functionality is enabled by default.

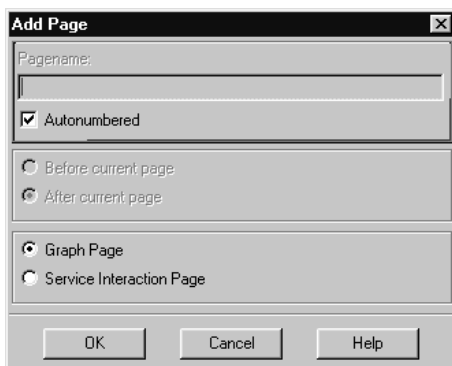


Figure 44: Prompting to add a page

2. Make sure the *Process Interaction Page* button is on and click *OK*.

The SDL Editor opens a window on page 1 of the newly created block diagram. The block diagram editor window is similar to the system diagram window; only the symbol menu differs.

[Figure 45](#) shows the appearance of the finished block GameBlock when printed. As you may see, the diagram contains two process reference symbols, five signal routes, three connection points and a text symbol with a signal declaration.

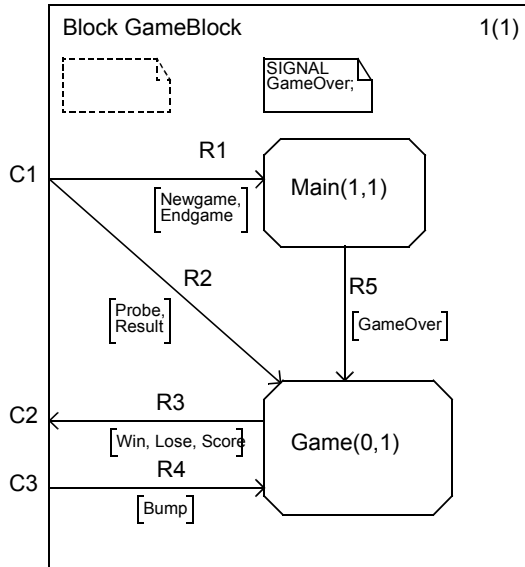


Figure 45: The block GameBlock

You should now draw the block diagram as depicted in [Figure 45](#). You add the symbols and lines in a similar fashion as when editing the system diagram. Please spend a minute reading the three sub-sections below before starting drawing the diagram, in order to familiarize yourself with the new concepts that are introduced and how you manage them.

Process Name and Number of Instances

When you add a process reference symbol, you should specify the number of instances by appending the text directly after the name of the process reference symbol. The number of instances is the text between parentheses '()'.

Signal Routes

Signal routes are drawn in a similar way as channels. When you select a process reference symbol, **two** “handles” are displayed.

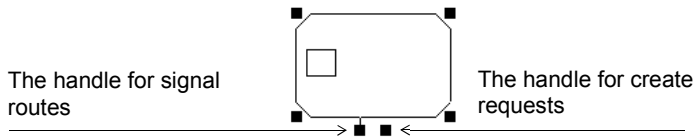


Figure 46: The two handles of a process reference symbol

- The left handle is used for drawing signal routes, in a similar fashion as channels.
- The right handle is used for drawing create requests. It is not used in this tutorial.

Connection Points

When you draw signal routes to / from the frame symbol, you should not only fill in the name and signal list, but also take advantage of graphical *connection points* to establish connections between the signal routes and the parent system diagram, i.e. connecting the signal routes to the channels. When you draw a signal route to the frame, an additional text object is created close to the frame symbol. In this text object, the name of the corresponding channel is entered. See [Figure 47](#).

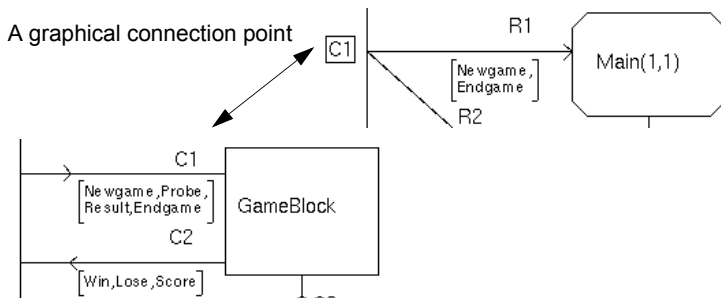


Figure 47: Graphical connection point

The figure depicts a signal route (R1) in the block diagram and the referencing system diagram with the connected channel (C1).

To edit a connection point:

- Simply select it and enter its textual contents.

Editing the Block Diagram

Now, draw the diagram as described in the following steps:

1. Start by adding the process reference symbol “Main(1,1)”.
 - As when drawing the system diagram, all text you enter is subject to an immediate syntax analysis. Errors are indicated by a red underlining, which disappear as soon as you have entered the complete text according to the SDL syntax.
 - Remember that before text editing has started, the text cursor is not flashing. Pressing <Delete> at this stage deletes the whole selected symbol, instead of just a typed character.
2. Draw a signal route **from** the environment to the process (use the *Redirect* command). Enter the name of the signal route: R1.

Using the Signal Dictionary for Individual Signals

You are now to specify the name of the signals to be conveyed on the signal route R1. The SDL Editor has the ability to assist you in reusing the signals that are already defined in the SDL structure (i.e. defined in the system diagram, since you are working in a top-down fashion!), with a facility known as the signal dictionary.

1. Select the signal list text field.
2. Select the command *Signal Dictionary* from the *Window* menu. The Signal Dictionary window is displayed. If necessary, move it so that you can see the signal list in the Editor window.

Note:

To function properly, the Signal Dictionary utility requires that the input SDL diagrams are syntactically correct. If not, you need to go back to the previous exercise (see [“Correcting Analysis Errors” on page 77](#)) and run the Analyzer in order to correct any errors.

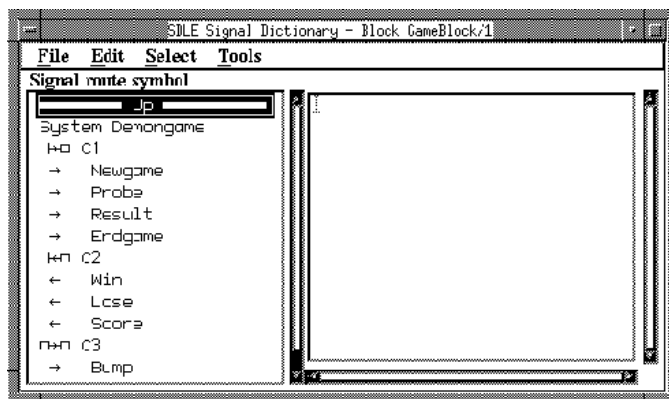


Figure 48: The Signal Dictionary window (on UNIX)

The exact appearance of the list to the left depends on the graphical capabilities available on your terminal.

- Look at the left list in the Signal Dictionary window. The first section starts with the separator *Up*. This section includes the signals that are available by looking one level up in the diagram structure (i.e. in the parent diagram, system *DemonGame*).



Figure 49: The Up separator (on UNIX)



Figure 50: The Up separator (in Windows)

- In this section, the first item identifies the *System DemonGame*, as expected:



Figure 51: The item symbolizing the system *DemonGame* (on UNIX)



Figure 52: The item symbolizing the system *DemonGame* (in **Windows**)

5. Since you are editing a signal route from the system diagram to the block diagram, you should look for all icons/lines symbolizing channels from the parent diagram, i.e. IN channels.



Figure 53: The icon symbolizing a channel from the parent diagram to the current diagram (on **UNIX**)

CH C1:In

Figure 54: The line specifying a channel from the parent diagram to the current diagram (in **Windows**)

- You should find exactly one channel that matches the criteria: C1. The remaining channels are either internal (C3) or are not directed into the current diagram (C2).
6. Beneath the channel all the signals that it conveys are listed. Click on the signal named *Newgame*:



Figure 55: The *Newgame* signal (on **UNIX**)

->-- **Newgame**

Figure 56: The *Newgame* signal (in **Windows**)

7. From the *Edit* menu, select the command *Insert*. The signal list in the SDL Editor is immediately updated.
 - Alternatively, you may double-click the signal in the list.

Note: Undoing the operation

If you happen to insert the signal into the wrong text field (such as the signal route name), you may select the *Undo* command from the Signal Dictionary's *Edit* menu. (The SDL Editor's Text Window has no *Undo* facility).

8. Insert the required comma and a newline in the Editor window (move the signal list if needed).
9. Double-click the signal Endgame in the Signal Dictionary.
10. The channel in the Signal Dictionary also gives a suggestion about how to fill in the connection point: C1. Select the connection point text field in the diagram and double-click the channel in the Signal Dictionary.

Using the Signal Dictionary for Multiple Signals

1. In the SDL Editor window, add the process reference symbol: Game(0,1).
2. Draw the signal route R3 from Game to the frame symbol.

When you are to enter the signal list for R3, you do not need to enter the signals one by one as for R1, since the channel C2 is not split up into multiple signal routes, in the way that C1 becomes R1 and R2 (see your printout or [Figure 27 on page 58](#)). Instead, you may insert all signals with one single operation:

3. Select the channel C2 in the Signal Dictionary. The right list is updated to list all signals conveyed on the channel. Insert them with the *Insert* command (or double-click the channel C2).

Creating a New Block Diagram

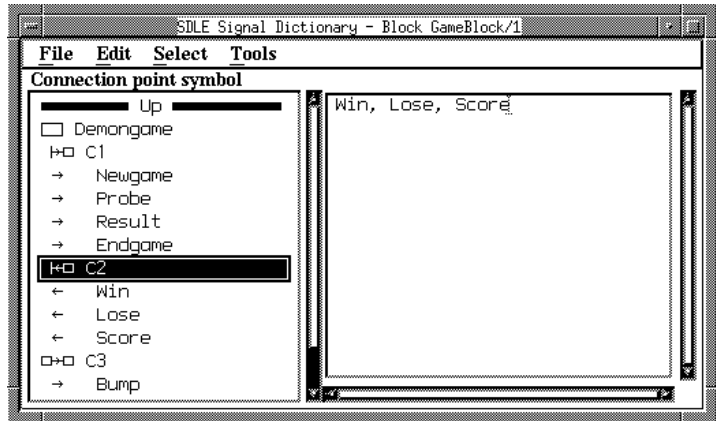


Figure 57: Selecting a channel in the Signal Dictionary window lists all signals (on UNIX)

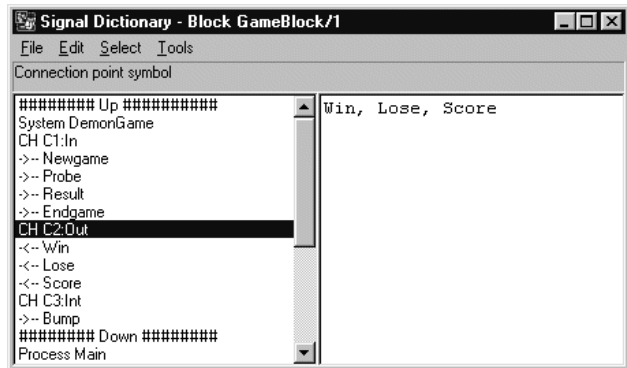


Figure 58: Selecting a channel in the Signal Dictionary window lists all signals (in Windows)

You are now somewhat familiar with the Signal Dictionary utility.

Completing the Diagram

1. Fill in the remaining parts of the diagram using your preferred method. When done, you may close the Signal Dictionary window (use the *Close* command from the *File* menu).
2. Once you are finished with the block GameBlock, save everything, for instance by clicking on the Organizer's *Save* quick button. Accept the default file name suggested by the Save dialog.

Working with Multiple Diagrams

You have now created two SDL diagrams. Both diagrams are currently opened by the SDL Editor; however only the diagram currently being edited is visible in a window.

The SDL Editor provides a menu named *Diagrams* where all diagrams and pages currently opened by the editor are listed.

1. Click on the *Diagrams* menu. It should now list **two** diagrams:

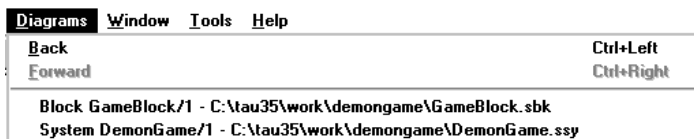


Figure 59: The Diagrams menu

Each of these menu choices correspond to a diagram and page currently opened by the SDL Editor. The file the diagram is stored on is also displayed to the right of the diagram name.

Now, bring the diagram for the system DemonGame into view:

2. Select the menu choice
*System DemonGame/1 DemonGame.ssy*¹
 The system diagram is instantly displayed. (The block diagram is now hidden.)
 - The menu choices *Back* and *Forward* can also be used to switch between diagrams and pages. The SDL Editor keeps track of



1. The exact appearance of the menu choice depends on the directory structure you are working on.

which pages you have edited and you can go back and forward in this list, in much the same way as for visited Web pages in a Web browser. There are also two quick buttons for this.

Working with Multiple Windows

So far, you have only worked with one single window on a page. The SDL Editor allows you to open new windows on the same diagram, which makes it possible to work on multiple views on a page. This is also called *instantiating* a window.

To open a new window on a page:

1. Make sure the page whose window is to be instantiated is the page currently in view in the SDL Editor. If not, use the *Diagrams* menu.
2. From the *Window* menu, select the command *New Window*. A new window showing the current page is instantly displayed (see [Figure 60](#)).
3. You may now use any window to work on the page. Any change causes both windows to be simultaneously updated. Try this, for instance by moving a symbol!
4. You probably need one window only for this tutorial, since the diagrams you are working on in this tutorial are small, on purpose. Close any of the two windows with the *Close Window* command from the *Window* menu.

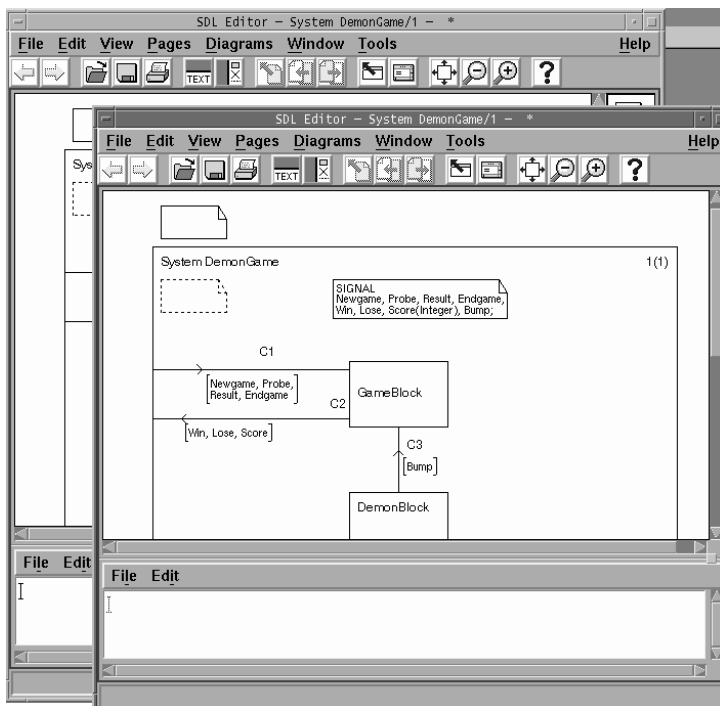


Figure 60: Two windows of the same page

Resulting Organizer View

Save everything. The resulting Organizer View should now look like:



Figure 61: The resulting Organizer view

Checking the Syntax of the Block Diagram

You may now want to use the Analyzer to check the syntax of the block diagram you just created.

To analyze the block GameBlock, do as follows:

1. Select the icon for the block GameBlock to specify the block as input to the Analyzer.
2. Select the command *Analyze* and analyze the block diagram.
 - Note that the Analyze dialog lets you select which part(s) of the system you wish to analyze. Make sure the top text in the dialog says *Analyze Block GameBlock* before clicking *Analyze*.
3. Proceed as for the system diagram, i.e.
 - Look for any syntax errors in the Organizer Log.
 - Correct these errors.
 - Repeat the procedure if required (see [“Correcting Analysis Errors” on page 77](#) if you do not remember how to do this).

Creating a Block Diagram From a Copy

What You Will Learn

- To create a diagram from an existing copy
- To save a diagram on a new file

Creating the Block DemonBlock

You created the block GameBlock from the Organizer by double-clicking the symbol. You may also do this from within the SDL Editor, by double-clicking on diagram reference symbols.

To create the block DemonBlock:

1. Locate the block reference symbol DemonBlock in the SDL Editor. Double-click on the reference symbol. A dialog is opened.

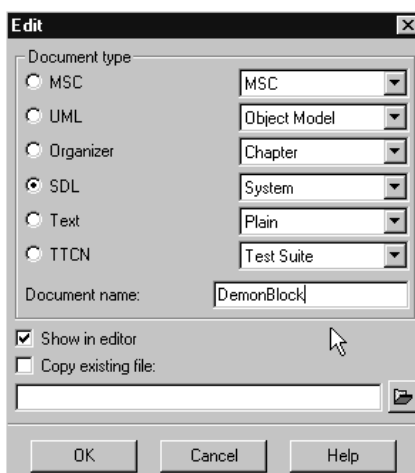


Figure 62: Prompting to create the block DemonBlock

You will now create the diagram by using a copy of an existing file. The installation contains a number of SDL examples, among which the diagrams that build up the DemonGame example may be found. These diagrams are by default stored in a subdirectory to the installation directory. The name of the directory should be `$stelelogic46/sdt/examples/demongame` (**on UNIX**), or `C:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\examples\demongame` (**in Windows**).

Your next task is to specify the location of the file that contains the block DemonBlock. This file is named `DemonBlock.sbk`

Note:

You may need to contact your system manager to find out the exact location of the directory mentioned above. If you fail in finding the directory with the DemonGame example, do not give up! You may always create the remaining diagrams with the New option, and design them with the SDL Editor in a hand-drawn fashion, as you learned in the previous exercises.

2. Make sure the *Copy existing file* button is turned on.
3. Then, either:
 - Type in the file name, **including the directory path** (according to above),

or
 - Click on the *folder button* to the right of the text field. A file selection dialog (with the title *Select File to Copy for Block DemonBlock*) is displayed.
 - In the dialog, navigate in the directory structure until you have located the directory where the diagrams are stored (see the directory path above). **On UNIX**, you double-click the directory names in the left list and find any block diagram files in the right list. **In Windows**, you select directories from the *Look in* box or double-click directories in the list.
 - Select the file `DemonBlock.sbk` that appears in the list and click *OK*.

Note: Accessing files on other disks

In Windows, you may use so-called UNC paths to access network disks by using the syntax `\\disk\directory\` when typing the path to file names.

On UNIX, the possibility to change to the `[root]` directory using the file selection dialog may or may not work properly, depending on your computer system and network file system. You may need to type in a leading slash (`'/'`), followed by a name, then click *OK* in order to access files that are stored on another disk than the one you are currently working on (but try first to double-click the `[root]` directory).

- 4. Close the *Edit* dialog by clicking on the *OK* button. The SDL Editor shows the diagram in a window. The diagram when printed should look like [Figure 63](#).

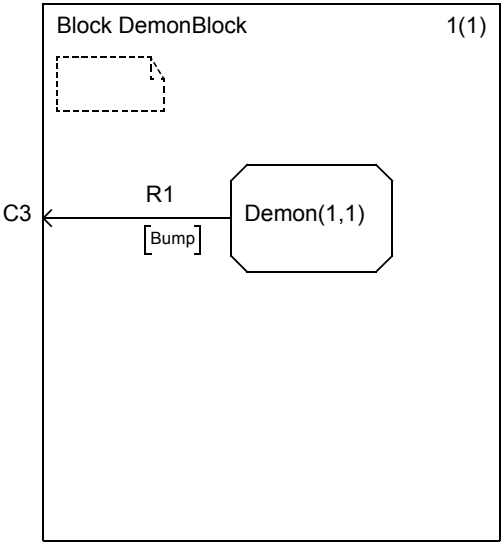


Figure 63: The block DemonBlock

- 5. Now, save the diagram from the SDL Editor (use the *Save* quick button). A file selection dialog is displayed, with the suggested file name `DemonBlock.sbk`.
- 6. Accept the suggestion by clicking the *OK* button. The resulting Organizer’s diagram structure should be as follows:

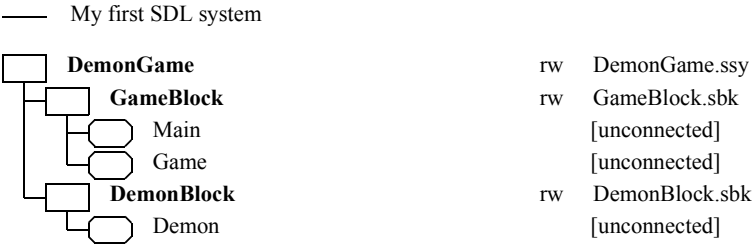


Figure 64: The resulting Organizer list

Creating a Block Diagram From a Copy

You may use the method described above to take a copy of an existing diagram for any diagram in the remainder of this tutorial. However, we recommend that you draw the diagrams from scratch to get yourself acquainted with all editing features of the SDL Editor. How to draw a process diagram (described next) is somewhat different from drawing a block diagram.

Creating a Process Diagram

You have now created the structural elements of your SDL system. This structure needs now to be completed with the implementation, i.e. the process flow charts that describe the behavior of the system.

In the previous exercises, you have learned how to create new diagrams, either from the Organizer or from the SDL Editor, so we will not focus on these details any more. Feel free to double-click icons in the Organizer or in the SDL Editor, or to use the Organizer *Edit* command, depending on your preference.

In the next exercise, you will instead learn how to use the SDL Editor for drawing process diagrams. Let us start with the process Demon, which is depicted in [Figure 65](#).

Editing the Process Demon

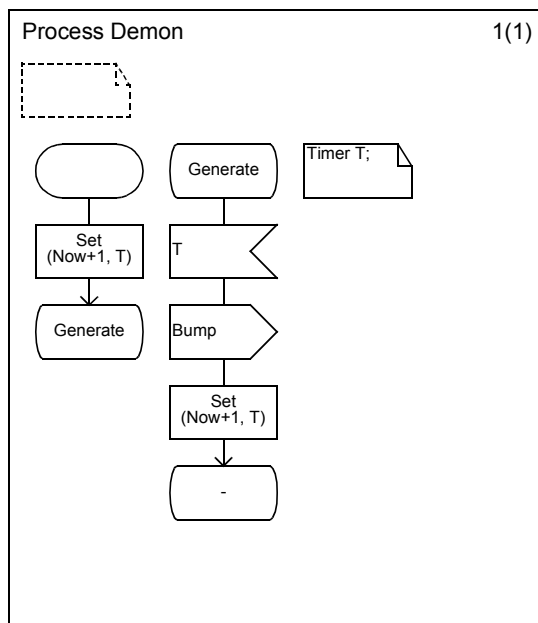


Figure 65: The process Demon

Creating a Process Diagram

On the next pages, you will find suggestions about how to use the SDL Editor to draw the diagram.

What You Will Learn

- To add symbols with the double-click facility
- To work with the clipboard functions
- To insert symbols in a flow
- To request grammar help

Creating the Diagram

1. Edit the Demon diagram. When you are prompted to add a page, make sure that you specify a page with the type set to *Graph Page*.

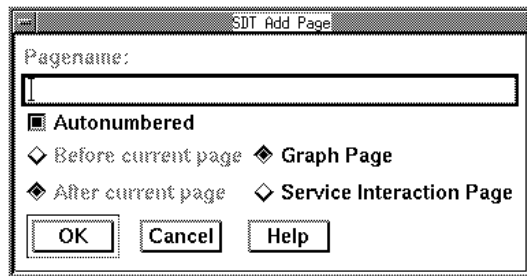


Figure 66: Specifying page type to graph page

2. When the SDL Editor responds by displaying the (empty) diagram, you notice that the appearance of the symbol menu is different; it now contains the symbols that are allowed on a flow diagram (such as state and input symbols).

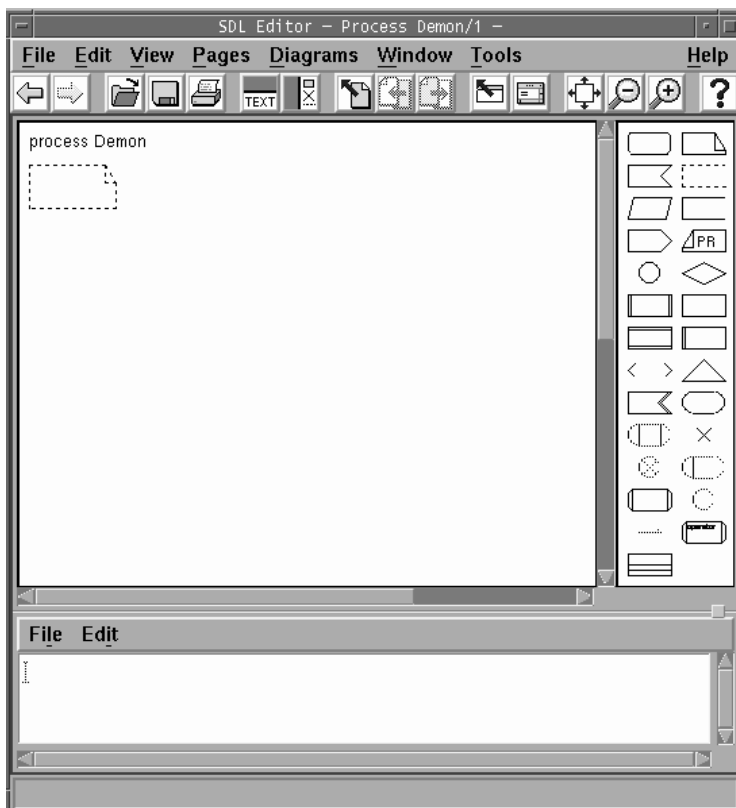


Figure 67: The SDL Editor window for flow diagrams (on UNIX)

The diagram consists of two branches of symbols (see [Figure 65 on page 96](#)). When you append symbols to a branch, the editor may automatically interconnect the symbols with flow lines.

You may select to enter the text into each symbol once the symbol has been inserted, or insert all symbols and then edit the text, or a mix of both methods.

Creating the Left Branch with Grammar Help

To create the left branch:

1. Select the start symbol in the symbol menu and place it in the drawing area at a suitable location.
 - Remember that when you point to or select a symbol in the symbol menu, its type is displayed in the Status Bar at the bottom of the SDL Editor window. Use this if you are not sure what symbol to pick.
2. Double-click the task symbol in the symbol menu. An empty task symbol should now be appended to the start symbol.

When you are to edit the task symbol containing the statement that sets the timer, let us assume, for the purpose of this exercise, that you do not have the grammar for the Set statement in mind.

The SDL Editor provides a context-sensitive facility, the *Grammar Help window*, that assists you in entering correct SDL expressions. You will now use it in order to fill in a correct set expression.

3. Select the command *Grammar Help* from the *Windows* menu. The SDL Editor responds by displaying the *Grammar Help window*.

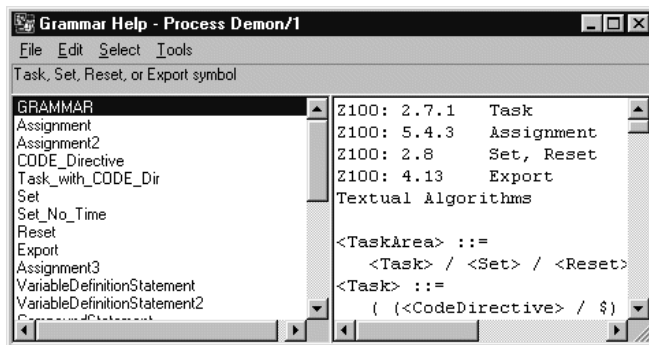


Figure 68: The Grammar Help window

- The left list shows a number of “use cases”, each of them identified with their name. The first one is the GRAMMAR for the selected object.
 - The top of the right list shows a number of references to the Z.100 definition.
 - Beneath the Z.100 references are listed the formal textual (SDL/PR) expressions that are legal to add to the symbol. (The formal expressions need of course to be replaced by the actual values that are used in your context).
4. The use case you are to use is the set of a timer, so locate the item titled *Set* in the left list and select it.
- The right list is updated to reflect the formal grammar for the expression: “SET(Now+Expr, TimerName)”.

Creating a Process Diagram

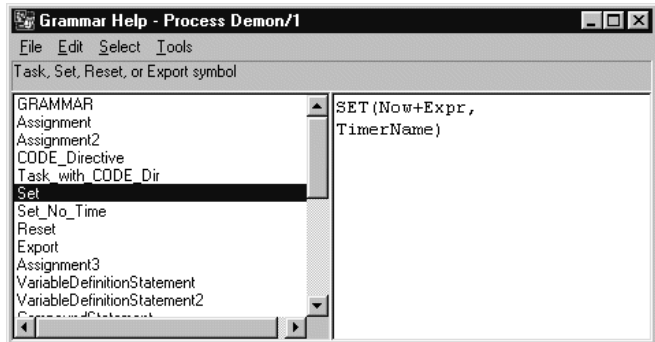


Figure 69: The grammar for a Timer Set

5. Insert the formal text into the task symbol by selecting the *Insert* command from the *Edit* menu. The task symbol is immediately updated.
 - You may also double-click the *Set* item in the list.
6. Now, change the generic names *Expr* and *TimerName* to their actual values (*I* and *T*, respectively).
 - You use the SDL Editor's text window for this. Drag for instance the mouse over the text to be changed and type in the new text to substitute it with.

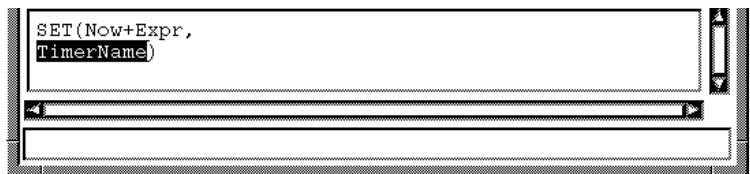


Figure 70: Edit the text in the text window (on UNIX)

You have now learned the basics about how to work with the Grammar Help.

7. To finish the left branch, double-click a state symbol and enter the text: **Generate**

Creating the Right Branch

To create the right branch:

1. Copy the newly added state symbol to the clipboard. You find the clipboard commands, e.g. *Copy*, on the *Edit* menu or on the pop up menu that is activated with the right mouse button.
2. Paste the state symbol. Following *Paste*, you should specify the location of the new symbol; move the mouse until you point to a suitable location and terminate with a click with the left mouse button.
3. Append an input symbol with a double-click. Enter the text: **T**
4. Append the output of the signal Bump with a double-click and enter the text **Bump**.
5. Copy the task symbol with the text “SET (Now+1, T)” to the clipboard. But, **do not** paste right now.
6. Point to the output symbol Bump. Press the right mouse button and select the *Insert Paste* command. This pastes and connects the task symbol.
7. Terminate the branch by double-clicking a state symbol and typing a hyphen (-).
8. Finally, add a text symbol and type in the declaration of the timer T.
9. If desired, resize the frame symbol.
10. *Save* the diagram with the file name `Demon.spr`.

This concludes the editing of the process Demon.

Editing the Process Game

First, create the process diagram Game in the usual way. In this exercise, you will learn some other editing functions:

What You Will Learn

- To edit parallel flow branches
- To interconnect symbols

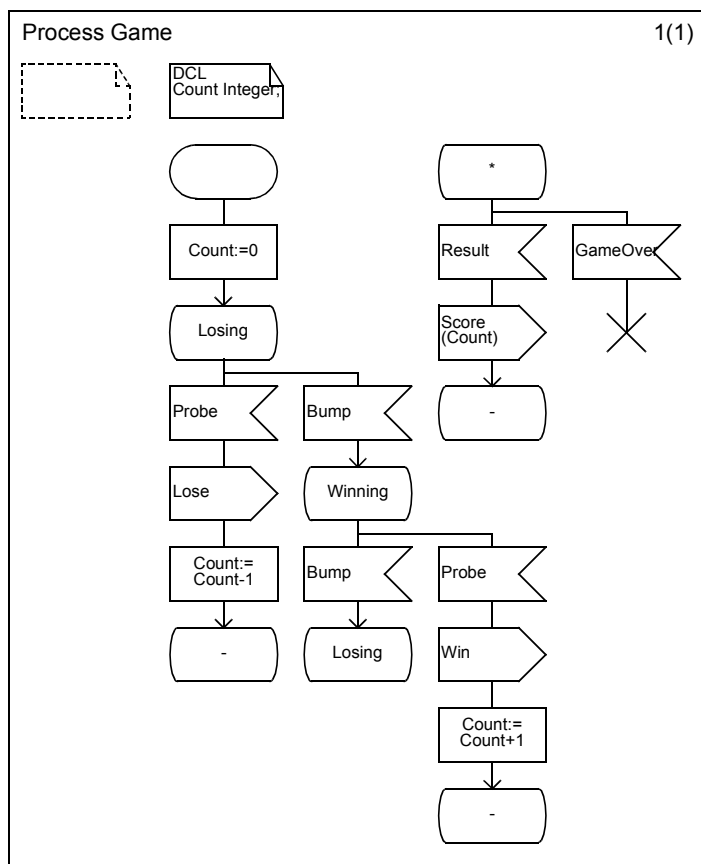


Figure 71: The process Game

You may proceed editing the process diagram in [Figure 71](#), as will be described below:

Editing the Start Transition

1. Insert the start symbol, the following task symbol and the state symbol Losing.
2. You will now insert two input symbols in parallel. To do this, first make sure the state symbol is selected. Then, press <Shift> and double-click two input symbols (<Shift> must be kept pressed while you do this).
3. Release <Shift> and select the left input symbol.
4. Fill in the name of the input symbol (**Probe**), and complete the left branch.
5. Select the right input symbol, fill in the name (**Bump**) and complete the branch without bothering about the subbranch that starts with the input of the signal Probe in the state Winning.
6. Select the Probe input symbol in the left branch. On the *Edit* menu, use the *Select Tail* command to extend the selection to the end of the branch.
7. *Copy* the selection and *Paste* it. Move the selection (which appears as a set of symbols) to a suitable place and paste it with a click with the left mouse button. If *Paste* fails (because of insufficient space or overlap), an alert sound is issued — please try again.
8. Change the text in the input symbol from Lose to **Win**.
9. Change the text in the task symbol to **Count:=Count+1**.
10. To interconnect the state symbol Winning with the input symbol Probe: select the state symbol – a handle appears –

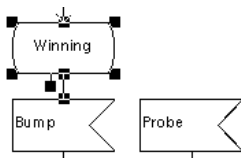


Figure 72: The selected state and its handle

drag the handle while pressing the mouse and release the mouse when it points to the input Probe symbol. A line is drawn between the symbols:

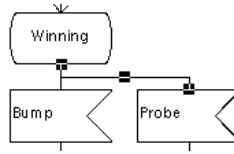


Figure 73: The two branches are connected

11. Conclude the diagram by drawing the remaining parts and saving the diagram.

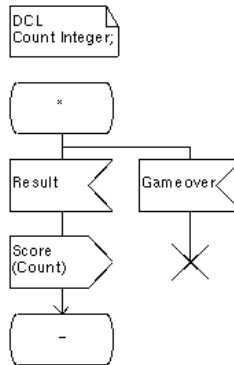


Figure 74: Remaining parts to edit

Editing the Process Main

The process Main is the last diagram to create and edit. If you find this tedious, you may skip this exercise and create the diagram as a copy from the files that are enclosed in the distribution (how to do this is described in section [“Creating a Block Diagram From a Copy” on page 91](#)). [Figure 75](#) shows the appearance of the diagram to create.

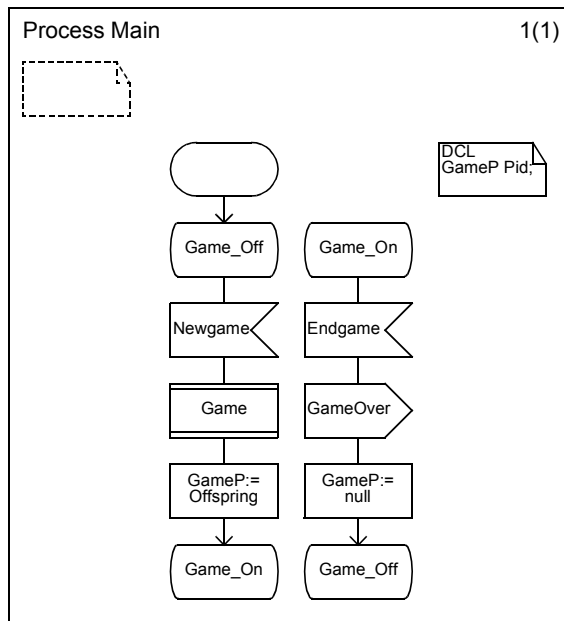


Figure 75: The process Main

More About the Organizer

When you are ready with the diagram, save everything. The diagram structure in the Organizer Window should now look like this:

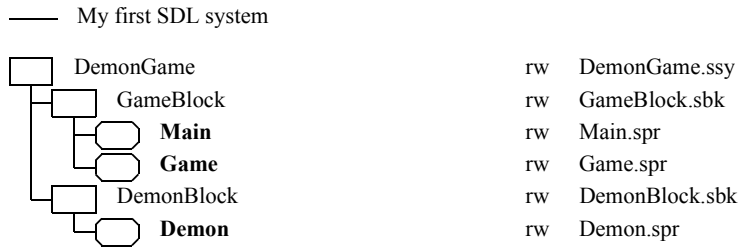


Figure 76: The resulting diagram structure

In this tutorial, you have only browsed through a minor part of the available functionality. You may for instance customize the Organizer to display the information using different view options.

What You Will Learn

- To work with vertical trees
- To expand and collapse the diagram structure
- To rearrange diagrams in an Organizer structure
- To display directories and pages
- To print the entire system

Tree View

1. Bring up the *View Options* dialog (see [Figure 33 on page 66](#)), click on the *Vertical tree* radio button and click on *Apply*. The Organizer window changes its presentation mode:

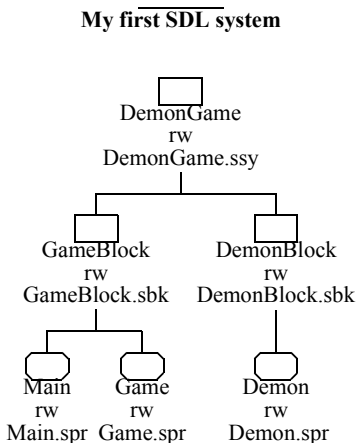


Figure 77: A vertical tree structure

2. Apply the *Indented list* mode again.

Expand / Collapse

You may make parts of the diagram structure invisible (and visible again) with the *Expand* and *Collapse* commands from the *View* menu. (*Expand* is available on collapsed nodes only (indicated by the small triangle), while *Collapse* is available on expanded nodes that have a sub-structure).

1. *Collapse* the block GameBlock. The subtree for the block GameBlock is reduced to the node, with a small triangle added.

— My first SDL system



Figure 78: A collapsed node

2. Expand the subtree again (use *Expand Substructure*).

Rearranging Diagrams

The Organizer lets you rearrange the order of appearance of symbols.

You can either do that with arrow keys (<Up>, <Down>, <Left> and <Right>) or with the quick buttons *Move up* and *Move down*.

Say that you want to rearrange the order of appearance of the blocks DemonBlock and GameBlock:

1. Select the block GameBlock.
2. Click once on the quick button *Move down*, alternatively press <Shift> and type the <Down> arrow key. The result becomes:



— My first SDL system



Figure 79: Rearranged GameBlock and DemonBlock

3. Change back to the original order of the diagrams.

Diagram Pages

1. In the *View Options* dialog, turn the *Page symbols* item on.
2. Apply the options – the result becomes a list where the SDL pages are made visible.

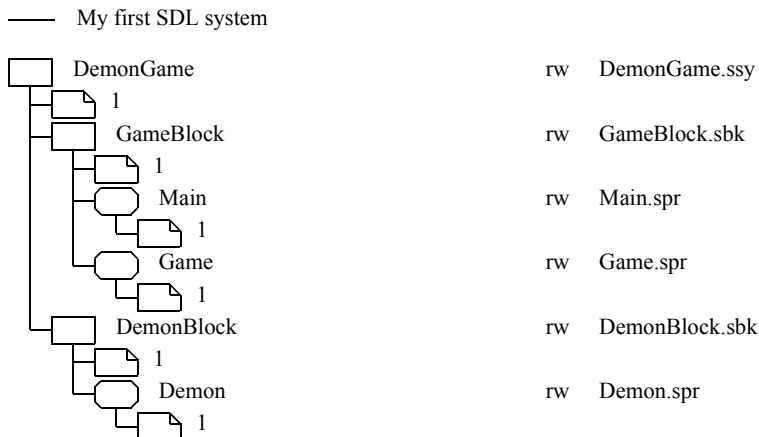


Figure 80: Diagram pages are displayed

Printing the System

The Organizer lets you print all diagrams that are included in the system with a single command. You may also include a table of contents:

1. De-select all diagram symbols, or select the system diagram.
2. Click the quick button for *Print*. This opens the Organizer's Print dialog.
3. Turn the *Table of contents* toggle button on and click on *Print* to order a global printout of all SDL diagrams, including a table of contents (see [Figure 81](#)).



More About the Organizer

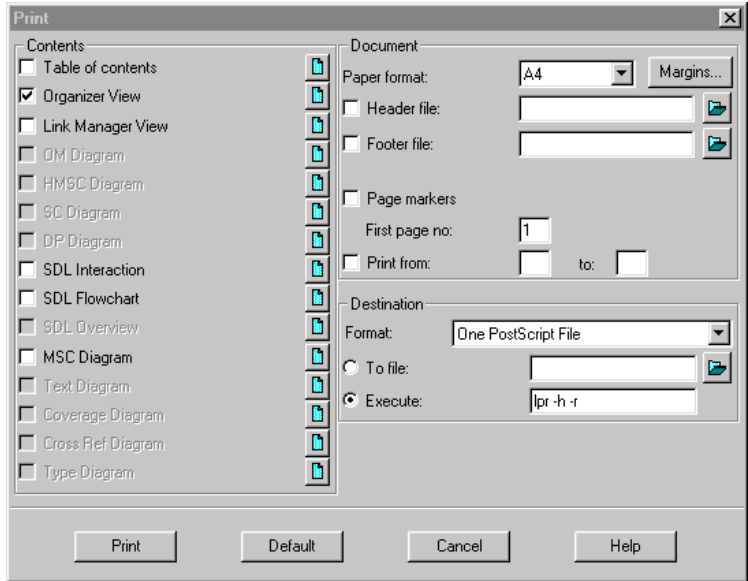


Figure 81: Including a table of contents

You have now created and printed your first complete SDL system using the SDL Suite. Your next task is to check the complete system with respect to SDL syntax and semantics.

Analyzing the Complete System

What You Will Learn

- To perform syntactic and semantic analysis on the whole system
- To generate files containing definitions and cross references

Enabling Semantic Analysis

To analyze the system, you should also enable the semantic checker. To do this:

1. Select the system diagram icon.
2. Use the *Analyze* command from the *Generate* menu.
3. Adjust the analyzer options according to the picture below:

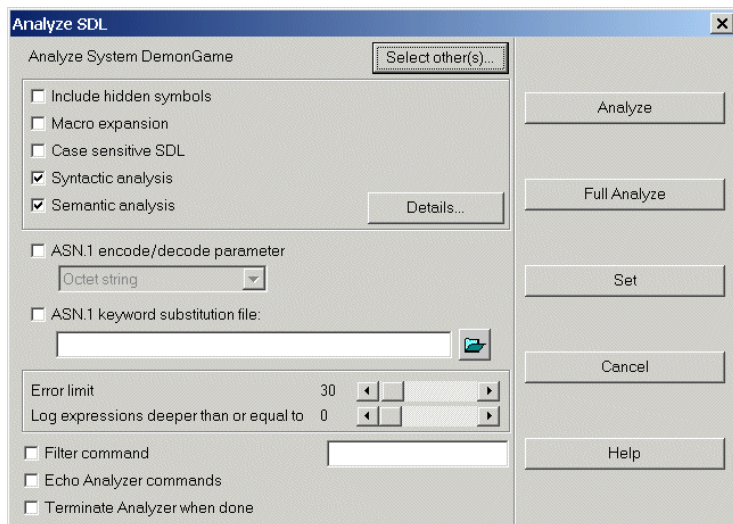


Figure 82: Including semantic analysis

Analyzing the Complete System

- Make sure the Analyzer generates a file with *cross references*, by turning the toggle button *Generate a cross reference file* on. You will need this file in a later exercise in this tutorial.
 - The semantic Analyzer has some other options, each one of these individually activated with a toggle button. They have no impact on this tutorial.
4. Click the *Analyze* button.
 5. When the Organizer status bar reads “Analyzer done”, look at the *Organizer Log* for any errors reported by the Analyzer.
 6. If required, correct the errors and repeat the procedure. How to locate errors in the source SDL diagrams was described in a previous exercise, see [“Looking for Analysis Errors” on page 76](#) and [“Correcting Analysis Errors” on page 77](#).
 7. The tail of the Organizer log should contain the following output when the system is syntactically and semantically correct:

```
+ Analysis started
Conversion of SDL to PR started
Conversion to PR completed
Syntactic analysis started
Syntactic analysis completed
Semantic analysis started
Semantic analysis completed
+ Analysis completed
```

Terminate this exercise by saving everything. You may also want to print the diagrams again (see [“Printing the System” on page 110](#) for how to do this).

Managing Message Sequence Charts

Besides the SDL tools, SDL Suite and TTCN Suite also support the Z.120 recommendation, also known as Message Sequence Charts (MSC). You should have a basic understanding of MSC symbols to fully understand this exercise.

In this tutorial we will demonstrate some application areas of MSCs.

- First, an MSC may be used for describing the requirements on the dynamic behavior of a system, viewed as a “black box” which receives external stimuli (corresponding to SDL signals issued from the environment) and respond by sending SDL signals to the environment.
- MSCs may also help you to understand a problem, by offering a way of presenting, in graphical form, some use cases which have been identified, before proceeding with the design in SDL.
- Generating MSCs as the result of a simulation of a system also provides a mean to understand the dynamic behavior and verify it against the expected behavior.
- Finally, MSCs can be input to an *SDL Explorer* where you can verify that the scenario that the MSC is describing may actually occur and under what circumstances.

What You Will Learn

- To add MSCs to the diagram structure
- To associate SDL diagrams and MSCs
- To create MSCs
- To edit MSCs

Inserting an MSC into the Organizer

To create an MSC, you use the Organizer, where the MSC will be managed as an *Other Document*. In this exercise, we will create an MSC where you will describe the dynamic behavior of the system *DemonGame*. You will also use this MSC as a reference when simulating and validating the system (this is done in later exercises).

Managing Message Sequence Charts

To create an MSC:



1. Select the Organizer chapter *Other Documents*.
2. From the *Edit* menu, select the command *Add New*, or click the quick button for this.
3. The *Add New* dialog is opened, prompting you to specify a diagram name and type.

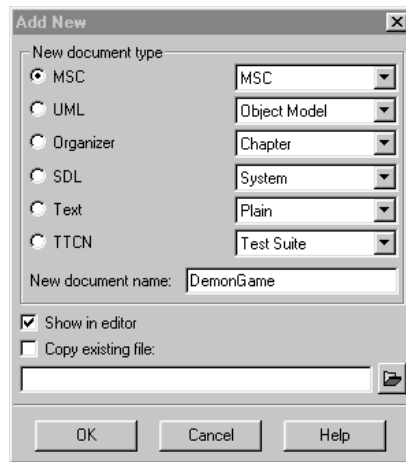


Figure 83: Specifying the name and type of the diagram to add

Adjust the dialog options as in [Figure 83](#) above:

- Set *New document type* to *MSC*
 - Change the name to **DemonGame**.
 - *Show in editor* should be turned on
4. Click the *OK* button. An *MSC icon* appears in the Organizer's *Other Documents* chapter; the lower part of your Organizer window should look like [Figure 84](#), once the MSC Editor is started (you may have to raise the Organizer window if the MSC Editor covers it).

— Other Documents



DemonGame

[unconnected]

Figure 84: The Organizer structure with an MSC added

The MSC you have inserted into the Organizer is intended to describe the behavior of the system and you will associate it with the system diagram. The association will be visible in the Organizer.

5. Make sure the MSC icon is selected and select the *Associate* command from the *Edit* menu. A dialog is displayed.

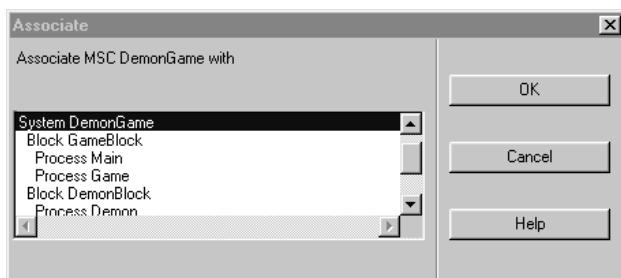


Figure 85: Associating an MSC with an SDL diagram

6. Select the system DemonGame item in the list and click *OK*.
7. Look at the resulting Organizer structure. In addition to the MSC icon in the *Other Documents* chapter, an MSC Link icon appears, connected to the system diagram icon. If you select it, the Organizer's status bar informs you about the link to the actual MSC.
 - If you do not see any MSC Link icon, check the Organizer's *View Options*, turn the option *Association Symbols* on and click *Apply*.

Managing Message Sequence Charts

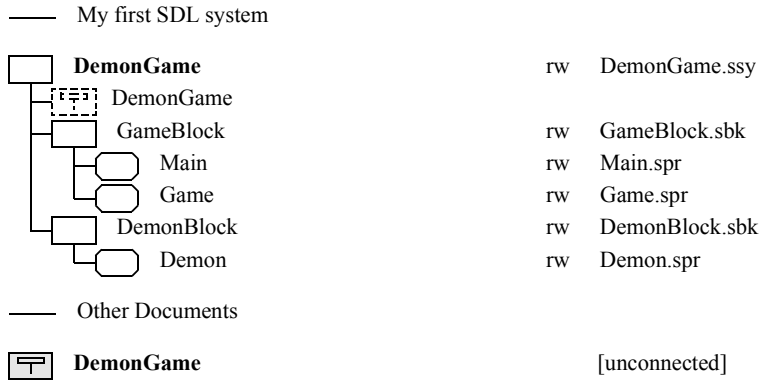


Figure 86: Association between the System diagram and the associated MSC

Editing an MSC

1. Raise the MSC Editor window for the newly added MSC symbol. The window of the MSC Editor looks similar to the SDL Editor window, but provides of course a different symbol menu and different set of commands and quick-buttons.

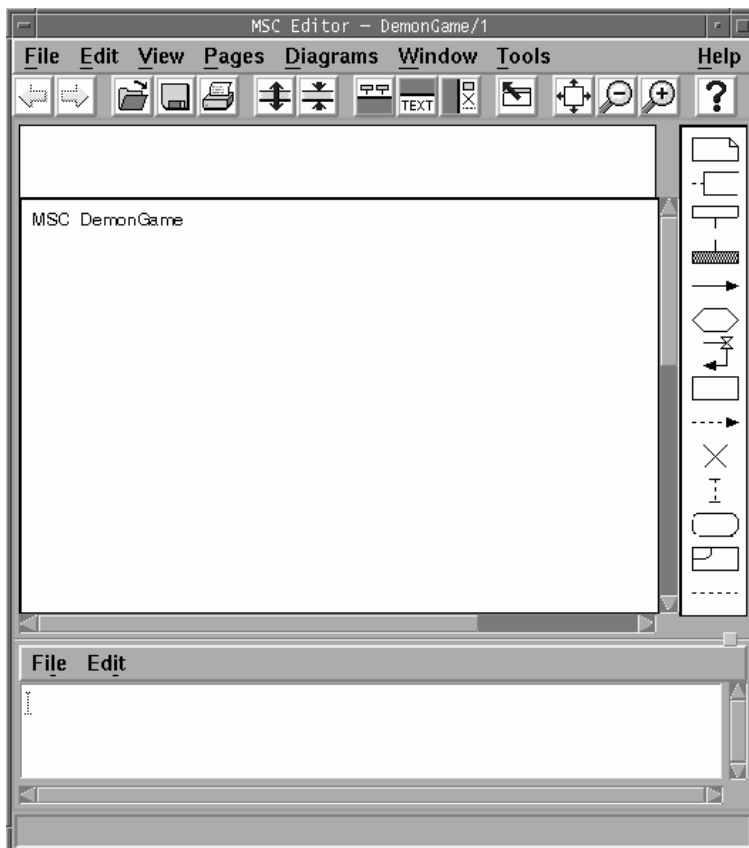


Figure 87: The MSC Editor window (on UNIX)

Managing Message Sequence Charts

Your next task is to use the MSC Editor to create the following diagram:

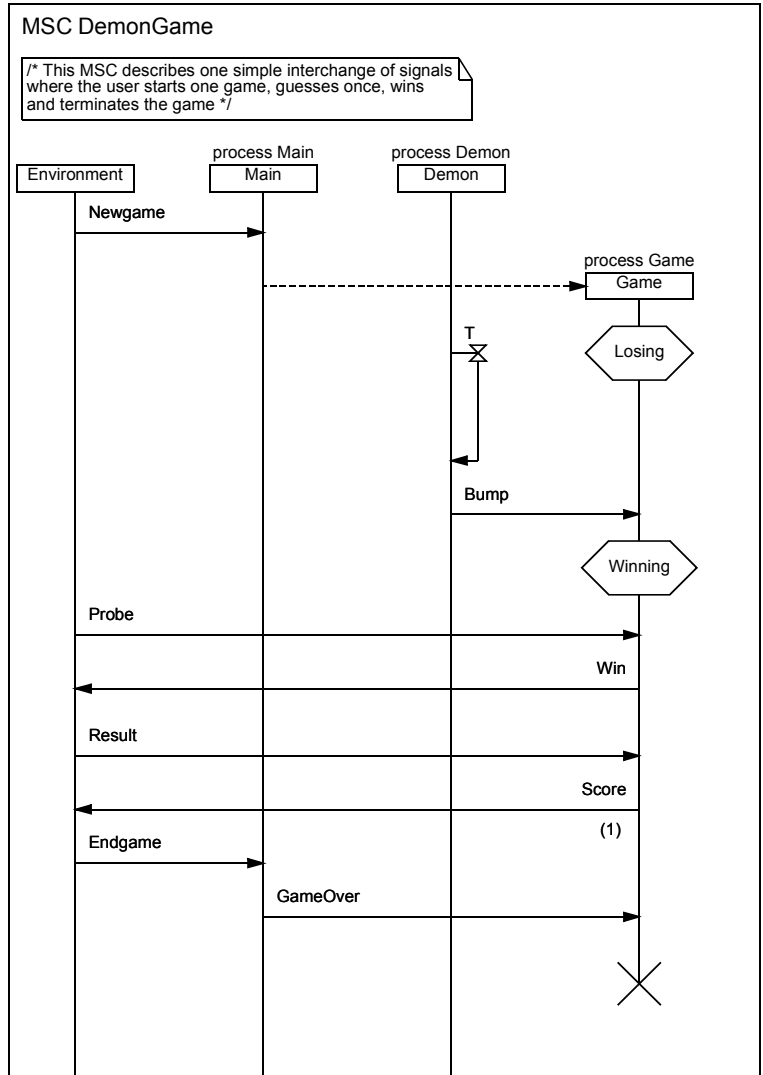


Figure 88: The MSC for the system DemonGame

The MSC basically consists of four instances (the vertical lines starting with a rectangle), a number of messages (the horizontal lines ending with an arrow), a create process (the dashed horizontal line), a timer (the symbol starting with an hourglass and ending with an arrow) and two condition symbols (with the shape of a hexagon). You also find a text symbol, containing a textual comment in it.

How to Draw the MSC

We suggest that you draw the MSC as described below. If you are unsure what symbol in the symbol menu to use, select or point to a symbol and look at the description in the Status Bar.

1. Start by entering the text symbol and fill in its contents. (This is done in the same way as with the SDL Editor).
2. Then, insert the three instances with the instance name **Environment**, **Main** and **Demon**:
 - To insert an instance, locate the instance head symbol in the symbol menu, select it and place it into the drawing area as shown in [Figure 88](#); as soon as you insert an instance head, the MSC Editor automatically appends an instance axis (with an infinite length).
 - Type in the text to assign the instance name (**Environment**, **Main**, **Demon**)
 - To assign the instance kind (**process Main**, **process Demon**), select the small rectangle located immediately above the instance head symbol and type in the text.

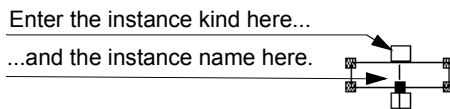


Figure 89: The text attributes associated to an instance head

- If you are not satisfied with the placement of an instance head, you may drag the symbol to a new location.

3. Once the three instances are added, insert the message Newgame:
 - Select the message symbol in the symbol menu
 - Move the pointer into the diagram. You will notice that a circle indicates a start position outside an instance axis.
 - Click once on the instance axis Environment to define the start of the message.
 - Move the pointer towards the instance axis Main. The message arrow follows the pointer, and a filled circle now indicates an end position outside an instance axis.
 - Click a second time on the instance axis Main to specify the end of the message.
 - Type in the name of the message (**Newgame**).
 - If you are not satisfied with the placement of a message, you may move it up or down by dragging the mouse. You may also move only the start or end position of the message along the instance axis.
4. The instance Game is dynamically created. To add Game, you use the create process symbol. You insert it in a similar fashion as a message:
 - Select the create process symbol in the symbol menu.
 - Click once on the instance axis Main to specify the source of the create process symbol.
 - Click a second time to specify the location of the instance head. A process create and an instance head with its axis are inserted.
 - Fill in the instance kind and instance name fields (after you have selected the instance head).
 - If desired, you may move the instance head symbol.

5. Continue by adding the first condition symbol to the instance axis Game:
 - Select the condition symbol in the symbol menu, and move the pointer to the instance axis. Click to insert the symbol and fill in the name of the condition: **Losing**.
 - The condition symbol may now be moved vertically along the instance axis.
6. Add a timer to the instance axis Demon:
 - Select the timer symbol in the symbol menu.
 - Click once on the instance axis to specify the base of the timer symbol.
 - Move the pointer downwards and click a second time on the same instance axis to locate the end of the timer (the end must reside below the source).
 - Enter the name of the timer: **T**
 - You may drag the start or endpoint to resize the timer symbol, if required. You may also drag the symbol to move it up or down.
7. Insert the message Bump.
8. Add the second condition symbol, **winning**, to the instance axis Game.
9. Add the remaining messages. The message Score also contains a parameter with the value 1. To enter the parameter value, select the lower of the two selection rectangles and type in the text **1**.



Figure 90: The text attributes associated to a message

10. Conclude the editing of the MSC by adding a process stop symbol.
 - Select the symbol in the symbol menu.
 - Place it by a click on the instance axis Game, below the last message.

11. Before leaving the MSC Editor, you should save the MSC. When saving the newly created diagram, the editor suggests the file name `DemonGame.msc`. Accept the suggested file name by clicking the *OK* button.

Using the Index Viewer

In this exercise, you will practice on the Index Viewer. The Index Viewer is a dedicated tool that presents a graphical view of the definitions and references to SDL entities available in an SDL system. It manages virtually all SDL information related to a system and has a number of facilities for navigating back to the source SDL diagrams.

A prerequisite to this exercise is an up-to-date cross reference file containing the definitions and references, which was generated when you last performed a semantic analysis of the system.

If you have changed any of the SDL diagrams since the last analysis of the system, you should regenerate the file. Perform a new semantic analysis and make sure the option *Generate a cross reference file* is on (see [Figure 82 on page 112](#)).

What You Will Learn

- To start the Index Viewer
- To look for definitions
- To look for references

Starting the Index Viewer

1. You start the tool with the *Index Viewer* command from the Organizer's *Tools* menu and its sub-menu *SDL*. Select the sub-menu *SDL* and the menu choice *Index Viewer*.
 - As an alternative, you may click the quick button for the *Index Viewer*. You will then be prompted to save unsaved diagrams. The SDL system is then analyzed and a new cross reference file is generated automatically.
2. The Index Viewer window is displayed. Start by opening the newly created file `DemonGame.xrf` (unless you used the quick button, in which case the file is automatically opened).
3. The Index Viewer reads the file, interprets the content and displays it in graphical form.



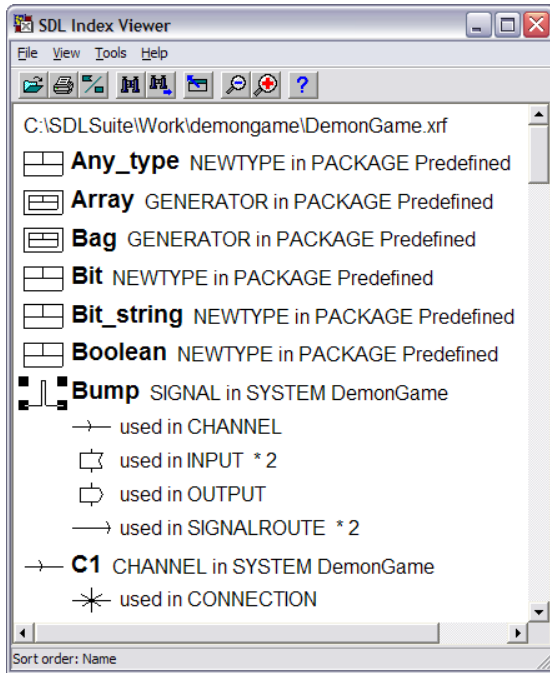


Figure 91: The Index Viewer window

- Some definitions are predefined in the SDL Suite environment (the ones containing *PACKAGE Predefined*). They are not of interest for the purpose of this tutorial.

In the next exercise, you will use the Index Viewer to identify all possible situations where a certain signal may be sent or received. We will also look for the definition of the signal.

Finding a Definition

Let us look for the definition of the signal Probe. By default, the definitions in the window are sorted alphabetically, but you do not need to scroll the window manually to find a definition.

There is a *Search* quick button that can be used to find **any** text in the window. However, if you want to search for the name of a definition, there is an even faster way.

1. Simply start keying in the name “Probe”. When you start typing, a search is started on the names that are displayed in bold face. As you type each letter on the keyboard, the status bar at the bottom of the window indicates what text is being searched for, and the window scrolls to show the first matched name. After a few keystrokes, the Index Viewer window shows the signal Probe selected:

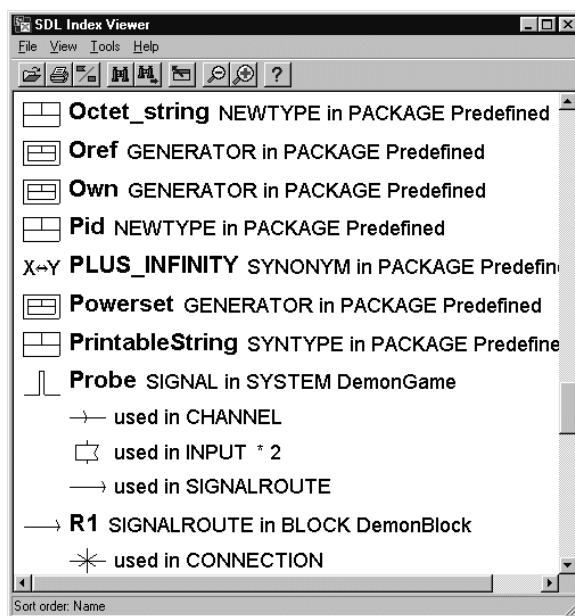


Figure 92: Finding the signal Probe

The selected row shows the icon for a signal, the name and type of the definition (“Probe SIGNAL”) and which diagram the signal is defined in (“SYSTEM DemonGame”).

We now wish to see where the signal is defined.

2. Make sure the Probe icon still is selected.

3. From the *Tools* menu, select the command *Show Definition*.
 - You can also double-click the icon.

An SDL Editor window is displayed on the diagram for the system DemonGame. The text symbol containing the declaration (i.e. the definition) of the signal is selected.

Finding References

Below the Probe icon in the Index Viewer, all uses (references) of the Probe signal are listed, including the icons for the SDL entities in which the signal has been referred. The information displayed in [Figure 92](#) should be interpreted as:

- The signal is conveyed on one channel,
- The signal may be input in two states,
- The signal is conveyed on one signal route.

To conclude this exercise, you will now locate the places where the signal is input.

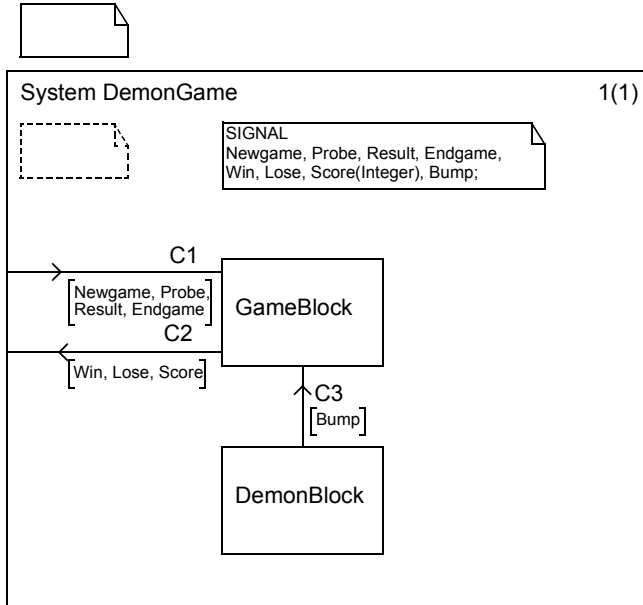
1. Select the input icon.
2. The *Tools* menu should now contain the two menu choices *Show Use 1* and *Show Use 2*.
3. Select the menu choice *Show Use 1*— an input symbol is selected in an SDL Editor window, showing the diagram for the process Game.
4. Select the menu choice *Show Use 2* – the second input symbol is selected, also in process Game. These are the two situations where the signal may be input.
5. Double-click the signal route icon. The signal route containing the Probe signal is selected in an SDL Editor window.
 - If you double-click an icon with more than one reference, the selection in the SDL Editor is automatically changed to the next occurrence. You may try this with the input icon.

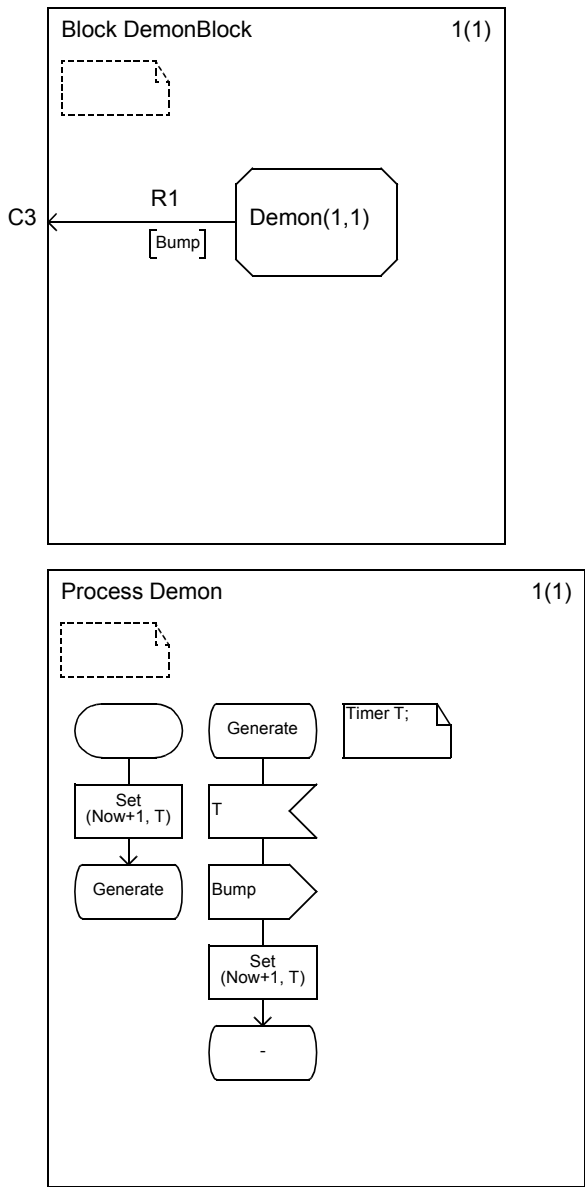
So Far...

You should now have learned how to use the basic functions in the SDL Suite – creating, managing, editing and printing SDL diagrams – and how to create Message Sequence Charts in the MSC Editor. You have also practiced on syntactic and semantic checks on your SDL diagrams. Finally, you have acquainted yourself with the Index Viewer.

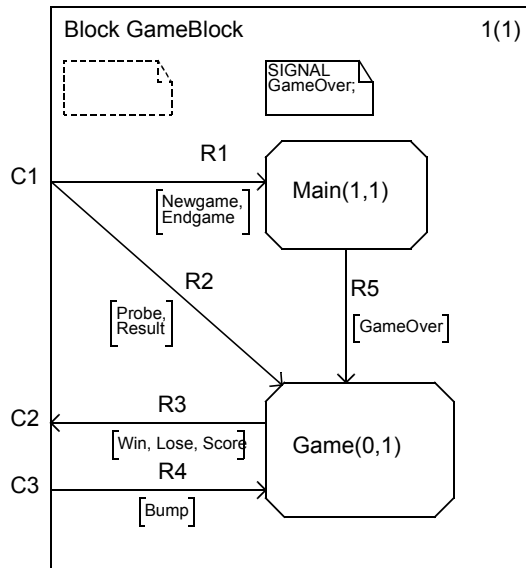
Your next task will be to “animate” your first SDL system by simulating it. A number of exercises are prepared in the next tutorial, starting with [“Purpose of This Tutorial” on page 136.](#)

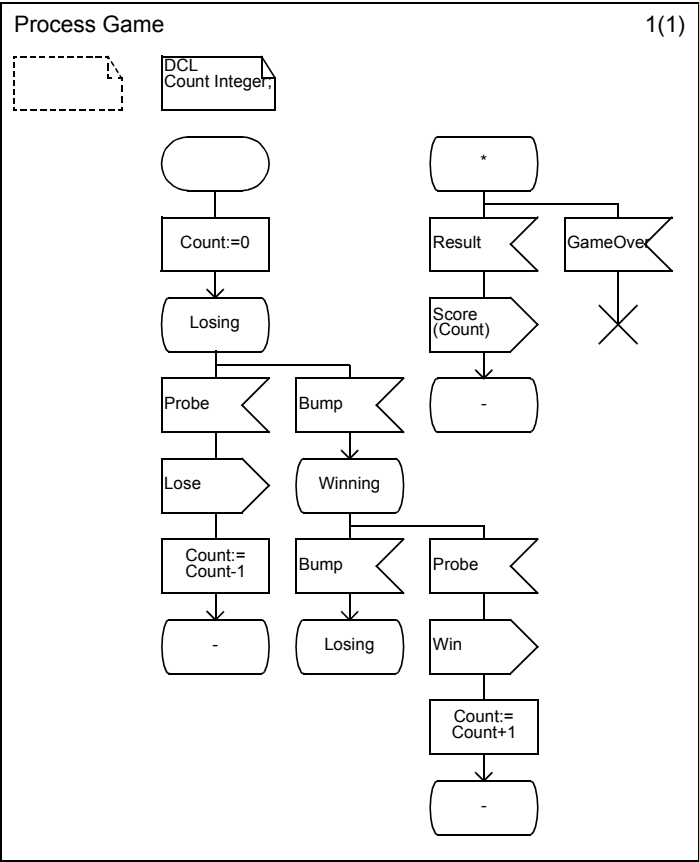
Appendix A: The Definition of the SDL-88 DemonGame

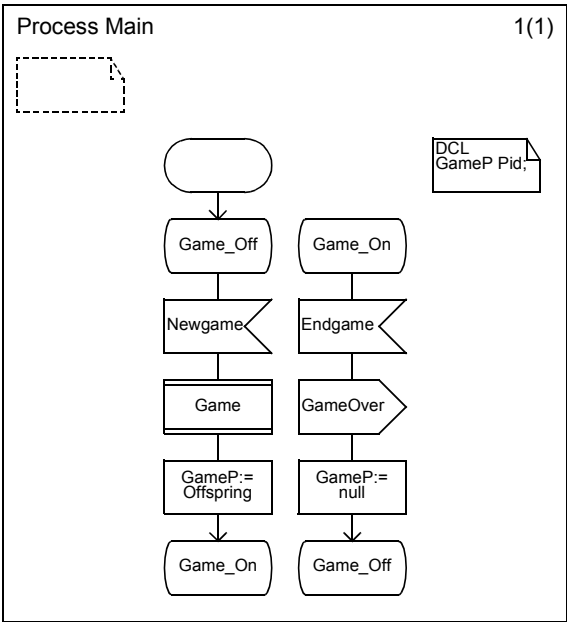




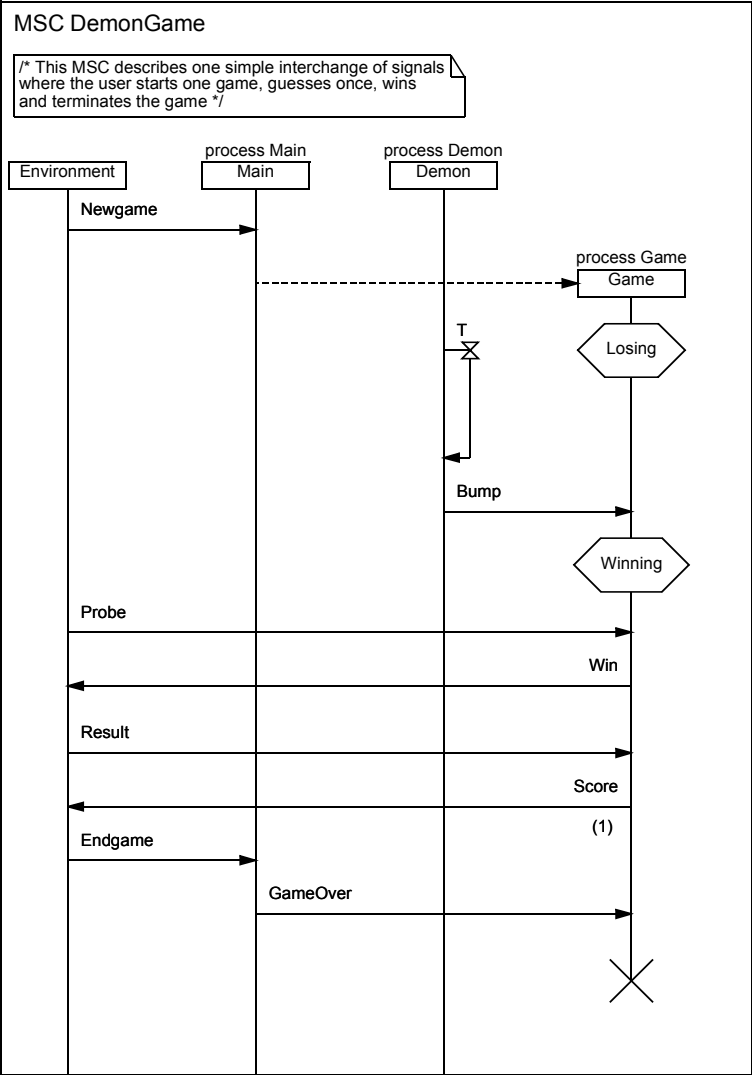
Appendix A: The Definition of the SDL-88 DemonGame







Appendix B: The MSC for the DemonGame



Tutorial: The SDL Simulator

The SDL Simulator is the tool that you use for testing the behavior of your SDL systems. In this tutorial, you will practice “hands-on” on the DemonGame system.

To be properly assimilated, this tutorial therefore assumes that you have gone through the exercises that are available in [chapter 3, *Tutorial: The Editors and the Analyzer*](#).

In order to learn how to use the Simulator, read through this entire chapter. As you read, you should perform the exercises on your computer system as they are described.

Purpose of This Tutorial

The purpose of this tutorial is to make you familiar with the essential simulation functionality in the SDL Suite. Typically, simulation means executing the system under user control; stepping, setting breakpoints, examining the system, processes and variables, sending signals and tracing the execution, as you would do with a debugger, but applied on the SDL domain.

This tutorial is designed as a guided tour through the SDL Suite, where a number of hands-on exercises should be performed on your computer as you read this chapter.

We have on purpose selected a simple example that should be easy to understand. It is assumed that you have a basic knowledge about SDL — this chapter is **not** a tutorial on SDL.

It is assumed that you have performed the exercises in [chapter 3, *Tutorial: The Editors and the Analyzer*](#) before starting with the tutorial on the simulator.

Note: C compiler

You must have a C compiler installed on your computer system in order to simulate an SDL system. Make sure you know which C compiler(s) you have access to before starting this tutorial.

Note: Platform differences

This tutorial, and the others that are possible to run on both the UNIX and Windows platform, are described in a way common to both platforms. In case there are differences between the platforms, this is indicated by texts like “on UNIX”, “Windows only”, etc. When such platform indicators are found, please pay attention only to the instructions for the platform you are running on.

Normally, screen shots will only be shown for one of the platforms, provided they contain the same information for both platforms. This means that **the layout and appearance of screen shots may differ** slightly from what you see when running the SDL Suite in your environment. Only if a screen shot differs in an important aspect between the platforms will two separate screen shots be shown.

Generating and Starting a Simulator

Once you have designed and analyzed a complete SDL system, it is possible to simulate the system, i.e. to interactively inspect and check its actual behavior. To be able to simulate the DemonGame system, you must first generate an executable simulator and then start the simulator with a suitable user interface.

Note:

In order to generate a simulator that behaves as stated in the exercises, you should use the SDL diagrams that are included in the distribution instead of your own diagrams. To do this:

- **On UNIX:** Copy all files from the directory
`$telelogic/sdt/examples/demongame`
to your work directory `~/demongame`.
- **In Windows:** Copy all files from the directory
`C:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\examples\demongame`
to your work directory
`C:\IBM\Rational\SDL_TTCN_Suite6.3\work\demongame`.

If you generate a simulator from the diagrams that you have created yourself, the scheduling of processes (i.e. the execution order) may differ.

If you choose to copy the distribution diagrams, you must then reopen the system file `demongame.sdt` from the Organizer.

What You Will Learn

- To generate an executable simulator
- To start the simulator user interface
- To start a simulator from the user interface

Generating the Simulator

To generate an executable simulation program, do as follows:

1. Make sure the system diagram icon is selected in the Organizer.
2. Select the *Make* command from the *Generate* menu. The Make dialog is opened:

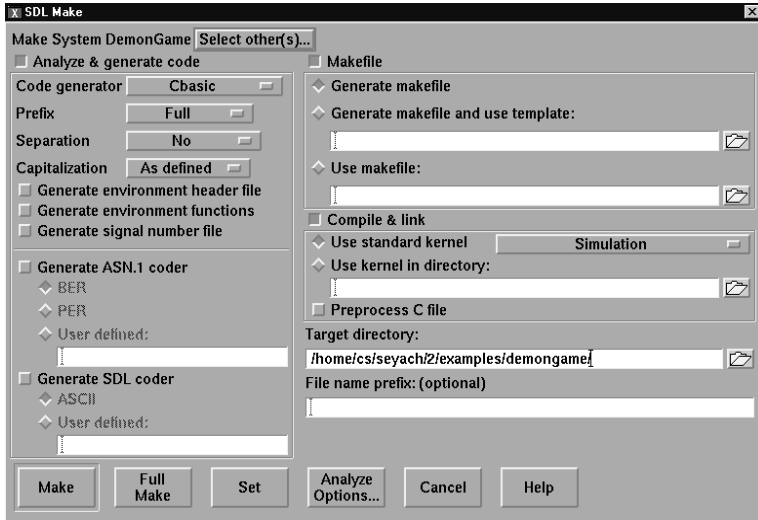


Figure 93: The Make dialog (on UNIX)

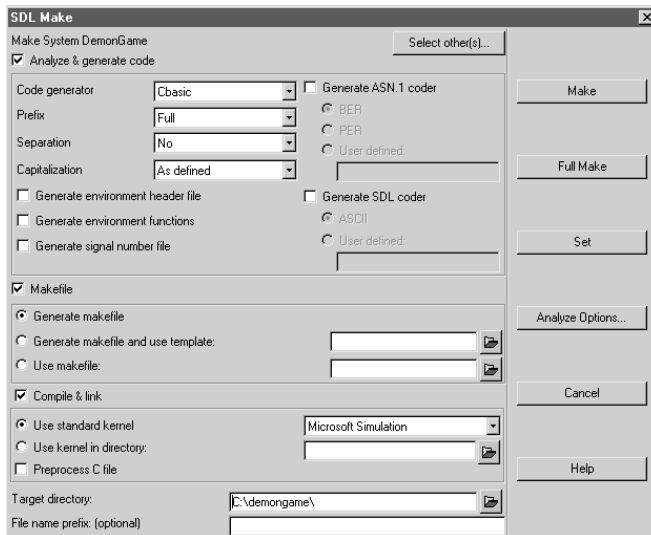


Figure 94: The Make dialog (in Windows)

3. Adjust the options in accordance to the figure, i.e.
 - *Analyze & generate code* on
 - *Makefile* and *Generate makefile* on
 - *Compile & link* on
 - *Use standard kernel* on. Make sure that a *Simulation* kernel is specified to the right; if not, select it from the option menu.
4. Click the *Make* button.
5. Select the *Organizer log* from the *Tools* menu. Check that no errors occurred. The Organizer's status bar should read "Analyzer done" and the Organizer Log should report no errors between the "Make started" and "Make completed" messages.
6. If errors were reported, bring up the Make dialog again, but click the *Full Make* button instead. This time, no errors should be reported.
 - Another problem could be with the C compiler used on your system. If you still receive errors, try changing to a Simulation kernel reflecting your C compiler, e.g. *gcc-Simulation* or *Microsoft Simulation*, and repeat the Make process.

Starting the Simulator

The generated simulator is now stored on a file called `demongame_XXX.sct` (**on UNIX**) or `demongame_XXX.exe` (**in Windows**) in the directory from which you started the SDL Suite (the `_XXX` suffix is platform or kernel/compiler specific). The simulator contains a *monitor system* that provides a set of commands which can be used to control and monitor the execution of the simulator.

It is possible for you to execute the simulator directly from an OS prompt, in which case you have to enter all commands to the monitor system textually using a simple command-line interface.

The SDL Suite provides a user-friendly graphical interface to the simulator that is started from the Organizer.

1. From the *Tools* menu, select the sub-menu *SDL* and the command *Simulator UI*. The Simulator UI window is opened:

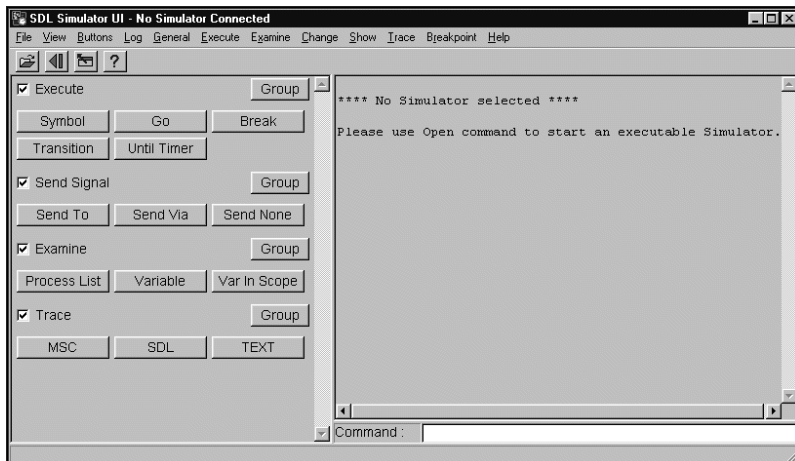


Figure 95: The main window of the Simulator UI (on Windows)

The text area to the right informs you that no simulator is running. The text area displays the textual input/output from the monitor system, such as entered commands, the results of commands, and error messages.



2. To start a simulator, select *Open* from the *File* menu, or click the *Open* quick button.
3. In the file selection dialog, the generated simulator file described above should be listed. Select it and click *OK* or *Open*.
4. The text area of the Simulator UI greets you with a welcome message to acknowledge that the simulator has been started:

Welcome to SDL SIMULATOR. Simulating system Demongame.

When a simulator is started, the static process instances in the system are created (in this case Main and Demon), but their initial transitions are not executed. The process in turn to be executed is the Main process.

The Simulator UI is now ready to accept commands to the monitor system. Whenever it is possible to enter a command, the prompt `Command:` is issued in the text area.

Executing Transition by Transition

What You Will Learn

- To enter commands textually
- To show the next symbol to be executed
- To interpret printed trace and graphical trace
- To execute the next transition
- To issue commands using buttons
- To send signals from the environment

Executing the Start Transitions

In this exercise, you will execute the start transitions in the process instances of the system. First, however, we must set the amount of trace information that we want printed during execution. The command **Set-Trace** is used for setting the textual trace level.

For now, you will enter commands textually by using the text field *Command*: just below the text area; this field is called the *input line*.

1. Click in the input line to place the cursor. Enter the command **set-trace 6** and hit <Return>. The value 6 specifies that we want full information about the actions that are performed during the transitions. The entered command is moved to the text area and the simulator monitor confirms the trace setting; the text area should now show:

```
Welcome to SDL SIMULATOR. Simulating system Demongame.
```

```
Command : set-trace 6  
Default trace set to 6
```

```
Command :
```

A graphical trace (*GR trace*) is also available, which means that the execution is traced in the SDL diagrams by selecting the next symbol to be executed. GR trace is by default off when you start the simulation.

2. To enable the GR trace, enter the command **set-gr-trace 1**. The value 1 specifies that the next symbol to be executed will be displayed in an SDL Editor each time the monitor system is entered.

3. To have an SDL Editor window appear now, enter the command **show-next-symbol**. An SDL Editor window appears, showing the diagram for the process Main with its start symbol selected (this is the next symbol in turn to be executed). This SDL Editor window will be used for graphically tracing the execution of the simulator during your simulation session.
4. If needed, move and resize the Simulator UI and SDL Editor windows so that they both are completely visible and fit on the screen together. A few useful advice are:



- Before resizing the SDL Editor, you may hide the symbol menu and the text window by using the quick buttons. The editor window will not be used for editing any diagrams, only viewing them.
- Check and, if required, adjust the SDL Editor's option *Always new window* to off. (Use the command *Editor Options* on the *View* menu for this).
- You may reduce the width of the Simulator UI window somewhat, which only affects the width of the text area. You may also reduce the height, but you should make sure that all buttons in the left part of the window are still visible.
- During this simulator tutorial, you will not need to see the contents of the Organizer window. You may cover it, or minimize it by using the window system.

To determine the transition in turn to be executed, you can look in the editor window. The start state of process Main is selected and the next state symbol is Game_Off. To execute this (empty) transition, you will use the command **Next-Transition**.

5. Execute the command Next-Transition by simply entering **n-t** in the input line. All commands may be abbreviated as long as the abbreviation is unique among all available commands.

The start transition of Main is now traced in two ways:

- In the text area, the textual trace information of the transition contains the process instance, the name of the initial state, and the current value of the simulation time. It ends with the Next-state action, giving the name of the resulting process state:

Executing Transition by Transition

```
*** TRANSITION START
*      PID      : Main:1
*      State    : start state
*      Now      : 0.0000
*** NEXTSTATE   Game_Off
```

- In the SDL Editor, the GR trace selects the next symbol to be executed. Since this is the start state of the process Demon, that diagram is loaded into the editor.

The next transition is the start transition of Demon, which contains the setting of a timer. To execute it, you can use another feature of the command interface:

6. Place and click the pointer on the input line and press the arrow key <Up>. The command you entered previously appears (n-t). Execute it by pressing <Return>.
- You can use the <Up> and <Down> arrow keys repeatedly on the input line to show previously entered commands. This feature is commonly known as a *command history*. You may also edit a command before it is executed, for instance by changing the value of a parameter.

The start transition of Demon executes. Also, the SDL Editor sets the selection on the text symbol where the declaration of the timer T is found. This is a convention adopted in the SDL Suite to show that the next event to take place in the system, if no signal is sent from the environment, is the expire of a timer.

Note that the printed trace also contains the action of setting the timer T.

Sending Signals from the Environment

To make something of interest happen in the system you have to send signals from the environment into the system. We will start by sending the signal Newgame to the Main process. For this, you can use the command **Output-Via**, which takes as parameters a signal name, the parameters of the signal (none in this case), and a channel name.

In this exercise, however, you will execute commands using buttons instead of entering them textually on the input line. The command buttons are arranged into different “modules” in the left part of the Simulator UI. You can “preview” the command that is executed by a button by se-

lecting it and move the mouse pointer away from the button before you release the mouse. The associated command is then listed in the Status Bar at the bottom of the window.

1. Locate the button module *Send Signal* and click on the *Send Via* button.

This button executes the Output-Via command, as is shown in the text area. A dialog is opened, asking for the value of the first parameter, the signal name. The list contains all signals possible to send from the environment:

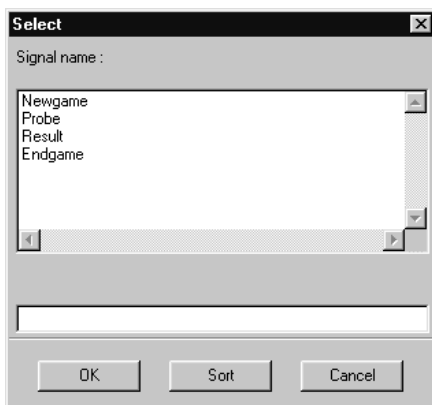


Figure 96: Sending the signal Newgame

2. Select the signal *Newgame* and click the *OK* button.
3. Another dialog is opened, asking for the channel name.

Executing Transition by Transition

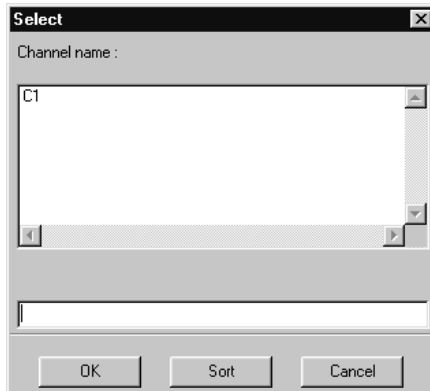


Figure 97: Selecting the channel to send via

4. As there is only one channel from the environment to the system, you do not have to select it explicitly. Simply click the *OK* button.

The signal is now sent, which is confirmed in the text area. The GR trace shows that the next symbol to execute is the input of Newgame (in process Main).

5. Execute the next transition by using the button *Transition* in the module *Execute* (this executes the Next-Transition command). The printed trace information shows the actions of the executed transition up until the state *Game_On*. Note that the start of the transition is described by the combination of a state (*Game_Off*) and the input of a signal (*Newgame*):

```
*** TRANSITION START
*      Pid      : Main:1
*      State    : Game_Off
*      Input     : Newgame
*      Sender    : env:1
*      Now       : 0.0000
*      CREATE Game:1
*      ASSIGN GameP := Game:1
*** NEXTSTATE Game_On
```

Since the process *Game* was created in the transition, the GR trace shows that the next symbol to execute is the start state of *Game*.

This clearly demonstrates the difference between printed trace and GR trace:

- The printed trace describes what happened in the previously executed transition, including the initial state and the reached state of the process.
 - The GR trace shows what will happen next, if the system is left on its own. The start symbol of the next transition is selected, which may be in a different process diagram.
6. Execute the start transition of Game with the *Transition* button. The Game process reaches the state Losing, and the GR trace changes back to the Demon process. The SDL Editor selection shows again the text symbol with the declaration of the timer T.
 7. Execute the next transition, in order to have the timer expire. This transition is a timer output, i.e., a timer that sends its signal to the process which earlier executed the Set action. A timer output is also considered to be a transition. Note that the simulation time has now been updated to 1:

```
*** TIMER signal was sent
*   Timer      : T
*   Receiver   : Demon:1
*** Now       : 1.0000
```

8. Execute another transition. The start of this transition is described by the combination of the Generate state and the input of the timer T. The signal Bump is now sent to the Game process:

```
*** TRANSITION START
*   Pid       : Demon:1
*   State     : Generate
*   Input     : T
*   Sender    : Demon:1
*   Now       : 1.0000
*   OUTPUT of Bump to Game:1
*   SET on timer T at 2.0000
*** NEXTSTATE  Generate
```

The GR trace shows the next symbol to be the input of Bump in Game. In the process diagram, note that after the input of Bump, the Game process will be in the state Winning awaiting either the input of another Bump or the input of a Probe signal.

Executing Transition by Transition

9. Execute the next transition to put the process Game in the state Winning. The GR trace switches back to the Demon process and indicates the next default behavior which is the expire of the timer T. However, you will instead send the signal Probe from the environment:
10. Send the Probe signal by using the *Send Via* button as before. The GR trace switches back to the Game process.
11. Execute the next transition. The Probe signal is consumed and the signal Win is output to the environment. The process returns to the state Winning and awaits a new Bump (or Probe) signal.

```
*** TRANSITION START
*      PId      : Game:1
*      State    : Winning
*      Input     : Probe
*      Sender    : env:1
*      Now      : 1.0000
*      OUTPUT of Win to env:1
*      ASSIGN   Count := 1
*** NEXTSTATE   Winning
```

We have now shown how you can use the commands Next-Transition and Output-Via to reach a certain point or state in the simulation.

Viewing the Internal Status

In this exercise we will introduce some of the available commands for viewing the internal status of the system. With the graphical user interface, it is also possible to continuously view the internal status without having to execute commands manually.

In the previous exercise, you learned how to interpret the available traces. We will not focus on these details anymore, unless we need to point out some important aspect.

What You Will Learn

- To restart the simulator without leaving the Simulator UI
- To use the Command and Watch windows
- To view and interpret the process ready queue
- To view the signal input port of a process
- To view variable values
- To examine process instances
- To view active timers

Restarting the Simulator

Before continuing, you need to restart the simulation from the beginning and set the trace level:

1. Select the *Restart* command from the *File* menu.
2. A dialog informs you that the current simulation will terminate. Confirm this by clicking *OK*. The text area is cleared and the simulator is now reset.
3. Set the trace level to 6 and the GR trace level to 1, as before. The easiest way to do this is to use the up arrow key on the input line to find the previous Set-Trace 6 and Set-GR-Trace 1 commands and then hit <Return>.
 - The Simulator UI remembers all previous commands entered on the input line in the same session, even if you have started a new simulator.

Viewing Process and Signal Queues

To view the internal status continuously, the *Command window* is used. This window shows, if your preferences are set up adequately, the process ready queue and a list of all processes.

This information is displayed in separate “modules” in the Command window by executing suitable monitor commands (**List-Ready-Queue** and **List-Process**). These modules are similar to the button modules in the main window.

1. Select *Command Window* from the *View* menu to open the Command window. Resize the window so that both command modules become visible (see [Figure 98](#)). Move the window so that you can still see the SDL Editor window and use the main window.

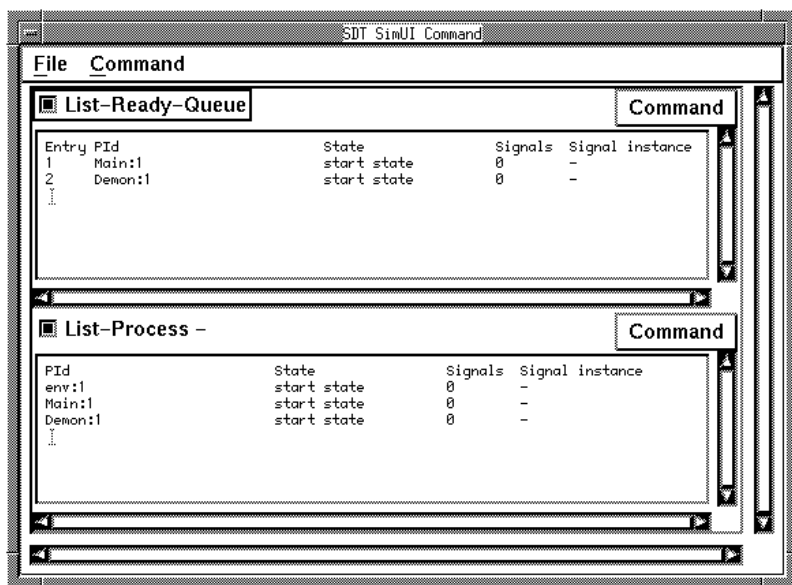


Figure 98: The Command window

Note:

Whether the Command window shows any commands or not is preference dependent. If the window does not show these command modules at start-up, you may add the commands using the *Add Command* menu choice from the *Command* menu and specify each of the commands to add. See [Figure 99](#).

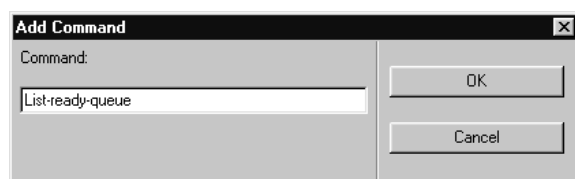


Figure 99: Specifying a command to add

The command modules show the following information:

- The List-Ready-Queue command displays the process ready queue, i.e., an ordered list of all processes that are ready to execute a transition. The list contains an entry number (the position in the queue), the process instance, its current state, the number of signals in its input port, and the name of the signal that will cause the next transition. At this stage, both processes (Main and Demon) are to execute their start transition, so no signals are listed.
 - The List-Process command displays a list of all active processes in the system. It shows the same information as the List-Ready-Queue command above (with the exception of the additional “env” process, which represents the environment). At this stage, the two lists are identical.
2. Execute the next transition and note the changes in the Command window. The Main process is removed from the ready queue since it needs a signal input (Newgame) to execute the next transition, but this signal has not yet been sent. The process list shows the new state of Main (Game_Off).
 3. Execute the next transition. The ready queue is now empty since the Demon process needs an input of the timer signal T, but this timer has not yet expired.

Viewing the Internal Status

4. Send the signal `Newgame` from the environment. The Command window shows that `Newgame` has entered the signal input port of `Main`, thus adding `Main` to the ready queue.
5. You may also print a list of all signals in the input port of the process in turn to execute. The command for this is not available through a command button. Instead, locate the menu *Examine* in the menu bar and select the *Input Port* command. For each signal an entry number (the position in the signal queue), the signal name, and the sender of the signal is printed. The asterisk before the entry number of `Newgame` indicates that this signal will be consumed by the process in the next transition.

Viewing Variables and Process Instances

Apart from the Command window, you can also continuously monitor variable values by using the *Watch window*. We will now monitor the variable `GameP` in process `Main` to see how its value changes as the process `Game` is started and later stopped.

1. Open the Watch window by selecting *Watch Window* from the *View* menu. If needed, move it so that you can also see the contents of the Command window.
2. In the Watch window, select *Add* from the *Watch* menu to add a variable to the list of variables to display.
3. In the dialog, you have to specify both the process (within parenthesis) and the variable name. Enter `(Main)GameP` and click *OK*.

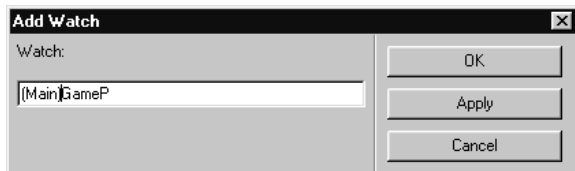


Figure 100: Adding a variable to watch

4. The value `null` should now be displayed in the Watch window:

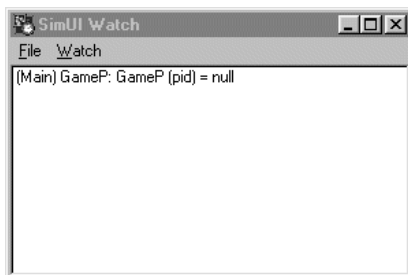


Figure 101: Adding GameP to the Watch window

If necessary, resize the window so that the value becomes visible.

5. Execute the next transition and check that the value of GameP in the Watch window changes to the value Game:1 as the process Game is created. Game is also added to the lists printed in the Command window.
6. You can examine the newly created Game process. The information is printed by giving the command **Examine-PId** (enter **ex-pid** on the input line) and contains the current values of Parent, Offspring, and Sender. Parent is `Main:1`, as expected.
7. Send the signal Endgame from the environment. Notice that Main is added to the ready queue, but **after** Game.
8. Execute the next transition, which is the start transition of Game.
9. You can now examine the Main process, since it is the next to execute. Give the command Examine-PId again and compare the values printed with those from the Game process.
10. Execute the two next transitions to stop the Game process. Notice that the value of GameP is reset to null and that Game no more is listed in the Command window.

Other Viewing Options

There are a number of other viewing commands available in the *Examine* menu. You can list the active timers in the system, check the parameters of signal and timer instances, etc. We will conclude this exercise by showing that the system is not idle, even though the ready queue is now empty.

1. Check that the timer T is still active by choosing *Timer List* in the *Examine* menu. The timer's name, corresponding process instance, and expiration time is printed.
2. Execute the next transition. Try to examine the timer instance by giving the command **Examine-Timer-Instance** (enter `ex-tim-ins` on the input line). You are informed that the timer queue is empty, i.e. the timer T is no longer active.

Dynamic Errors

What You Will Learn

- To recognize and interpret a dynamic error
- To find the SDL symbol last executed
- To continue an interrupted transition

Finding a Dynamic Error

In this exercise, a dynamic error in the Demongame system will be detected. The error is found by simply executing the first four transitions of the system:

1. Select the *Restart* command from the *File* menu.
2. Set the trace level to 6.
3. We will not use graphical trace in this exercise. So, exit the currently open SDL Editor from its *File* menu.
4. As you will not need the Command and Watch windows, close them by selecting *Close All* from the *View* menu.

5. Execute the four next transitions until a warning message is printed in the text area:

```
***** WARNING *****
Warning in SDL Output of signal Bump
Signal sent to NULL, signal discarded
Sender: Demon:1
TRANSITION
  Process      : Demon:1
  State       : Generate
  Input       : T
  Symbol      :
#SDTREF(SDL,c:\IBM\Rational\SDL_TTCN_Suite6.3\work\...
TRACE BACK
  Process      : Demon
  Block       : DemonBlock
  System      : Demongame
*****
```

The message indicates that there was no receiver for the Bump signal sent from the Demon process. This is quite true, as no process instance of type Game has been created. The definition of the Demon game is thus not correct, as it actually requires that the user always has a game running, when Bump signals are sent. A better (and correct) solution would be to direct the Bump signals from Demon to Main, which then retransmits the signal to the instance of the Game process, if it exists.

6. When no GR trace is in effect, you can still see where the error occurred. Choose *Prev Symbol* in the *Show* menu. This opens an SDL Editor and selects the last symbol that was executed. In this case, the output of Bump in the Demon process.

After a dynamic error has occurred it is, of course, possible to continue the simulation, both to execute more transitions and to examine the status of the system. Note that the execution was stopped directly after the symbol in which the dynamic error occurred, i.e. the transition was interrupted.

7. To execute the interrupted transition to its end, issue a Next-Transition command as usual. In the printed trace you can see that no signal was sent in the erroneous output statement.

Using Different Trace Values

The amount of trace information printed during transitions is set by the command Set-Trace. So far, you have used this command to set the trace value to 6. The higher the trace value you set, the more information is printed.

You can also define trace values for different parts of the system. In this way, blocks, process types, process instances, etc. can have different trace values. If a process does not have a trace value defined, the value for the enclosing block is used. If the block does not have a defined value, the value for the next enclosing structure is used, etc. The system always have a trace value defined, which initially is 4.

In this exercise, you will use these facilities to run the demon game and only print trace information for transitions executed by the processes Main and Game. The process Demon will not be traced. This is accomplished by setting the trace value for the system to 0 and the value for the block GameBlock to 6.

The GR trace value will be set to 1 throughout this exercise.

What You Will Learn

- To set trace values for different parts of a system
- To list the current trace values
- To execute the next SDL symbol only
- To execute a sequence of transitions until trace is printed

Setting Trace Values

The command Set-Trace actually takes two parameters, the name of a unit and a trace value, and assigns the trace value to the unit. To easily specify the unit, you will now execute Set-Trace by using a menu, instead of entering it on the input line.

1. First, restart the simulator. If needed, resize and move the SDL Editor window that is open.
2. Locate the *Trace* menu and select the *Text Level : Set* entry. In the first dialog, select the unit *System Demongame* and click OK. In the second dialog, select the trace value 0. Note that all possible trace values (0-6) have a short explanation.

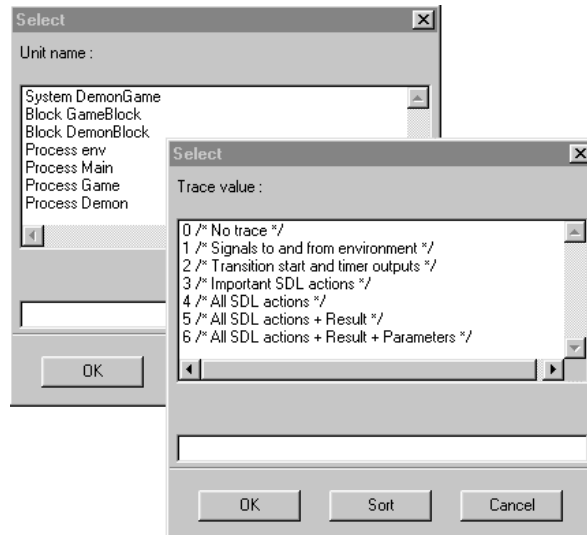


Figure 102: Setting the trace 0 for System DemonGame

3. In a similar way, set the trace value for the block GameBlock to 6.
4. The menu choice *SDL Level : Set* can be used to set the GR trace value in the same way. Use it to set the trace value for the system DemonGame to 1.
5. Check that you have set the correct trace values by using the menu choices *Text Level : Show* and then *SDL Level : Show*. The following information should be printed:

```
Default      4 = All SDL actions
System  Demongame : 0 = No trace
Block   GameBlock : 6 = All SDL actions + Result +
Parameters
```

```
Default      0 = GR trace off
System  Demongame 1 = Show next symbol when entering
monitor
```

Executing Symbol by Symbol

To clearly see that the Demon process is not traced in the text area, we will follow the execution in smaller steps than complete transitions. The smallest execution step possible is one SDL symbol at a time. The command **Step-Symbol** is used for this.

1. First, send the Newgame signal from the environment.
2. Execute the start symbol of Main by clicking the button *Symbol* (in the *Execute* module). Note that the printed trace does not include information about the next state (*Game_Off*) since that symbol has not yet been executed (it is selected to be executed next):

```
*** TRANSITION START
*      PID      : Main:1
*      State    : start state
*      Now      : 0.0000
```

3. Execute the next symbol with the *Symbol* button. Now, the printed trace gives information about the *Game_Off* state being reached:

```
*** NEXTSTATE   Game_Off
```

4. The execution now continues in the Demon process. Execute the three symbols in the start transition of Demon. Note that no trace is printed in the text area, since Demon is not part of the block Game-Block.
5. Continue executing the symbols in the Main and Game processes until the Demon process is entered again (you will need to press the *Symbol* button a number of times; watch the SDL Editor window for monitoring the execution). Note that trace is printed for each symbol.
6. When the Demon process is entered, you can continue to execute symbol by symbol, or you may execute the complete transition by using the *Transition* button as usual. No trace is printed.
7. Stop executing when you are back in the Game process.

Hiding Uninteresting Transitions

If you would continue to execute transition by transition at this point, trace would only be printed while executing the Game process. But, you would still have to manually execute the “silent” transitions in the Demon process. To avoid this, you can use another command,

Next-Visible-Transition. This command executes a sequence of transitions; it stops after it has reached a process with a trace value greater than 0, i.e., when the first “visible” transition is executed. In this way, transitions by uninteresting parts of the system are hidden.

1. Execute the command by choosing *Until Trace* in the *Execute* menu. The execution does not stop until the Game process is entered again and the state Winning (or Losing) is reached. Trace is then printed for the last executed transition.
2. Repeat the command a number of times. The printed trace shows that you are now switching between the states Losing and Winning in the Game process. The execution in the Demon process is hidden in the printed trace.

You should note, however, that the GR trace only shows the Demon process. Remember that the GR trace selects the next symbol to be executed, which is always in the Demon process when the Game process has reached the state Winning or Losing. If you want to check where in the Game process you are, do as follows:

3. Choose *Prev Symbol* in the *Show* menu to select the last executed symbol. This should be a state symbol, Winning or Losing, in the Game process.

Looking at the External Behavior

What You Will Learn

- To use the signal log facility
- To add command buttons to the Simulator UI
- To execute transitions up to a certain time value

Setting Trace and Signal Logging

During this exercise, you will look at the external behavior of the system, which is the same as actually playing the Demon game. To achieve this, we will set the system trace to 1. This means that you will see only signals sent to the environment and none of the actions performed during transitions. In order to log the external behavior on a file, you will also use the *signal log* facility.

1. As usual, restart the simulator.
2. Set the trace value for the system to 1.
3. To log the signals sent to and from the environment, enter the command **signal-log** in the input line and hit <Return>. (This command has no associated button or menu choice.)

The **Signal-Log** command takes two parameters, which are now asked for in dialogs. The first parameter is a unit name. All signals sent to, from or through the specified unit will be logged to file.

4. Instead of selecting one of the units in the list, enter the unit name **env** in the dialog's text field and click **OK**. This is the way to specify the environment of the system.

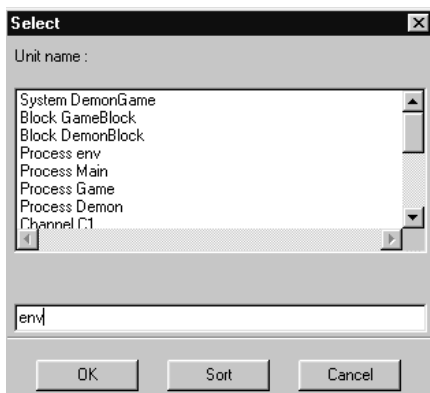


Figure 103: Specifying the environment

The second parameter is the name of a file name to which information about the signals will be written. A file selection dialog is opened.

5. In the *File* field, enter the file name `signal.log` and click *OK*.

Adding Buttons for Common Commands

When you are playing the Demon game, you are sending signals to the system from the environment. You will start by sending the signal New-game. Since this is an action often performed in the simulation of this system, we will first define a new button that executes the proper command. In this way, you only need to click the button to send the signal.

1. In the *Send Signal* module, select *Add* from the *Group* menu to the far right:

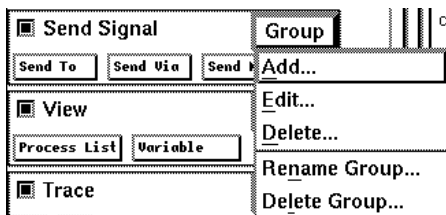


Figure 104: Adding a new button to a module

2. In the dialog, enter **Newgame** as the button label, but **do not** hit <Return>. Enter **output-via newgame -** as the command definition.

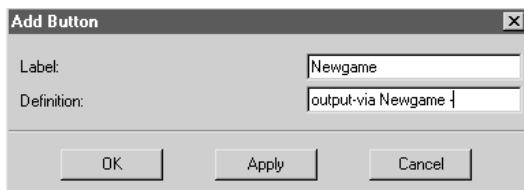


Figure 105: Adding a button

- The '-' as the last parameter to Output-Via indicates the default value, in this case for the channel parameter. Specifying '-' as a parameter is the same as just clicking *OK* in the corresponding parameter dialog.
3. Click *Apply*. The new button appears in the module, and the dialog is ready for another button definition.
 4. Since sending the Probe signal also is a common action, add a button *Probe* in the same way as above.
 5. If you wish, add buttons for the signals Result and Endgame in the same way. Finally, close the dialog with the *OK* button.
(If the *Error message* "Button label must not be empty" occur, just ignore the message.)

Playing the Game

You are now ready to start playing the game. You will use the new buttons to send signals to the game, and the command **Proceed-Until** to execute transitions up to the next point in time when you want to send a signal.

1. Send the signal Newgame with a click on the *Newgame* button.
2. Execute transitions until the time is 5.5 by selecting *Until Time* (in the *Execute* menu). Enter the value **5.5** in the dialog. This executes the command **Proceed-Until 5.5**. This will execute all transitions up to the point in time when the simulation time first becomes equal to the specified time value.

3. Send a Probe signal.
4. Execute transitions until the time is 10.3. Note the output of the signal Win or Lose to the environment.
5. Send a Probe signal again. Then, send another Probe signal. The two signals will enter the input port of the Game process. Check this by selecting *Input Port* in the *Examine* menu:

```
Input port of Game:1
Entry   Signal name      Sender
*1      Probe            env:1
2       Probe            env:1
```

6. Send the signal Result. Use the button if you have defined one; otherwise, use the *Send Via* button or enter the command on the input line.
7. Execute transitions until the time is 13.5. Note the output of the signals Lose or Win (one for each Probe) and Score to the environment.

Examining the Signal Log File

1. Exit the simulation by choosing *Stop Sim* in the *Execute* menu. This is needed to finish the signal logging. The Simulator UI itself is not closed by this command.
2. Examine the file `signal.log` from outside the simulator. The file contains a specification of all signals sent to and from the environment. It should look like this:

```
Signal log for system Demongame with unit Process
env on file ...
0.0000 Newgame from env:1 to Main:1
5.5000 Probe from env:1 to Game:1
5.5000 Win from Game:1 to env:1
10.3000 Probe from env:1 to Game:1
10.3000 Probe from env:1 to Game:1
10.3000 Result from env:1 to Game:1
10.3000 Lose from Game:1 to env:1
10.3000 Lose from Game:1 to env:1
10.3000 Score from Game:1 to env:1
Parameter(s) : -1
```

Using Breakpoints

The facility of a simulator demonstrated in this exercise is the *breakpoint*. A breakpoint can be used to stop the execution and activate the monitor system at a certain point of interest. There are four kinds of breakpoints; symbol, transition, output and variable. The first two kinds will be explained in the following.

What You Will Learn

- To set a breakpoint on an SDL symbol
- To set a breakpoint on a transition
- To list all defined breakpoints
- To graphically trace each executed SDL symbol
- To get a textual SDT reference to an SDL symbol
- To continuously execute the system

Setting Up the System

To see where a breakpoint is reached, you will start the execution of the system with the Go command. This command continuously executes transitions until an error occurs, a breakpoint is reached, or the system is completely idle. You should first set up the system in a way suitable for continuous execution:

1. Restart the simulator.
2. Set the trace value for the system to 0 to avoid trace information being printed during execution (you may use the *Text level : Set* entry from the *Trace* menu to do this) .
3. Set the GR trace value for the system to 2. Each SDL symbol will then be selected in the SDL Editor as it is executed, allowing you to follow the execution even though no trace is printed.

Setting a Symbol Breakpoint

A *symbol breakpoint* is set at a specific SDL symbol in the process diagrams. Symbol breakpoints are checked **before** symbols are executed, i.e. the symbol is not executed when the breakpoint is reached. We will now show how to set a breakpoint on the first task symbol in the Game process, i.e. the initializing of the variable Count to 0.

1. First, send the signal Newgame (using the new button). This is very important, as otherwise the Game process will not be created and the breakpoint will never be reached!
2. In the SDL Editor, bring up the Game process diagram from the *Diagrams* menu.
 - If, for some reason, this diagram is not included in the menu, you have to open it explicitly. Either click the *Open* quick button and select the file `Game.spr`, or double-click the Game diagram icon in the Organizer window.
3. In the Simulator UI, choose *Connect sdle* in the *Breakpoint* menu (you may have to resize the window to see the menu). This establishes a connection between the Simulator and the SDL Editor. As a consequence, a new menu *Breakpoints* appears in the SDL Editor's menu bar (you may have to resize the window to see the menu).
4. Go back to the SDL diagram and select the task symbol "Count:=0". Then, select the **second** command *Set Breakpoint* from the new *Breakpoints* menu in the editor (the one without trailing dots). The symbol breakpoint is now defined.

A red "stop" sign is added to the task symbol to indicate the breakpoint. Back in the Simulator UI, the definition of the symbol breakpoint is printed.

5. Start executing the system by pressing the *Go* button in the *Execute* module. Note how the SDL symbols are selected in rapid succession as they are executed. Finally, the breakpoint is reached and the execution stops. The symbol where the breakpoint was set is next to be executed.

Setting a Transition Breakpoint

A *transition breakpoint* is set at a specific transition in the system. Transition breakpoints are checked **before** transitions are executed, i.e. the transition is not executed when the breakpoint is reached. We will set a breakpoint in the Demon process, when it is in the state Generate and receives the timer T.

1. To define the breakpoint, choose *Transition* in the *Breakpoint* menu of the Simulator UI. This command takes a number of parameters. In the dialogs that appear:
2. Select the Demon process and click *OK*:

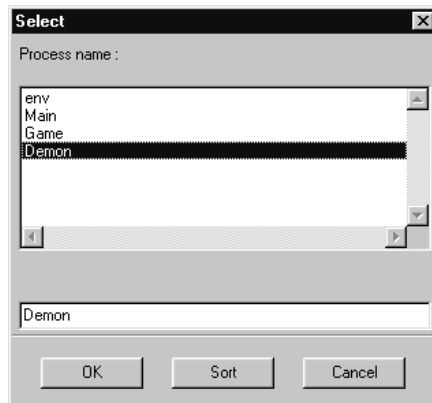


Figure 106: Specifying the process

3. Leave the instance number empty and click *OK*:

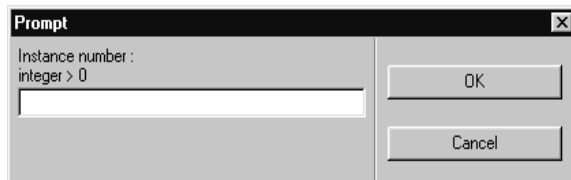


Figure 107: Leave the instance number empty

4. Do not specify a service name; simply click *OK*:

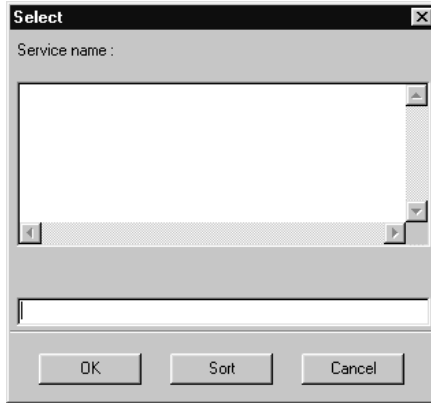


Figure 108: Do not specify a service name

5. Select the Generate state and click *OK*:

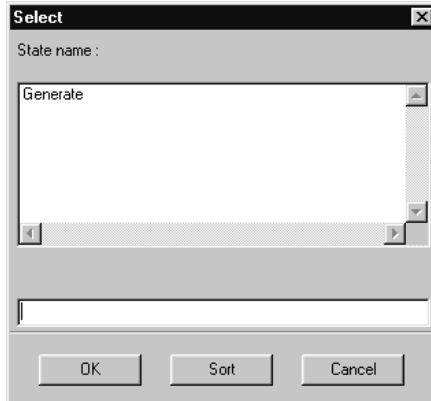


Figure 109: Specifying the state Generate

6. Select the timer T and click *OK*:

Using Breakpoints

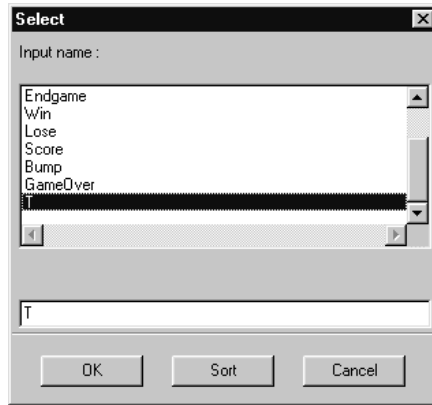


Figure 110: Specifying the timer T

7. Simply click **OK** in the remaining dialogs:

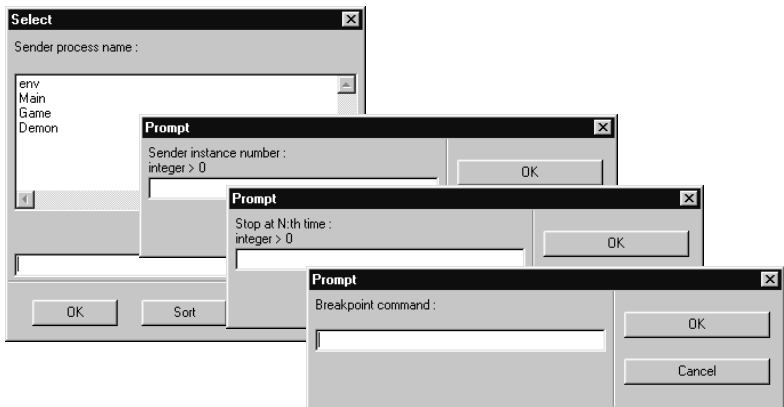


Figure 111: Leaving the remaining dialogs empty

To omit selecting a parameter value is interpreted so that any value, name or number, will match this parameter. In this case, any instance of Demon and any sender will match the breakpoint.

8. To see how the new breakpoint was defined, list all breakpoints with the *Show* entry in the *Breakpoint* menu:

```
1
Process name      : Demon
Instance         : any
Service name     : any
State            : Generate
Input            : T
Sender name      : any
Sender instance  : any
Stop each time
```

9. Resume execution of transitions by clicking the *Go* button. When the breakpoint is reached, you can see that the current state of the system matches the breakpoint definition:

```
Breakpoint matched by transition
Pid      : Demon:1
State    : Generate
Input    : T
Sender   : Demon:1
Now      : 1.0000
```

Changing the System

There are a number of commands in the simulator monitor that change the behavior of the system. These commands should be used with care, since it is no longer the original system that is simulated after such a command has been issued. These commands are still useful, especially in debugging situations, for making minor changes so that it is possible to continue the simulation after an error has been detected. They can also be used to force the system into certain situations, that otherwise would require a large number of transitions to be attained.

What You Will Learn

- To change and add commands in the Command window
- To create a process manually
- To change the process scope
- To change the value of a process variable
- To change the state of a process
- To set and reset a timer manually

Some Preparations

In the following two exercises, you will be changing processes and timers. Before continuing, you will set up the simulation and the Simulator UI in a suitable way.

1. Restart the simulator.
2. Open the Command window through the *View* menu. You will now change the commands executed in the Command window. The List-Process command is to be replaced by **Examine-Pid**, and a new command, **List-Timer**, will be added.
3. In the List-Process command module, select *Edit* from the *Command* menu to the far right. (This menu works in the same way as the *Group* menu in the button modules.)
4. In the dialog, change the command to `examine-pid` and click *OK*.
5. Go to the Command window's menu bar and select *Add Command* from the *Command* menu. In the dialog, enter the command `list-timer` and click *OK*.
6. A new command module is added to the window. Resize the window so that all three modules are visible.
 - You may also reduce the number of text lines displayed in a module by selecting *Size* from the module's *Command* menu. In a dialog, you can set the number of lines with a slider.

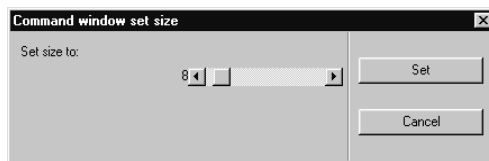


Figure 112: Adjusting the number of lines

7. Set the system trace value to 6 to get full trace.
8. Execute the two first transitions so the processes Main and Demon are started.
9. If the GR trace in the SDL Editor window makes it difficult to see the output in the Command window, set the system GR trace to 0.

Creating a Process

In this exercise, we will put the system in the state it would be in after the reception of a Newgame signal. This will be accomplished without actually sending the signal. Instead, we will manually create an instance of process type Game, using the Create command.

1. Create the Game process by selecting the *Create Process* entry (in the *Change* menu). Select the process Game, and click *OK*.
2. In the next dialog, select the parent process Main and click *OK*. This sets up the Parent-Offspring link between the process instances.

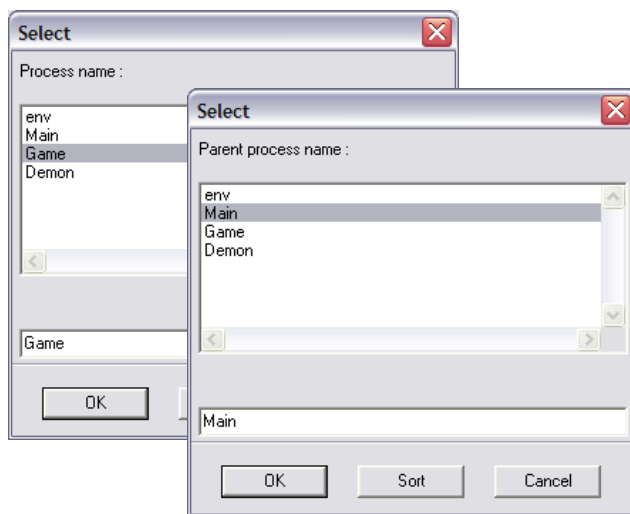
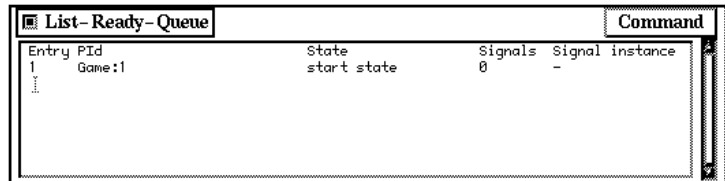


Figure 113: Creating the process Game from process Main

To complete the actions taken when a Newgame signal is received, you must also set the GameP variable in Main, and put Main in the state Game_On. This is done with the commands **Assign-Value** and **Nextstate**. However, these commands operate on the process next to execute, which at this stage is Game, as can be seen in the ready queue printed in the Command window.

Changing the System



Entry	PId	Game	State	start state	Signals	Signal instance
1	Game:1		start state		0	-

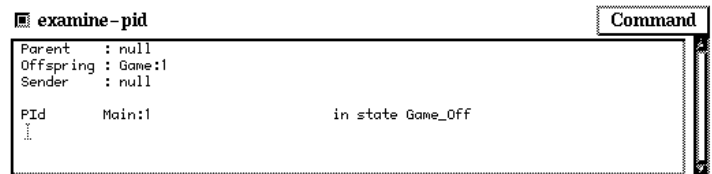
Figure 114: The Ready Queue

Next process to execute is *Game*.

Therefore, you first have to change the current process, also known as the *process scope*.

3. Change the scope by selecting *Set Scope* in the *Examine* menu. Select the process *Main* in the dialog and click *OK*.

The *Examine-PId* command in the *Command* window shows that *Main* is the current process. The variable *GameP* must be set to the value of *Offspring* in *Main*. In the *Command* window, check that this value is *Game:1*.



examine-pid	
Parent	: null
Offspring	: Game:1
Sender	: null
PId	Main:1
	in state Game_Off

Figure 115: Process *Main* is the current process

The *Offspring* of the current process is *Game:1*.

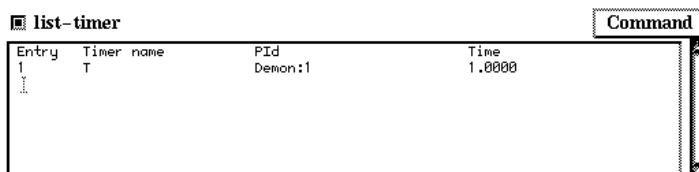
4. Assign the *GameP* variable by selecting *Variable* in the *Change* menu. In the dialogs that follow, select the variable *GameP*, and enter the PId value **Game:1**.
5. To put the *Main* process in the correct state, use the *State* entry in the *Change* menu. Select the state *Game_On* and click *OK*.

The system is now in exactly the state it would be in after the reception of a *Newgame* signal. Even though you have changed the process scope, the next transition to be executed is still the start transition of *Game*. (You can check this by viewing the process ready queue. See [Figure 114 on page 171](#).)

- Execute the next transition and check that the Game process is started.

Changing the State of Timers

In this exercise, we will execute Set and Reset actions on timers directly in the monitor system. At this stage, the timer T is active, as it has been set by the Demon process. You can check this by looking at the List-Timer module in the Command window.



Entry	Timer name	PID	Time
1	T	Demon:1	1.0000

Figure 116: The timer T is active

- Reset the timer by choosing *Reset Timer* in the *Change* menu. Select the timer T and click *OK*.
- Try to execute the next transition and note the message printed:

```
No process instance scheduled for a transition
```

The system is now completely idle, i.e., there are no transitions in the system that can be executed. The Command window shows that both the ready queue and the timer queue is empty. To restart the system you must perform a set operation on timer T in process Demon.

- Choose *Set Timer* in the *Change* menu, select the timer T and enter a time value of 10.
- Execute the next transition and check that the timer was set at time 10 (look at the trace in the main window).

```
*** TIMER signal was sent
*   Timer       : T
*   Receiver    : Demon:1
*** Now        : 10.0000
```

Generating Message Sequence Charts

In this exercise, we will demonstrate the power of Message Sequence Charts as a method of illustrating, in a graphical way, the dynamic behavior of the system. This can easily be done when simulating the system by using *MSC trace*. MSC trace transforms some of the SDL events that take place into MSC events; typically sending of signals and dynamic creation of processes. The trace can then be graphically logged in an MSC Editor during the execution.

Earlier in this tutorial, you drew a Message Sequence Chart which illustrated a simple sequence of messages. We will now run the simulator and generate the MSC trace of the events which actually take place.

What You Will Learn

- To start and stop logging of MSC events
- To trace back from an MSC to an SDL diagram

Initializing the MSC Trace

1. Restart the simulation.
2. Make sure the GR trace is disabled to avoid having the SDL Editor window being updated and raised (select the *SDL Level : Show* entry in the *Trace* menu and verify that system GR trace is 0).

By default, the MSC trace is enabled for the entire system. You must, however, explicitly start the interactive logging of MSC events:

3. Choose *MSC Trace : Start* in the *Trace* menu. A dialog is issued, where you are prompted to specify the amount of symbols to include in the MSC trace. You will include states in the MSC trace, so select 1 and click *OK*.

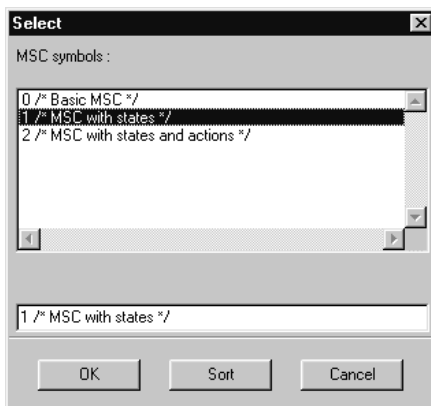


Figure 117: Specifying to include states in the MSC trace

4. An MSC Editor is opened, displaying an MSC diagram named “SimulatorTrace.” If needed, move and resize the window to make it fit on the screen together with the Simulator UI.
 - Before resizing the MSC Editor, you may hide the symbol menu and the text window by using the quick buttons. You should not edit the generated MSC, so there is no need for these windows.



Initially, the system has three active instances, the processes `Main_1_1` and `Demon_1_2`, as well as `env_0` which has been introduced in order to represent the environment to the system:

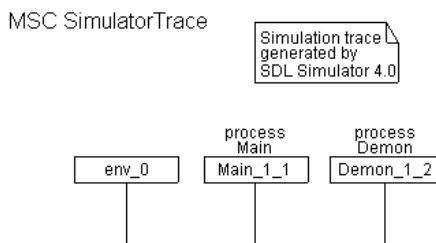


Figure 118: The initial appearance of the MSC

Tracing the Execution in the MSC

1. Send the signal Newgame from the environment. This is now displayed in the MSC Editor as a message sent from the instance env_0.

At this stage, the message is connected to the instance Main_1_1, since it has not yet been consumed. Instead, the message is temporarily drawn as a “lost” message, indicated by the filled circle. You may also see the text “Main_1_1” associated with the circle, indicating the intended receiver of the message.

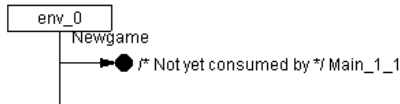


Figure 119: The sent Newgame signal

2. Execute the next transition. A condition symbol with the text Game_Off appears on the instance axis for the process Main. This symbol shows that the process has started executing and has reached the corresponding SDL state, Game_Off.

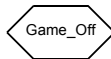


Figure 120: The Condition symbol

3. Execute the next transition. The timer T is set in the Demon process. The vertical coordinate is incremented downwards in the MSC, enhancing the impression of an absolute order of events. Also, a condition symbol with the text Generate is drawn on the instance axis.
4. Execute the next symbol only (use the *Symbol* button). The Main process consumes the Newgame message; the filled circle disappears. Note that the start point and the end point of the message have different vertical positions, since the timer T was set after the message was sent.
5. Execute the next symbol. An instance of the Game process is created, thus adding a new instance head and instance axis. The MSC should now look like this:

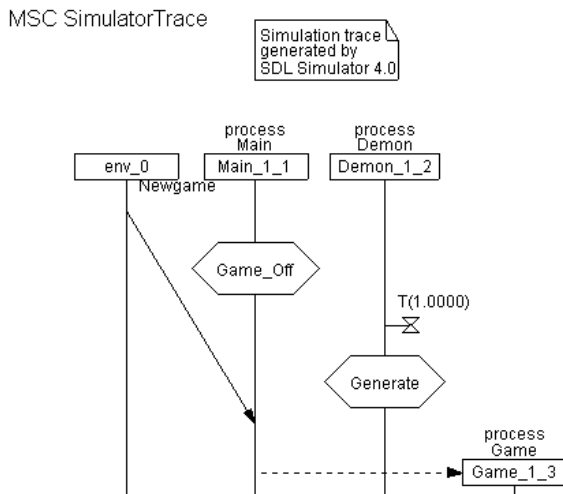


Figure 121: The MSC after creating the Game process

6. Execute the next transition by clicking the *Transition* button twice. The second transition causes the Game process to enter the state Losing.
7. Execute the next three transitions. The timer signal T is consumed and the signal Bump is sent and consumed. The Game process is now in the state Winning. Note how the signal interchange is shown in the MSC.
8. Next we illustrate a message which is consumed immediately. Send the signal Probe from the environment and execute the next transition. First, the message Probe is displayed (marked with filled circle), then it is redrawn, keeping its horizontal alignment.
9. The system responds with the signal Win.
10. Send the signal Result and execute the next transition. In the MSC, you can see that the message Score has the parameter 1.
11. End the game by sending the Endgame signal and execute the next two transitions. The Game process is stopped.

Generating Message Sequence Charts

The MSC should now look like in the figures below. (You may note a dotted horizontal line in the MSC diagram on screen. This indicates where a page break will occur if you would print out the diagram.)

Compare this diagram with the one in [Figure 88 on page 119](#). You will notice differences between the hand-drawn and the generated diagram. These discrepancies are quite natural, since it is impossible to predict the dynamic behavior of a system just by looking at the SDL diagrams.

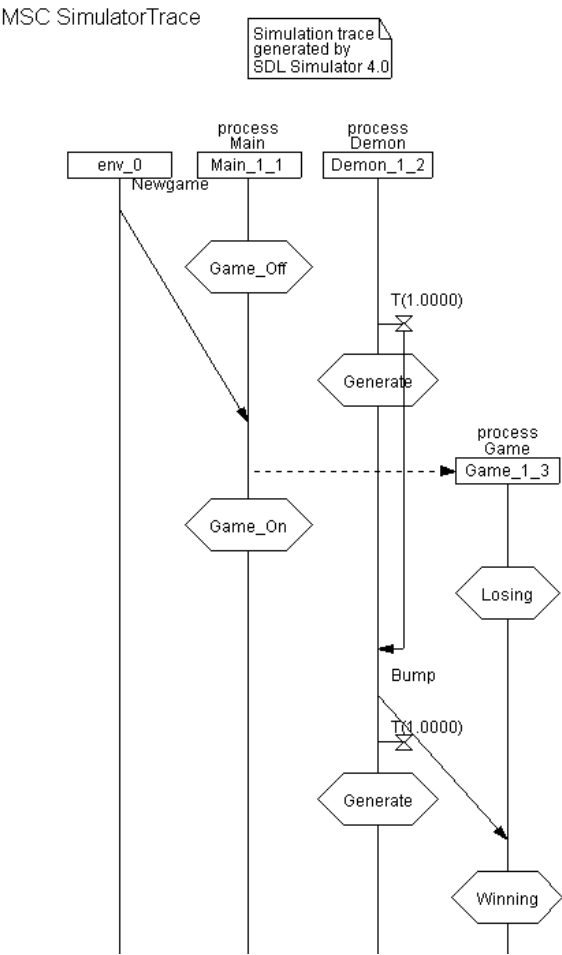


Figure 122: The finished MSC 1(2)

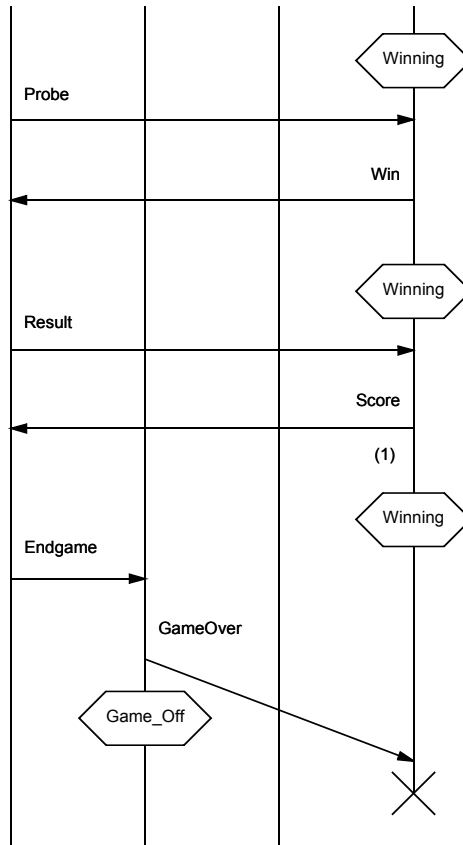


Figure 123: The finished MSC 2(2)

Trace-Back to SDL Diagrams

From the generated MSC diagram, you may obtain a trace back to the SDL source diagrams.

1. From the MSC Editor's *Window* menu, select *Info Window*. A window is opened, containing information about the graphical object which is currently selected. (The amount and type of information depends on what sort of object you have selected.)

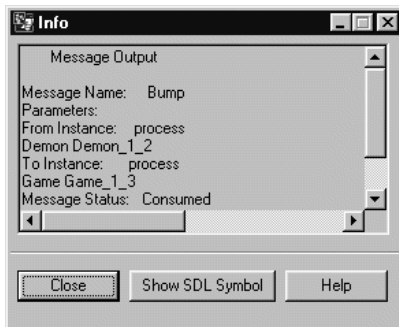


Figure 124: The Info window

The window shows information related to the message *Bump*.

2. Select a few different objects in the MSC Editor and note how the information in the Info window changes.
3. Select the message *Bump* and click on the button *Show SDL Symbol* in the Info window. An SDL Editor is opened, in which the symbol corresponding to the actions of sending or consuming the SDL signal *Bump* is selected:
 - If you have selected the message *Bump* by clicking on a point which is closer to the start point of the message than to the end point, the corresponding SDL **output** symbol will be selected.
 - Otherwise, the corresponding SDL **input** symbol will be selected.

Ending the MSC Trace

1. Stop the logging of MSC events by selecting the *MSC Trace : Stop* entry in the *Trace* menu.
2. In the MSC Editor, save the generated MSC diagram under the file name `SimulatorTrace.msc`.
3. *Exit* the MSC Editor.

The Coverage Viewer

In this final exercise of the SDL Simulator, you will learn to use the Coverage Viewer. The Coverage Viewer is a graphical tool that shows how much of a system has been covered during a simulation in the terms of executed transitions or symbols. By checking the system coverage, you can for instance see what parts of the system that have not been executed in the simulation so far.

What You Will Learn

- To create a coverage file
- To start the Coverage Viewer
- To interpret and change the coverage tree information
- To open a more detailed coverage chart
- To exit the Simulator UI

Starting the Coverage Viewer

1. Restart the simulator.
2. Send the signal Newgame. Execute seven (7) transitions until the printed trace shows that the Game process is in the state Winning.
3. Send the signal Probe and execute the next transition.

Let us see how much of the system we have executed so far. By simply starting the Coverage Viewer from the Simulator UI, the current coverage information is displayed.

4. Select *Coverage* from the *Show* menu. The main window of the Coverage Viewer is opened.

Using the Coverage Viewer



1. If a *symbol coverage tree* is displayed, switch to a transition coverage tree by clicking on the *Tree Mode* quick button.
2. To see all of the *transition coverage tree*, click on the *All Nodes* quick button.

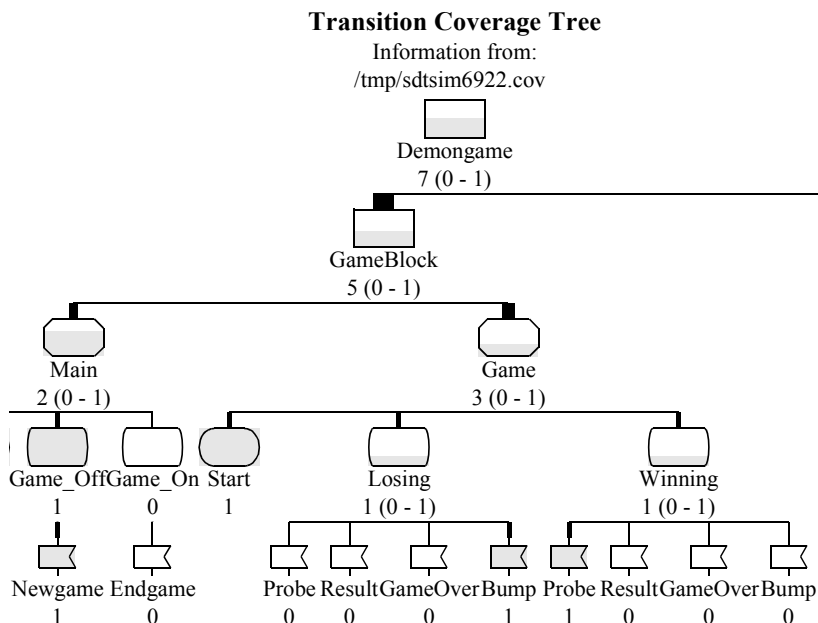


Figure 125: A transition coverage tree

Only a part of the tree is depicted.

- Since the coverage tree is quite large, you may maximize the window (by using the window manager) while working with the Coverage Viewer. When returning to the Simulator UI, you can simply restore the size of the window to its original size.
- Alternatively, you may also zoom out the window contents with a number of clicks on the quick button.



The coverage tree shows the diagram structure of the Demongame system. Beneath each process diagram (Main, Game, Demon), you see all possible transitions in that process, defined by a state and the possible signal inputs from that state. The start transitions are represented by an SDL start symbol.

The number below each symbol in the tree is the number of times the symbol has been executed so far. In addition, each symbol is filled with a gray pattern to indicate to what extent it has been executed. Parts of the system that never have been executed have a zero

value and “empty” symbols. Parts that have been completely traversed by the execution so far have a non-zero value and completely filled symbols.

From the displayed tree, you can find out the following information:

- In the states Losing and Winning, one out of four transitions have been executed. Thus, these state symbols are filled to 1/4.
- In the Main process, two out of three transitions have been executed. Thus, the process symbol is filled to 2/3.
- In the Demon process and the block DemonBlock, all transitions have been executed at least once. Thus, these symbols are completely filled.
- In the system as a whole, a little more than half of all possible transitions have been executed.



3. To only see those transitions that never have been executed, click on the *Least* quick button. You can now see which signals must be sent in which states to execute the rest of the system
 - The *Least* quick button actually shows those symbols that have been executed the **least** number of times. The symbols that are dashed are present only to make the structure complete.



4. In the same way, to only see those transitions that have been executed at least once, click on the *Most* quick button. You can now see which signals have been sent so far in the system.
 - The *Most* quick button actually shows those symbols that have been executed the **most** number of times. In this case, no transition has been executed more than once.



5. To see the whole tree again, click on the *All Nodes* quick button. If you want to see a transition in the SDL Editor, just double-click on one of the signal input symbols or start state symbols. Try this!

The Coverage Viewer can also show *a symbol coverage tree*, i.e. how many times each SDL symbol in the process diagrams have been executed:



6. Switch to a symbol coverage tree by clicking on the *Tree Mode* quick button. Beneath each process diagram, you will now see a small icon for each SDL symbol. To see which SDL symbol an icon represents, double-click the small icon.
7. Switch back to a transition coverage tree and go back to the Simulator UI.

Augmenting the Coverage

1. Execute three more transitions to put the Game process in the state Losing again.
2. Send the signal Result. Execute four more transitions to return to state Winning.
3. Send the signal Endgame. Execute two more transitions to stop the Game process.
4. Check the current system coverage in the Coverage Viewer by selecting *Coverage* from the *Show* menu.
5. Change to a transition coverage tree and show *All Nodes*. As you can see, the Main process has now been completely executed. The Losing and Winning states are also more filled.
6. To see what transitions have still not been executed, you can click the *Least* quick button. If, however, you click the *Most* quick button you will only see the most executed transition (and other symbols), i.e. the input of timer T, not all transitions that have been executed.
7. To see more of the tree, select *Increase Tree* from the *Tree* menu. The input of Bump should now be added. Select the command again and all remaining executed transitions should be added.

Looking at Coverage Details



1. Select the system diagram in the Coverage Viewer. From the *Tools* menu, select *Show Details*, or click the quick button for this. The *Coverage Details* window is opened.

The displayed coverage chart shows how many transitions that have been executed a certain number of times. The chart should contain four bars:

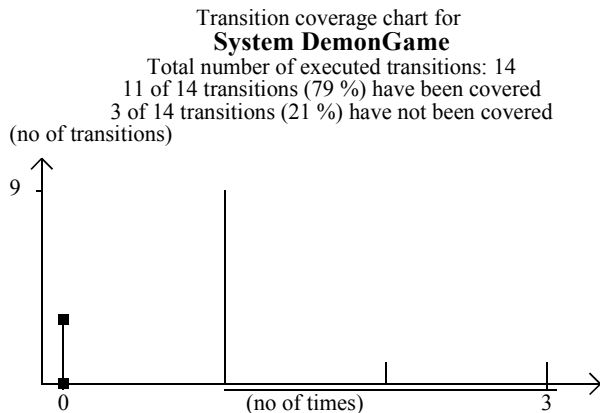


Figure 126: The Coverage Details window

2. Select the bars one at a time and look at the text in the Status Bar at the bottom of the window. You will now see how many transitions that have been executed 0, 1, 2 and 3 times. To see which transitions that have not been executed at all, do as follows:
3. Select the “zero bar” furthest to the left. Click the *Show Editor* quick button. The SDL Editor is opened, showing the three transitions in the Game process that remain to be executed.
 - If the transitions can be found in more than one diagram, a confirmation dialog is issued between each diagram that is opened. Simply click *OK* to see the transitions in the next diagram.
4. Select another symbol in the coverage tree in the main window. The Coverage Details window is now updated to show the coverage chart for that symbol.



5. “Play around” in the Coverage Viewer as much as you like. You should note, however, that the Demongame system is a bit too simple to give full justice to the power of the Coverage Viewer.

Exiting the Simulator UI

You will now close down the Simulator UI.

1. First, exit the Coverage Viewer from the *File* menu.
2. Then, exit the Simulator UI from the *File* menu. You will be asked whether to save the changes to the sets of variables in the Watch window, commands in the Command window, and buttons in the button area. If you choose to save them (with the suggested file names), they will become the default the next time you start the Simulator UI from the same directory.

So Far...

You have now learned how to “animate” an SDL system by generating, executing and tracing a simulator.

If your configuration includes the SDL Explorer tool, we suggest that you proceed with the exercises on the SDL Explorer. These exercises start in [chapter 5, Tutorial: The SDL Explorer](#).

In all cases, the example you have been practising on, the system DemonGame, is rather simple. To deepen your knowledge of the SDL Suite, you may practise on a number of exercises that illustrate the advantages of SDL-92 when adopting an object-oriented design methodology. These exercises are described in [chapter 6, Tutorial: Applying SDL-92 to the DemonGame](#).

Tutorial: The SDL Explorer

The SDL Explorer is the tool that you use for validating the behavior of your SDL systems, using state space exploration techniques. In this chapter, you will practice “hands-on” on the DemonGame system.

To be properly assimilated, this tutorial therefore assumes that you have gone through the exercises that are available in [chapter 3, *Tutorial: The Editors and the Analyzer*](#) as well as [chapter 4, *Tutorial: The SDL Simulator*](#).

In order to learn how to use the Explorer, read through this entire chapter. As you read, you should perform the exercises on your computer system as they are described.

Purpose of This Tutorial

The purpose of this tutorial is to make you familiar with the essential validation functionality in the SDL Suite. With validation we mean exploring the state space of an SDL system with powerful methods and tools that will find virtually any kind of possible run-time errors that may be difficult to find with regular simulation and debugging techniques.

This tutorial is designed as a guided tour through the SDL Suite, where a number of hands-on exercises should be performed on your computer as you read this chapter.

We have on purpose selected a simple example that should be easy to understand. It is assumed that you have a basic knowledge about SDL — this chapter is **not** a tutorial on SDL.

It is assumed that you have performed the exercises in [chapter 3, Tutorial: The Editors and the Analyzer](#) as well as [chapter 4, Tutorial: The SDL Simulator](#) before starting with the tutorial on the SDL Explorer.

Note: C compiler

You must have a C compiler installed on your computer system in order to validate an SDL system. Make sure you know what C compiler(s) you have access to before starting this tutorial.

Note: Platform differences

This tutorial, and all tutorials that are possible to run on both the UNIX and Windows platform, are described in a way common to both platforms. In case there are differences between the platforms, this is indicated by texts like “on UNIX”, “Windows only”, etc. When such platform indicators are found, please pay attention only to the instructions for the platform you are running on.

Normally, screen shots will only be shown for one of the platforms, provided they contain the same information for both platforms. This means that **the layout and appearance of screen shots may differ** slightly from what you see when running the SDL Suite in your environment. Only if a screen shot differs in an important aspect between the platforms will two separate screen shots be shown.

Generating and Starting an SDL Explorer

In addition to simulating a system, it is also possible to validate the system using the SDL Explorer. An explorer can be used to automatically find errors and inconsistencies in a system, or to verify the system against requirements.

In the same way as for a simulator, you must generate an executable explorer and start it with a suitable user interface.

Note:

In order to generate an explorer that behaves as stated in the exercises, you should use the SDL diagrams that are included in the distribution instead of your own diagrams. To do this:

- **On UNIX:** Copy all files from the directory
`$telelogic/sdt/examples/demongame`
to your work directory `~/demongame`.
- **In Windows:** Copy all files from the directory
`C:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\examples\demongame`
to your work directory
`C:\IBM\Rational\SDL_TTCN_Suite6.3\work\demongame`.

If you generate an explorer from the diagrams that you have created yourself, the scheduling of processes (i.e. the execution order) may differ.

If you choose to copy the distribution diagrams, you must then re-open the system file `demongame.sdt` from the Organizer.

What You Will Learn

- To quickly generate and start an executable explorer

Quick Start of an SDL Explorer

An explorer can be generated and started in the same way as described earlier for the simulator, i.e., by using the *Make* dialog and the *Tools* menu in the Organizer. However, we will now show a quicker way.

1. Make sure the system diagram icon is selected in the Organizer.
2. Click the *Explore* quick button. The following things will now happen, in rapid succession:
 - An executable explorer is generated. Messages similar to when generating a simulator are displayed in the Status Bar, ending with “Analyzer done.” This is the same action as manually using the *Make* dialog and selecting an explorer kernel. If you like, you can verify that an explorer kernel has been used by looking at the tail of the Organizer log.
 - A graphical user interface to the explorer is started. The status bar of the Organizer will read “Explorer UI started.” This is the same action as manually selecting *Explorer UI* from the *Tools* menu.
 - The generated explorer is started. The Explorer UI shows the message “Welcome to SDL EXPLORER.” This is the same action as manually using the *Open* quick button and selecting the executable explorer (named `demongame_XXX.val` (**on UNIX**), or `demongame_XXX.exe` (**in Windows**), where the `_XXX` suffix is platform or kernel/compiler specific).



Note:

If you receive errors from the Make process (in the Organizer Log window) or if no Explorer is started, do as follows:

- Open the *Make* dialog and change to a Validation kernel reflecting the C compiler used on your computer system, e.g. *gcc-Validation* or *Microsoft Validation*.
- Click the *Full Make* button and check that no errors were reported.
- Click the *Explore* quick button again. An Explorer should now be started as described above.

Generating and Starting an SDL Explorer

The Explorer UI looks like this:

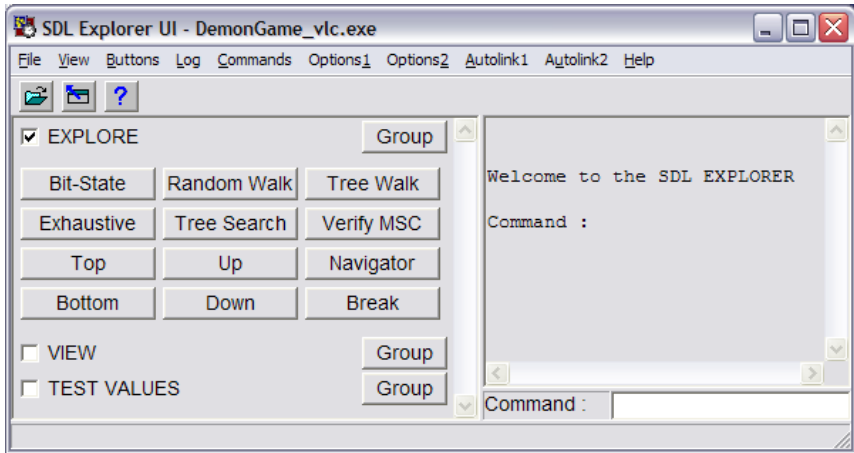


Figure 127: The main window of the Explorer UI

As you can see, the graphical user interface of an explorer is very similar to a simulator GUI, which you have learned to use in the previous exercises. However, the button modules to the left are different and a few extra menus are available.

An explorer contains the same type of *monitor system* as a simulator. The only difference is the set of available commands.

When an explorer is started, the static process instances in the system are created (in this case Main and Demon), but their initial transitions are not executed. The process in turn to be executed is the Main process. You can check this by viewing the process ready queue:

1. Locate the button module *View* in the left part of the window, and click the *Ready Q* button. The first entry in the ready queue is Main, waiting to execute its start transition.
 - If the *View* module appears to be empty, you have to click the toggle button to the left of the module's name. The button module is then expanded. You may collapse and expand any button module by using these toggle buttons:



Figure 128: A collapsed button module

- The buttons in the *View* module execute the same type of commands as those in the Simulator UI.
- 2. If required, resize the Explorer UI window so that all button modules are visible. You may also reduce the width of the text area. In the exercises to come, you will have a number of windows open at the same time.

Basics of an SDL Explorer

Before you start working with the explorer exercises, you should have an understanding about the basic concepts of the SDL Explorer.

- When examining an SDL system using the explorer, the SDL system is represented by a structure called a *behavior tree*. In this tree structure, a node represents a state of the complete SDL system. The collection of all such system states is known as the *state space* of the system.
- By moving around in the behavior tree, you can explore the behavior of the SDL system and examine each system state that is encountered. This is called *state space exploration*, and it can be performed either manually or automatically.
- The size and structure of the behavior tree is determined by a number of *state space options* in the explorer. These options affect the number of system states generated for a transition in an SDL process graph, and the number of possible branches from a state in the behavior tree.

Navigating in a Behavior Tree

In this first exercise, we will explore the state space of the Demongame system by manually navigating in the behavior tree. The explorer will then behave in a way similar to when running a simulator. However, there are also important differences, which will be pointed out.

By default, the explorer is set up in a way that results in a state space as small as possible. In this set-up, a transition between two states in the behavior tree always corresponds to a complete transition in the SDL process graphs. Also, the number of possible branches from a state is limited to a minimum.

What You Will Learn

- To use the Navigator tool
- To get printed trace and GR trace

Setting Up the Exploration

When interactively exploring the behavior tree, an explorer tool called the *Navigator* is used.

1. Start the Navigator by clicking the *Navigator* button in the *Explore* module. The Navigator window is opened:

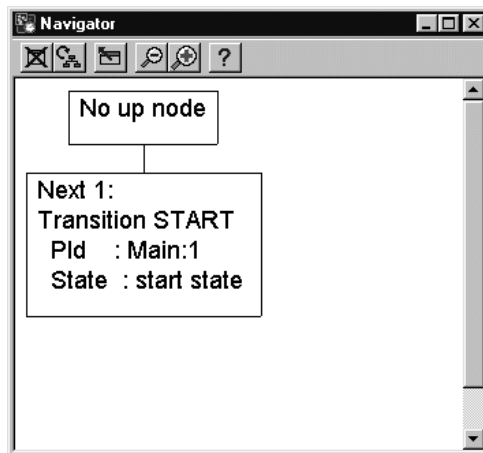


Figure 129: The Navigator tool

The Navigator shows part of the behavior tree around the current system state. In general, the upper box represents the behavior tree transition leading to the current state, i.e., the transition that just has been executed. The boxes below represent the possible tree transitions from the current state. They are labelled *Next 1*, *Next 2*, etc. and have not yet been executed.

Since the system now is in its start state, there is no *up node*. The only next node is the start transition of Main.

2. To be able to see the printed trace familiar from simulation, open the Command window from the *View* menu. (The trace is not printed in the main window of the explorer.)
3. To switch on GR trace of SDL symbols, select *Toggle SDL Trace* from the *Commands* menu in the Explorer window; SDL trace is now enabled. However, an SDL Editor will not be opened until the first transition is executed.

Using the Navigator

1. In the Navigator, execute the next transition by double-clicking on the *Next 1* node. The following happens, in order:
 - In the Navigator, the *Up 1* node shows the just executed transition, while the *Next 1* node shows the next possible transition, the start transition of Demon. You have now moved down to a system state in the next level of the behavior tree.

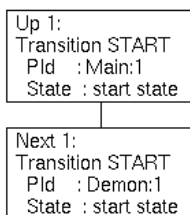


Figure 130: The last and next transition

- An SDL Editor is opened and the symbols that were just executed becomes selected. Note the difference compared to the simulator, where the SDL Editor instead selects the next symbol to be executed.

Navigating in a Behavior Tree

- The Command window shows the printed trace for the executed transition, the start transition of Main (you may have to scroll or resize the window to see the trace):

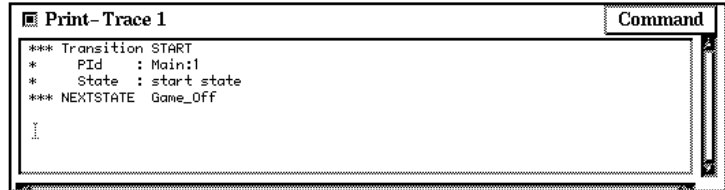


Figure 131: The printed trace for the executed transition

2. If needed, move and resize all opened windows to make them completely visible and still fit on the screen together.
3. Double-click the *Next 1* node to execute the next transition. The start transition of Demon is traced in the Command window and in the SDL Editor.

At this stage, neither of the two active processes can continue without signal input: Main awaits the signal Newgame from the environment, and Demon awaits the sending of the timer signal T. These are the two transitions from the current state now shown in the Navigator as *Next 1* and *Next 2*. As you can see, the transitions in the boxes are described by the same type of information as in a printed trace.

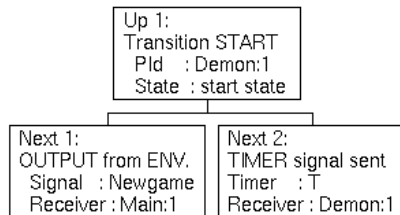
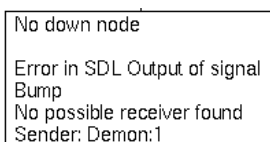


Figure 132: Transition descriptions in the Navigator

This means that the explorer gives information of all possible transitions from the current system state, even though they have not been executed yet. (This information cannot easily be obtained when running a simulator.)

4. Send the timer signal by double-clicking the *Next 2* node. The Command window tells us that the timer signal is sent and the Navigator shows that the next transition is the input of the timer T.
5. Execute the next transition by double-clicking the *Next 1* node. This is where the dynamic error in the Demongame system occurs, as explained in the simulator tutorial earlier (see [“Dynamic Errors” on page 153 in chapter 4, Tutorial: The SDL Simulator](#)). Instead of showing the next transition, the Navigator displays the error message in the next box.



```
No down node
Error in SDL Output of signal
Bump
No possible receiver found
Sender: Demon:1
```

Figure 133: The dynamic error

- The error message can also be found in the tail of the Command window, if you scroll the Print-Trace module.

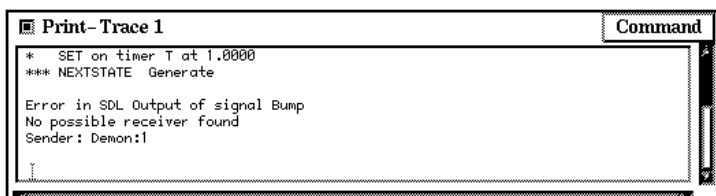


Figure 134: The tail of the Print-Trace module

We cannot go further down this branch of the behavior tree, since a reported error by default truncates the tree at the current state. Instead, we will back up to the state where we could select the output of Newgame.

6. Double-click the *Up 1* node to go back to the previous state. Repeat this action again to go to the state we were in after step 3 above. This way of backing up in the execution is not possible when running a simulator, as you may have noticed when running the Simulator tutorial.

Navigating in a Behavior Tree

You should also note that the *Next 2* node is marked with three asterisks “***”. This is used to indicate that this is the transition we have been backing up through:

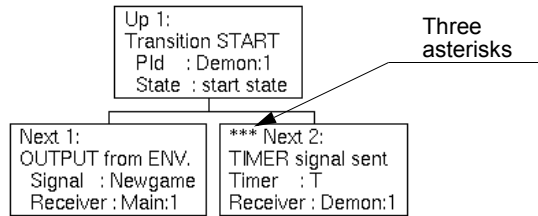


Figure 135: Marking a transition that has been backed through

7. Execute the *Next 1* transition instead. The printed trace shows that the signal *Newgame* was sent from the environment. The Main process is ready to receive the signal. Note that you do not have to send the signal yourself; this is taken care of automatically by the explorer.
8. Execute the next transition. The printed trace and the SDL trace show that Main now is in the state *Game_On*. The Navigator displays the start transition of the newly created Game process.
9. Execute the start transition of Game. The Navigator will now show the different signal inputs that are required to continue execution: *Endgame*, *Probe*, *Result*, and the timer *T*.

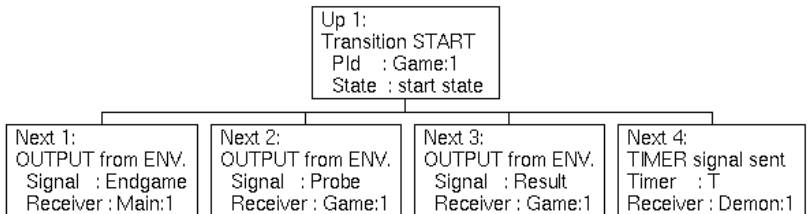


Figure 136: Signal inputs required for continued execution

If the number of transitions from a state is large, it may be difficult to see them all in the Navigator when a tree structure is used. To overcome this problem, you can change the display to a list structure.



10. Click the *Toggle Tree* quick-button to see how the list structure looks like. Now it is easier to see the possible signals.

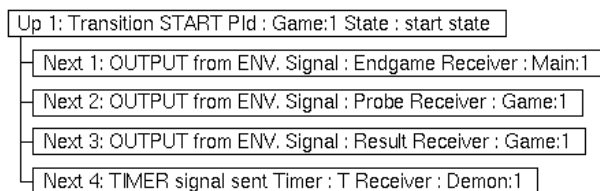


Figure 137: The list structure

11. Change back to the tree structure.

We will not continue further down in the behavior tree in this exercise. [Figure 138 on page 199](#) shows the part of the behavior tree we have explored so far. The nodes in the figure represent states of the complete SDL system. Each node lists the active process instances that have changed since the previous system state, what process state they are in and the content of their input queues. The arrows between the nodes represent the possible tree transitions. They are tagged with a number and the SDL action that causes the transition. The arrow numbers are the same numbers as printed in the *Next* nodes in the Navigator.

Note that this is a somewhat different view of the behavior tree compared to the Navigator. In the Navigator, the nodes represent the tree transitions and the process states are not shown.

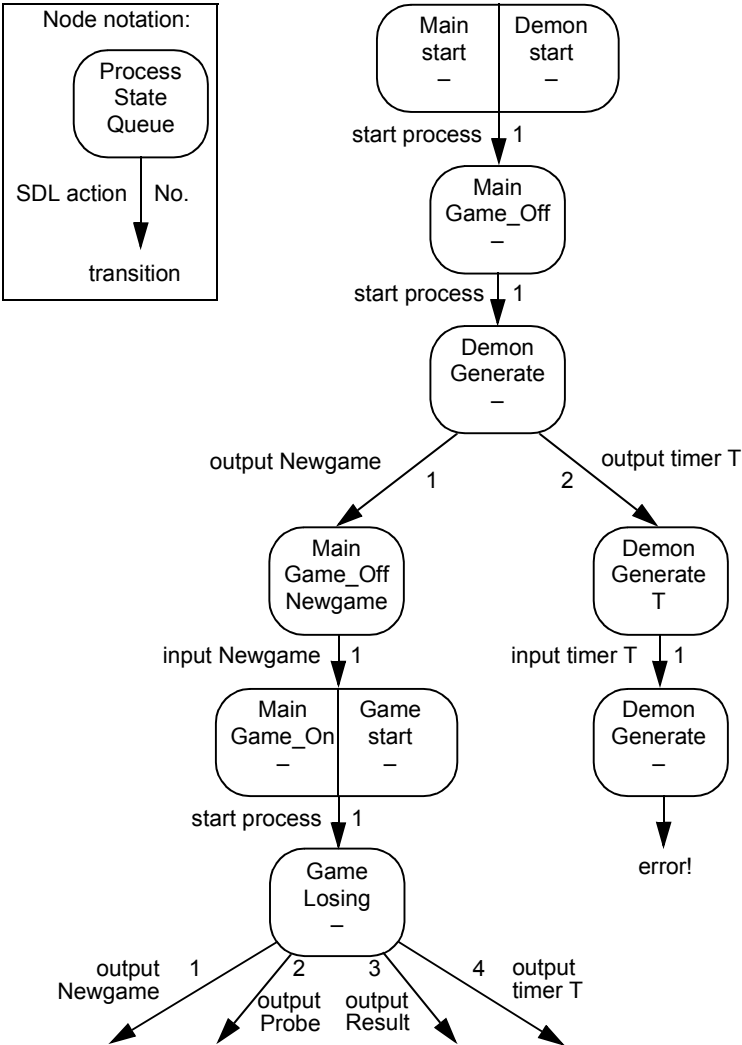


Figure 138: A Demongame behavior tree

More Tracing and Viewing Possibilities

In this exercise, we will take a look at some of the additional tracing and viewing possibilities of the SDL Explorer.

What You Will Learn

- To print a complete trace from the start state
- To use the view commands
- To use the MSC trace facility
- To go to a state by using the path commands

Using the View Commands

1. Make sure you are still in the same state as after the last step in the previous exercise.

To see a complete printed trace from the start state to the current state, you can use the **Print-Trace** command. As parameter, it takes the number of levels back to print the trace from.

2. On the input line of the Explorer UI, enter the command `pr-tr 9` (you can use any large number). The trace is printed in the text area of the main window. This trace gives an overview of what has happened in the SDL system so far.
3. The SDL Explorer supports the same viewing possibilities as the SDL Simulator. Click the *Timer List* button in the *View* module to list the active timer set by the Demon process.
4. Examine the GameP variable in the Main process by first setting the scope to the Main process (click the *Set Scope* button and select the Main process), and then clicking the *Variable* button and selecting the GameP variable.
 - You may also use the Watch window in the explorer to continuously monitor the values of variables.

Using MSC Trace

In addition to textual and graphical traces, the SDL Explorer can also perform an MSC trace.

1. First, turn off SDL trace by selecting *Toggle SDL Trace* from the *Commands* menu. Then, turn on MSC trace from the same menu. An MSC Editor is opened, showing a Message Sequence Chart for the trace from the start state to the current state.
 - You may also close down the SDL Editor to avoid having too many windows on-screen.

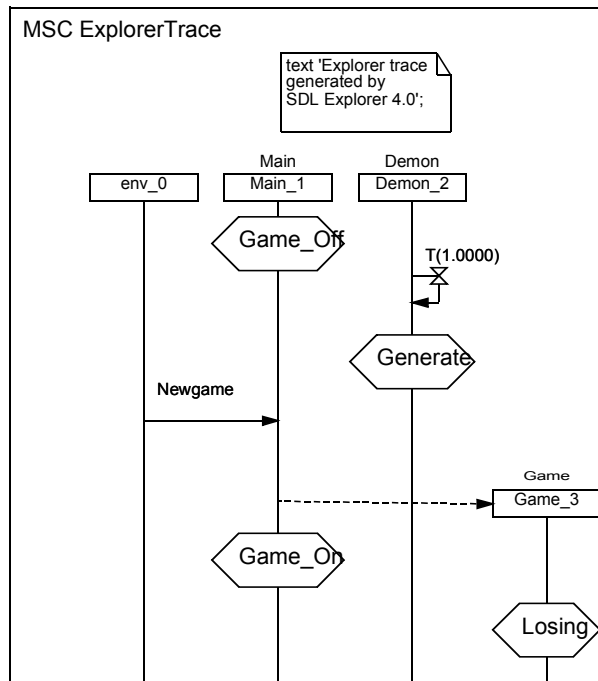


Figure 139: The current MSC trace

2. When the MSC appears, execute, with a double-click, one of the signal transitions in the Navigator, e.g. Probe. The message is appended to the MSC (but it is not yet consumed).
3. Go up a few levels in the Navigator.

Note how the selection in the MSC Editor changes to reflect the MSC event corresponding to the current state!

4. Go down again, but select a different path than before, i.e., send one of the other signals.

Note how the MSC diagram is redrawn to show the new behavior of the system!

5. *Toggle MSC trace* off in the *Commands* menu. Unless other MSC diagrams were opened, the MSC Editor is closed.

Going to a State Using Path Commands

You can use the commands **Print-Path** and **Goto-Path** to return to a state where you have been before.

1. Execute the command **Print-Path** from the input line. The output represents the path taken in the behavior tree from the start state to the current state.

```
Command : print-path  
1 1 1 1 3 0
```

- The numbers in the path are the same as the transition numbers in the Navigator, and the arrow numbers shown in [Figure 138 on page 199](#).
2. Go up a few levels in the Navigator.
 3. In the text area, locate the path printed by the **Print-Path** command above (you may have to scroll the text area). **On UNIX**, select the numbers in the path with the mouse by dragging the mouse to the end of the line. Make sure you select the final zero.
 4. In the input line, enter **goto-path** and the path printed by the **Print-Path** command. **On UNIX**, paste in the path numbers by positioning the mouse pointer at the end of the entered text and clicking the **middle** mouse button.
 5. Hit **<Return>** to execute the command. You now end up in the previous state.
 - If you make an error while entering the path numbers, you can clear the input line by using the **<Down>** arrow key and try again.

Validating an SDL System

In the previous exercises, we have navigated manually in the behavior tree. We have also found an error situation by studying the Navigator and the printed trace in the Command window.

In this exercise, we will show how to find errors and possible problems by automatically exploring the state space of the Demongame system. This is referred to as *validating* an SDL system.

What You Will Learn

- To perform an automatic state space exploration
- To examine reported errors using the Report Viewer
- To change state space and exploration options
- To restrict the state space without affecting the behavior
- To check the system coverage of an exploration
- To use user-defined rules
- To perform a random walk exploration

Performing a Bit State Exploration

Automatic state space exploration can be performed using different algorithms. The algorithm called *bit state exploration* can be used to efficiently validate reasonably large SDL systems. It uses a data structure called a *hash table* to represent the system states that are generated during the exploration.

An automatic state space exploration always starts from the current system state. Since we want to explore the complete Demongame system, we must first go back to the start state of the behavior tree.

1. Go to the top of the tree by clicking the *Top* button in the *Explore* module.
2. Start a bit state exploration by clicking the *Bit-State* button. After a few seconds, a tool called the *Report Viewer* is opened. We will soon describe this window; in the meantime, just move it away from the main window.
3. For a small system such as Demongame, the exploration is finished almost immediately and some statistics are printed in the text area. They should look something like:

```
** Starting bit state exploration **
Search depth      : 100
Hash table size   : 1000000 bytes

** Bit state exploration statistics **
No of reports: 1.
Generated states: 2569.
Truncated paths: 156.
Unique system states: 1887.
Size of hash table: 8000000 (1000000 bytes)
No of bits set in hash table: 3642
Collision risk: 0 %
Max depth: 100
Current depth: -1
Min state size: 68
Max state size: 124
Symbol coverage : 100.00
```

Of the printed information, you should note the following:

- Search depth : 100
The *search depth* limits the exploration; it is the maximum depth, or level, of the behavior tree. If this level is reached during the exploration, the current path in the tree is truncated and the exploration continues in another branch of the tree. It is possible to change the search depth by setting an option in the SDL Explorer UI.
- No of reports: 1.
The exploration found one error situation. This error will be examined in the next exercise.
- Truncated paths: 156.
The maximum depth was reached 156 times, i.e., there are parts of the behavior tree that were not explored. This is a normal situation for SDL systems with infinite state spaces. Demongame is such a system, since the game can go on forever.
- Collision risk: 0 %
The risk for collisions was very small in the hash table that is used to represent the generated system states. If this value is greater than zero, the size of the hash table may have to be increased by setting an option; otherwise, some paths may be truncated by mistake. This situation will not occur in this tutorial.

- Symbol coverage : 100.00

All SDL symbols in the system were executed during the exploration. If the symbol coverage is not 100%, the validation cannot be considered finished. This situation will occur in a later exercise.

Examining Reports

The error situations reported from a state space exploration can be examined in the Report Viewer. The Report Viewer window displays the reports in the form of boxes in a tree structure.

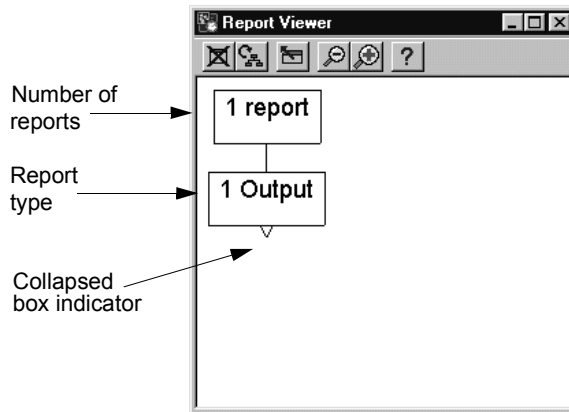


Figure 140: The Report Viewer

- The top box shows how many reports there are (in this case only one).
- On the next level in the report tree, there is one box for each type of report, stating the number of reports of that type.
- On the next level, it is possible to see the actual reports. However, this level of the tree is by default collapsed, indicated by the small triangle icon below the report type boxes.

1. To expand the report, double-click on the report type box *Output*. You will now see a box reporting the error we have found manually earlier. In addition, the tree depth of the error situation is shown.

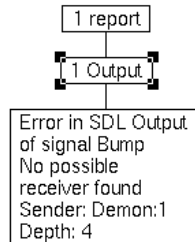


Figure 141: An expanded report

If you look in the Navigator and Command windows, you can see that the SDL Explorer is still in the start state of the system, even though a state space exploration has been performed. We will now go to the state where the error has occurred.

2. Double-click the report box in the Report Viewer. The following things will now happen:
 - The printed trace of the error situation is displayed in the text area of the Explorer UI and in the Command window.
 - The Navigator moves to the error state and displays the error.
 - An MSC Editor is opened, showing the MSC trace to the current state. You can see that the signal Bump was not received by any process, since the Game process has not yet been created. You should move the MSC Editor window so that it does not cover the other windows.

Once you have used the Report Viewer to go to a reported situation, you can easily move up and down the path to this state. Simply use the *Up* and *Down* buttons in the *Explore* module, instead of double-clicking a node in the Navigator:

3. Move up two steps by using the *Up* button. Of the two transitions possible from this state, the one that is part of the path leading to the error is indicated by three asterisks "***" (see [Figure 135 on page 197](#)). This is the transition chosen when using the *Down* button.

4. Move up to the top of the tree (click the *Top* button in the *Explore* module). Move down again to the error by using the *Down* button repeatedly.

Note that you do not have to chose which way to go when the tree branches. The path to the error is remembered by the explorer until you manually chose another transition.

Exploring a Larger State Space

We will now run a more advanced bit state exploration, with a different setting of the state space options. This will make the state space much larger, so that more error situations can be found.

1. Go back to the top of the behavior tree (use the *Top* button).
2. In the *Options1* menu, select *Advanced*. This sets a number of the available state space options in one step, as you can see by the commands executed in the text area:

```
Command : def-sched all
```

```
Command : def-prio 1 1 1 1 1
```

```
Command : def-max-input-port 2  
Max input port length is set to 2.
```

```
Command : def-rep-log maxq off  
No log for MaxQueueLength reports
```

Note that the Navigator now shows two possible transitions from the start state; this is an immediate effect of the larger state space.

3. In addition, we will increase the search depth of the exploration from 100 (the default) to 300. From the *Options2* menu, select *Bit-State: Depth*. In the dialog, enter **300** and click **OK**.

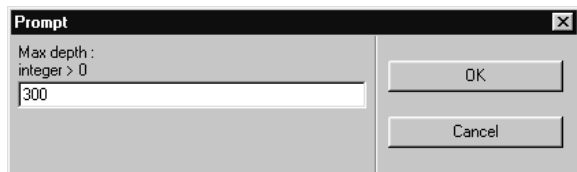


Figure 142: Specifying Depth = 300

Since the behavior tree becomes much larger with these option settings, the exploration will take longer to finish. We will therefore show how to stop the exploration manually.

4. Start a new bit state exploration. In the text area, a status message is printed every 20,000 transitions that are executed. Stop the exploration after one of the first status messages by pushing the *Break* button in the *Explore* module. The text area should now display something like this:

```
*** Break at user input ***

** Bit state exploration statistics **
No of reports: 2.
Generated states: 50000.
Truncated paths: 1250.
Unique system states: 21435.
Size of hash table: 8000000 (1000000 bytes)
No of bits set in hash table: 41557
Collision risk: 0 %
Max depth: 300
Current depth: 235
Min state size: 68
Max state size: 168
Symbol coverage : 100.00
```

Note:

If the exploration finishes by itself before you have had a chance to stop it manually, redo this exercise from step [1. on page 207](#) but increase the search depth even more, e.g. 400 or 500.

Note the following differences in the printed information compared to the previous exploration:

- No of reports: 2.
The exploration found an additional error situation. This is an effect of more transitions being able to execute from each state in the behavior tree.
- Max depth: 300
Current depth: <number>
The exploration was at the printed depth in the behavior tree at the moment it was stopped. However, since the exploration uses a depth-first algorithm, the maximum depth of 300 was reached at an earlier stage. The exploration may be continued from the current depth if you wish to explore the remaining parts of the behavior tree.

Validating an SDL System

5. In the Report Viewer, open the two report type boxes to see both reports with a double-click on each. The Report Viewer window should now look something like:

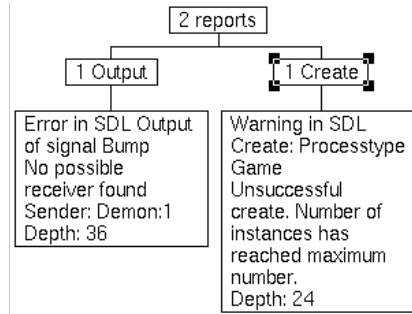


Figure 143: The two reports as displayed in the window

6. For now, just note on which depth each of the reported situations occurred; **do not** double-click any of the reports. (The depths may be different from the ones shown in the figure.)
7. Continue the exploration by clicking the *Bit-State* button again. A dialog is opened, asking if you would like to continue the interrupted exploration or restart it from the beginning.

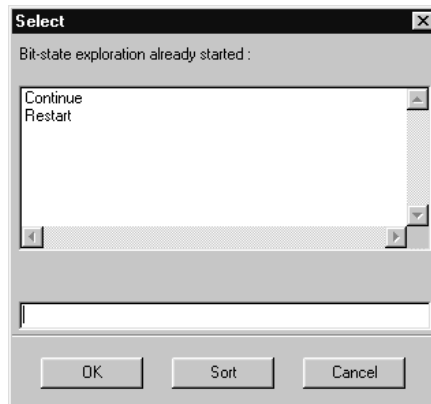


Figure 144: Continuing the exploration

8. In the dialog, select *Continue* and click *OK*. Wait for the exploration to finish by itself.
9. In the Report Viewer, open the two reports again. Note that the depth values have changed. This is because only one occurrence of each report is printed; the one found at the lowest depth so far.
10. Go to the state where an unsuccessful create of the Game process was reported (double-click the Create report).

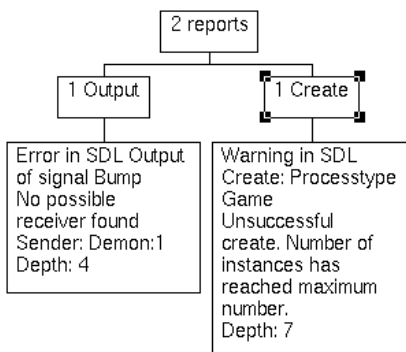


Figure 145: The report about an unsuccessful process create

11. To see what caused the unsuccessful create, look at the MSC trace.

At the receipt of the last Newgame signal, the Main process attempts to create a Game process. However, the already active Game process has not yet consumed the previous GameOver signal, and has therefore not been terminated. Since you cannot have more than one instance of the Game process in the Demongame system, the process create could not be executed!

Restricting the State Space

The SDL Explorer makes it possible to limit the state space in several different ways. We will now explore one of these methods that in many cases is very efficient. This is done by using the **Define-Variable-Mode** command.

This command is used to instruct the Explorer to ignore certain variables when matching states during the state space exploration. The

mode can for each variable be set to either “Skip” or “Compare”. The implication of setting the mode to “Skip” is that the search may be pruned even if a new state is encountered during the search. This happens if the only difference between the new state and a previously visited state is that the values of some of the skipped variables are different.

We will now apply this to our DemonGame system. The variable *Count* in the *Game* process keeps track of the current score for the game, and the value of this variable does not have any real impact on the behavior of the system. So, we will now instruct the explorer to ignore this variable when performing a search.

1. Go to the top of the tree by clicking on the *Top* button.
2. Enter the command **define-variable-mode** in the command line in the Explorer UI, select the *Game* process in the first dialog, the *Count* variable in the second dialog and *Skip* in the last dialog. You have now instructed the Explorer to ignore the Count variable.
3. Start a bit state exploration by clicking on the *Bit-State* button. (Select to *Restart* the exploration if a dialog is opened.)
4. When the search stops compare it with the previous exploration. The only difference between the two explorations is that the second one ignores the Count variable. However, while the first exploration took a long time to finish, the second one only took a few seconds! The printed statistics show very small numbers in comparison.

The lesson to learn from this is that it in many cases it is possible to drastically reduce the time needed for explorations by checking the variables in the system. Look for variables that do not have any impact on the behavior (i.e. that does not influence decision statements or the expression used in an “output to” statement). Also look for variables that do not change their value during the exploration. This can for example be arrays that are initialized at system start up but then never changes (or at least not changed in the intended exploration). The mode for these types of variables should be set to “Skip”.

Checking the Validation Coverage

If the symbol coverage after an automatic state space exploration is less than 100%, the Coverage Viewer can be used to check what parts of the system that have not been executed. To attain a symbol coverage less than 100% for the Demongame system, we will set up the exploration in a special way.

1. Go to the top of the tree.
2. First, we need to restore the smaller, default state space. Select *Default* from the *Options1* menu. Note that the Navigator changes back to display only a single possible transition from the top node.
3. To avoid reaching all system states, we will reduce the search depth of the exploration from 100 to just 10. Use the *Bit-State: Depth* menu choice from the *Options2* menu and specify a maximum depth of 10.
4. Start a bit state exploration. The printed statistics should now inform you that the symbol coverage is about 82%.
 - If the symbol coverage still is 100%, select *Reset* from the *Options1* menu and repeat steps 3 and 4 above.
5. To find out which parts of the Demongame system that have not been reached, open the Coverage Viewer from the *Commands* menu.

A symbol coverage tree is displayed, showing all symbols which have not been executed yet.



6. Change to a transition coverage tree by clicking the *Tree Mode* quick-button.

Transition Coverage Tree

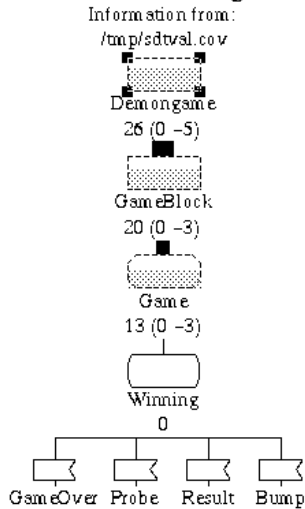


Figure 146: The transition coverage tree

You can now see that none of the transitions from the state **Winning** in the **Game** process has been executed. To explore this part of the system in the explorer, you can go to the state **Winning** and start a new exploration from there. How to do this is explained in the following exercises.

Going to a State Using User-Defined Rules

To go to a particular system state, you could use the Navigator to manually find the state by studying the transition descriptions and the printed trace in the Command window. This can be both tedious and difficult, especially for larger systems than **Demongame**. Instead, we will show an easier way: by using a *user-defined rule*.

When performing state space exploration, the explorer checks a number of predefined rules in each system state that is reached. It is when such a rule is satisfied that a report is generated.

In this exercise, we will show how to define a new rule to be checked during state space exploration. The rule will be used to find the state Winning in the Game process.

1. Make sure you still are at the top of the behavior tree.
2. Define a new rule by selecting *Define Rule* from the *Commands* menu. In the dialog that appears, enter the rule definition **state(Game:1)=Winning**

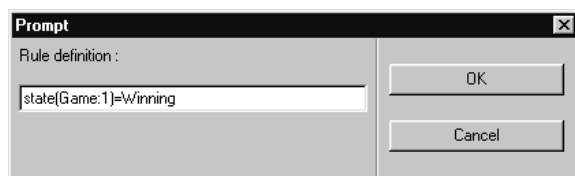


Figure 147: Specifying a new rule

This very simple rule says that the state of the process instance Game:1 must be equal to Winning. By defining the rule, a report will be generated when a state space exploration reaches a state that satisfies the rule.

3. Start a bit state exploration. Since we have not changed any of the options since the last exploration the same statistics will be printed, with the exception that an additional report is generated.
4. From the Report Viewer, go to the reported situation where the user-defined rule was satisfied. You have now reached the first place in the behavior tree where the Game process is in the state Winning.
5. We now instruct the explorer to use this state as the root of the behavior tree. To do this, enter the command **define-root** on the input line and select *Current* in the dialog.

We can now change options, define a new rule or load an MSC. These new settings will then be used in all explorations based on the new root. Also all list/goto-path commands will use the path from the new root and the MSC trace will give the trace from the new root.

6. Before continuing, do not forget to clear the user-defined rule. To do this, enter the command **clear-rule** on the input line.

In our case we will only clear the rule and start another type of state space exploration from this state; a random walk.

Performing a Random Walk

Apart from bit state exploration, there is another exploration method known as *random walk*. A random walk simply explores the behavior tree by repeatedly choosing a random path down the tree. This is mainly useful for SDL systems where the state space can be very large. But also for a small system like Demongame, it can be as effective as other exploration methods.

1. Start a random walk exploration from the current state by clicking the *Random Walk* button. From the printed statistics, you can see that the symbol coverage now has become 100%.
2. Load the Coverage Viewer with the new coverage information by selecting *Show Coverage Viewer* from the *Commands* menu. Change to transition coverage and display the whole tree. Note that all transitions have executed a large number of times. When the exploration selects a random path down the tree, there is no mechanism to avoid that already explored paths are explored once more. Therefore, the same transition may be executed any number of times.
3. Exit the Coverage Viewer from the *File* menu.
4. Reset the system by selecting *Reset* from the *Options1* menu. You are now back at the top of the tree, and the root of the tree is reset to the original root, the start state of the system.

Verifying a Message Sequence Chart

Another main area of use for an SDL Explorer is to verify a Message Sequence Chart. To verify an MSC is to check if there is a possible execution path for the SDL system that satisfies the MSC. This is done by loading the MSC and performing a state space exploration set up in a way suitable for verifying MSCs.

What You Will Learn

- To verify an MSC

Verifying a System Level MSC

In this exercise, we will verify one MSC made on the system level, i.e., an MSC that only defines signals to and from the environment. The name of the MSC file is `SystemLevel.msc` and is located in the same directory as the remaining files for the DemonGame example. The MSC is shown in the figure below.

Verifying a Message Sequence Chart

MSC SystemLevel

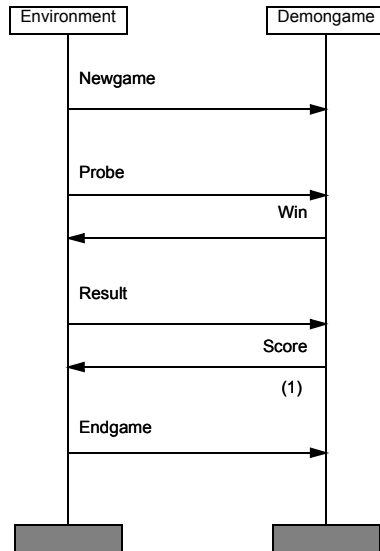


Figure 148: A system level MSC

1. Reset the system. This time do it by choosing *Restart* in the *File* menu. Choose “No” if you are asked to save options. The *Restart* command will actually terminate the running Explorer and start it again.
2. Start an MSC verification by clicking the *Verify MSC* button. A file selection dialog is opened, in which you select the MSC to verify.
3. Select `SystemLevel.msc` and click *OK*. A state space exploration is now started, which is guided by the loaded MSC.

In the printed statistics, note that the exploration is completed without any truncated paths. This is because the loaded MSC restricts the size of the behavior tree; only the parts dealing with the events in the MSC are executed. The maximum depth of it is not more than 20.

Note the line that tells if the MSC was verified or violated:

```
** MSC SystemLevel verified **
```


In this case the MSC was verified, i.e., the behavior described in the MSC was indeed possible. In the Report Viewer, however, one (or two) of the reports is a *violation* of the loaded MSC, while the other one is a verification of the MSC. The exploration may very well find states that violate the MSC; it is the existence of states that verify the MSC that determines the result of the verification.

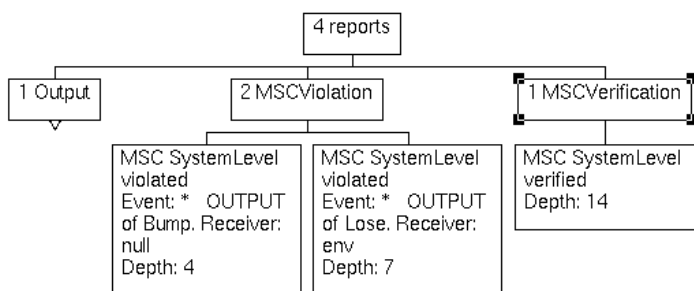


Figure 149: Violations and verifications of the MSC

4. Go to the state where the MSC was verified. The printed trace in the Command window shows that the Main process has received the Endgame signal, and sent the GameOver signal to the Game process:


```

*   OUTPUT of GameOver. Receiver: Game:1
*   Signal GameOver received by Game:1
      
```
5. Take a look at the MSC trace and compare it with the loaded MSC in [Figure 148 on page 217](#). Note that the loaded MSC only defines signals to and from the environment and therefore is less detailed than the MSC trace. An MSC trace in the explorer is always made on the process level.

Verifying a Message Sequence Chart

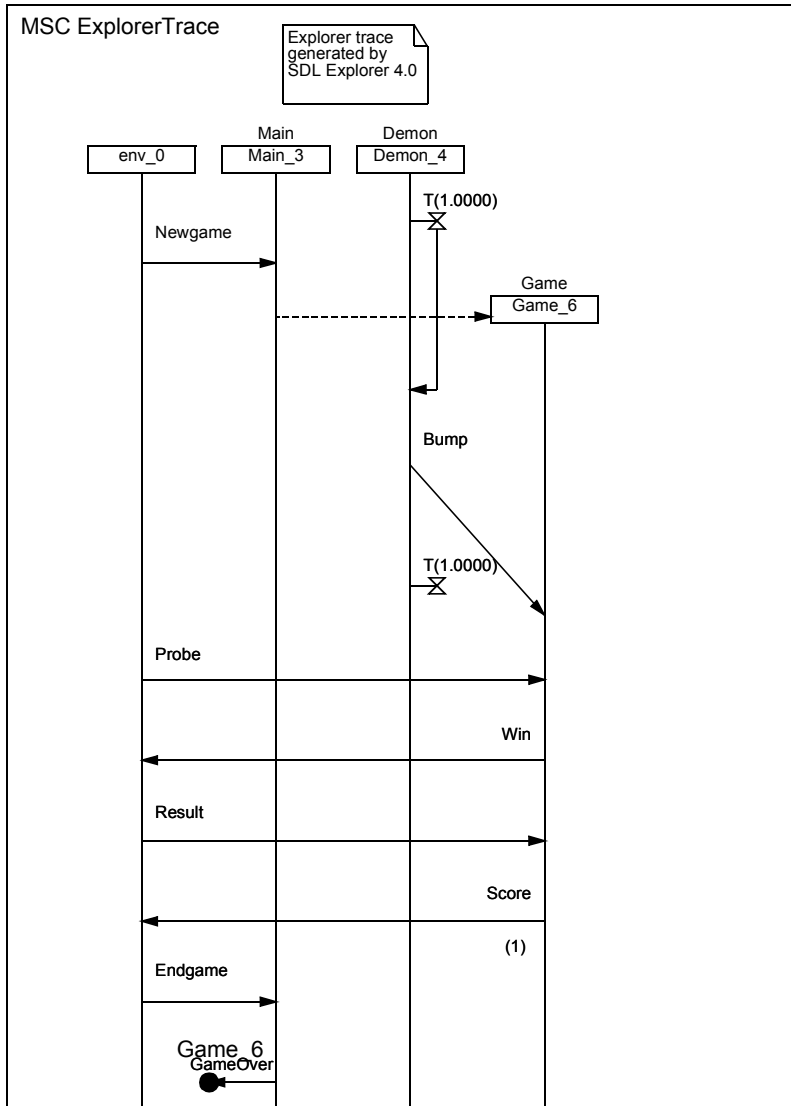


Figure 150: The MSC trace

The trace in the figure does not show the condition symbols that indicates the state of the processes.

Exiting the SDL Explorer UI

The first part of the SDL Explorer tutorial is now finished. Close the explorer windows in the following way:



1. To close the Navigator and the Report Viewer, click the *Close* quick button in these windows.
2. To close the Command window, select *Close* from the *File* menu.
3. Exit the Explorer UI from the *File* menu. You may be asked in a dialog whether to save changes to the Explorer options.

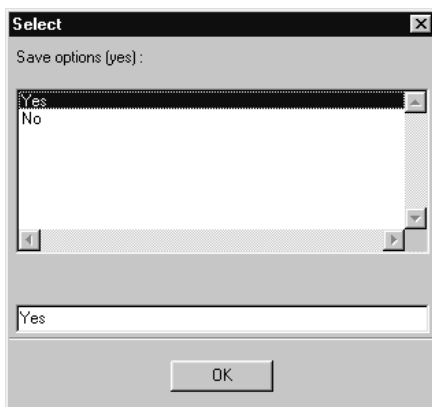


Figure 151: Saving changed options

4. If you select *Yes* and click *OK*, the option settings are saved in a file called **.valinit (on UNIX)**, or **valinit.com (in Windows)**. This file is read each time the Explorer UI is started from the same directory, or when the explorer is restarted or reset from the Explorer UI. You should select *No* and click *OK*.

Using Test Values

In this final exercise we will explore the test value feature in the Explorer. This feature is used to control the way the environment interacts with the system during state space exploration. In practise, the test values define what signals will be sent from the environment to the system, including the exact values of their parameters.

In this part of the Explorer tutorial we will use another SDL system, the Inres system.

1. Copy the Inres system from the installation to a working directory of your own. Copy all files from the directory
`$telelogic/sdt/examples/inres` **(on UNIX)**, or
`C:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\examples\inres` **(in Windows)**.

What You Will Learn

- To examine and use the automatically generated test values
- To manually change the test values

Using the Automatic Test Value Generation

When the Explorer is started, test values for a number of SDL sorts are automatically generated. For example, all integer parameters will have the test values -55, 0 or 55. We will now take a look at the automatically generated test values for the Inres system.

1. Open the Inres system file from the Organizer's *File* menu. If any part of the existing DemonGame system needs saving, you are first prompted to do so before the *Open* file dialog appears. Locate the file `inres.sdt` that you have copied and open it.
2. Generate and start an Explorer for the system by clicking on the *Explore* quick button; see [“Quick Start of an SDL Explorer” on page 190](#) for more information. If you are asked in a dialog whether to start a new Explorer UI or use an existing one, select the existing explorer in the list and click *OK*.
3. Expand the *Test Values* button module in the Explorer UI and make the window bigger so you can see all the buttons.

The button module contains four rows of buttons. The top three buttons *List Value*, *Def Value* And *Clear Value* make it possible to define test values for each sort (data type) in the SDL system. The

middle row with the buttons *List Par*, *Def Par* and *Clear Par* handles test values for specific signal parameters. The bottom rows handles test values for entire signals.

4. Click on the *List Value* button to see what default test values have been generated. The following values should be listed:

```
Sort integer:
0
-55
55

Sort Sequencenumber:
zero
one

Sort IPDUType:
CR
CC
DR
DT
AK
```

As you can see, there were test values defined for the predefined sort *integer* and for two system specific enumerated sorts *Sequencenumber* and *IPDUType*. For enumerated types, all the values will by default be used as test values if there are 10 or less values. Note that only sorts that appear on parameters to signals to or from the environment are listed.

5. Click on the *List Signal* button to see what signals will be sent to the system based on the test values for the sorts.

You should now see a list of signals similar to the following. Note that there might be differences in the parameters to the *MDATreq* signal since this is computed using a random function that is depending on the compiler used.

```
ICONreq
IDATreq(0)
IDATreq(-55)
IDATreq(55)
IDISreq
MDATreq((. CR, zero, -55 .))
MDATreq((. CR, zero, -55 .))
MDATreq((. CC, one, 55 .))
MDATreq((. AK, one, 55 .))
MDATreq((. CR, one, 55 .))
MDATreq((. DT, one, -55 .))
MDATreq((. CC, one, 0 .))
```

```
MDATreq((. CC, one, -55 .))  
MDATreq((. AK, one, 55 .))  
MDATreq((. DR, one, 0 .))
```

The signals *ICONreq* and *IDISreq* have no parameters so there will only be one signal definition for each of these signals. The *IDATreq* signal has one integer parameter, and as you can see there will be three test values for this signal, one for each of the test values for the integer sort.

The *MDATreq* signal takes a parameter that is a structure with three fields: one *IPDUType*, one *Sequencenumber* and one integer. Whenever the Explorer finds a structure, it tries to generate test values for the sort based on all combination of test values for each field. However, if the number of test values is larger than a maximum value, a randomly chosen subset is used instead. The maximum number is by default 10, but can be changed with the *Define-Max-Test-Values* command.

The consequence of this is that for the *MDATreq* signal, 10 different randomly chosen parameter values are generated.

Now, let us check how the test values influence the behavior of the system during state space exploration.

6. Start the Navigator by clicking on the *Navigator* button in the *Explore* button module.
7. Double-click on the down node 4 times (until there is more than one alternative down node).

You should now have a choice between 11 different down nodes that each one represents an input from the environment to the SDL system. If you check the inputs more carefully, you will see that these 11 inputs correspond to the test values defined for the signals *ICONreq* and *MDATreq*.

This is the way the test values have an impact on the state space exploration. Whenever a signal can be sent from the environment to the system, the Explorer uses the test values defined for the signal to determine what parameters to use when sending the signal.

Changing the Test Values Manually

Now, we will use the other commands in the *Test Values* button module to manually change the test values.

1. Click on the *Top* button in the *Explore* module to return to the start state in the state space.

First we will change the test values for integer to only test the values 1 and 99.

2. Click on the *Clear Value* button, select the *integer* type in the dialog and give the value '-' (a dash) in the value dialog. Dash indicates that we would like to remove all test values currently defined for the sort.

Note that the Explorer tries to recompute test values for various sorts and signals when you have changed the test values for integer. Since integer is used in a number of other sorts and signals, the Explorer is now unable to compute test values for these sorts and signals.

3. Check the signal definitions that now is used by clicking on the *List Signal* button. The current signal definitions should now be:

```
ICONreq  
IDISreq
```

Since there are no test values for integer, only the signals that does not contain integer parameters are listed. In this case this means that only ICONreq and IDISreq would have been sent to the system from the environment if you would start an exploration.

4. Click on the *Def Value* button, select *integer* in the sort dialog and give the value 1 in the value dialog.
5. Click on the *Def Value* button once more. Select integer in the sort dialog again, but this time give the value 99 in the value dialog.
6. Click on the *List Signal* button to check the signal definitions and make sure that the signals with integer parameters are once again on the list. This time with the test values 1 and 99.

```
ICONreq
IDATreq(1)
IDATreq(99)
IDISreq
MDATreq((. AK, zero, 1 .))
MDATreq((. DR, zero, 99 .))
MDATreq((. AK, one, 1 .))
MDATreq((. DR, one, 1 .))
MDATreq((. DR, zero, 99 .))
MDATreq((. AK, zero, 1 .))
MDATreq((. AK, one, 99 .))
MDATreq((. AK, zero, 99 .))
MDATreq((. AK, one, 1 .))
MDATreq((. CR, one, 1 .))
```

You have now explored some of the most frequently used test value features in the Explorer. There are also possibilities to set test values for specific parameters and to enumerate all signal definitions manually. You can find more information about this in the section [“Defining Signals from the Environment” on page 2447 in chapter 53, *Validating a System, in the User’s Manual*](#).

Exiting the SDL Explorer

To exit the Explorer follow the same steps as before:

1. Select *Exit* from the *File* menu.
2. Choose *Yes* when asked whether you want to save the new options or not. To select *Yes* in this dialog implies that commands that re-creates your new test value definitions will be saved in the file `.valinit` **(on UNIX)** or `valinit.com` **(in Windows)**.

So Far...

By practicing this and the previous tutorials, you have learned the basics of the SDL Suite and we hope you have enjoyed the “tour”. The examples you have been practising on, the DemonGame and Inres systems, are however rather simple. To deepen your knowledge about the SDL Suite components, you may practise on a number of exercises that illustrate the advantages of SDL-92 when adopting an object-oriented design methodology. These exercises are described in [chapter 6, *Tutorial: Applying SDL-92 to the DemonGame*](#).

Tutorial: Applying SDL-92 to the DemonGame

This tutorial will teach you how to take advantage of the object-oriented extensions that have been added to SDL, also known as SDL-92. The example that has been selected for this purpose is the well known DemonGame, which you should have already practiced on, in the previous tutorials presented in [chapter 3, *Tutorial: The Editors and the Analyzer*](#) and [chapter 4, *Tutorial: The SDL Simulator*](#).

In order to learn how to take advantage of the object oriented extensions in SDL, read through this entire chapter. As you read, you should perform the exercises on your computer system as they are described.

Purpose of This Tutorial

The purpose of this tutorial is to make you familiar with the essential object-oriented SDL functionality in the SDL Suite tools. This tutorial is designed as a guided tour through the SDL Suite, where a number of hands-on exercises should be performed on your computer as you read this chapter.

We have on purpose selected a simple example that should be easy to understand. It is assumed that you have a basic knowledge about SDL — this chapter is **not** a tutorial on SDL.

The example is DemonGame, which was used in the earlier tutorials in this volume. It is assumed that you have performed the exercises in [chapter 3, *Tutorial: The Editors and the Analyzer*](#) as well as [chapter 4, *Tutorial: The SDL Simulator*](#) before starting with this tutorial.

Note: Platform differences

This tutorial, and the others that are possible to run on both the UNIX and Windows platform, are described in a way common to both platforms. In case there are differences between the platforms, this is indicated by texts like “on UNIX”, “Windows only”, etc. When such platform indicators are found, please pay attention only to the instructions for the platform you are running on.

Normally, screen shots will only be shown for one of the platforms, provided they contain the same information for both platforms. This means that **the layout and appearance of screen shots may differ** slightly from what you see when running the SDL Suite in your environment. Only if a screen shot differ in an important aspect between the platforms will two separate screen shots be shown.

Applying SDL-92 to the DemonGame

In the previous tutorials, you have practiced using some of the basic language elements in SDL; all of these elements were already defined in the non-object-oriented version of SDL, known as SDL-88.

To introduce SDL-92 to you, we have prepared a number of exercises in which you will add features to the DemonGame. You will do this by redefining and adding properties to the process Game in an object-oriented fashion.

We will introduce the following SDL-92 language constructs:

- Process types
 - Inheriting process types and adding properties
 - Virtual and redefined process types
 - Virtual and redefined transitions
- Packages
 - Using packages
 - Reusing packages
- Block types
 - Inheriting block types and adding properties.

Note:

In this chapter, the term SDL-92 denotes the object-oriented SDL that was introduced in the 1992 version of the language. These object-oriented features remain unchanged in SDL-96.

Some Preparatory Work

Instead of continue working on the original DemonGame system, we suggest you to continue from a version that is better designed for introducing SDL-92. The changes that have been made are the following:

- All signals from the environment (Newgame, Endgame, Probe, Result) are now directed to the administrating process Main, that will send them further to the Game process, if there is such a process.
- The Bump signal is also sent to the process Main, which in turn transfers it to the Game process. This eliminates the annoying behavior when a signal is sent to a nonexisting receiver.

- Signal routes and signal lists have been updated to reflect the new routing of signals.
- The internal signal `GameOver` is really not necessary and is therefore replaced by the signal `EndGame`.

From the user's point of view, the system will show the same functionality as before, but is more robust.

The new versions of the block `GameBlock` and the process `Main` are depicted below, in [Figure 152](#) and [Figure 153](#).

To use the new version:

1. Make a new empty directory `sd192` of your own (under `~/demongame` **on UNIX**, and under `C:\IBM\Rational\SDL_TTCN_Suite6.3\work` **in Windows**).
2. Copy all files in the directory `$telelogic/sdt/examples/demongame/sd192/process_type` **(on UNIX)**, or `C:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\examples\demongame\sd192\process_type` **(in Windows)**, to this new directory.

Note: Installation directory

On UNIX, the installation directory is pointed out by the environment variable `$telelogic`. If this variable is not set in your UNIX environment, you should ask your system manager or the person responsible for the SDL Suite environment at your site for instructions on how to set this variable correctly.

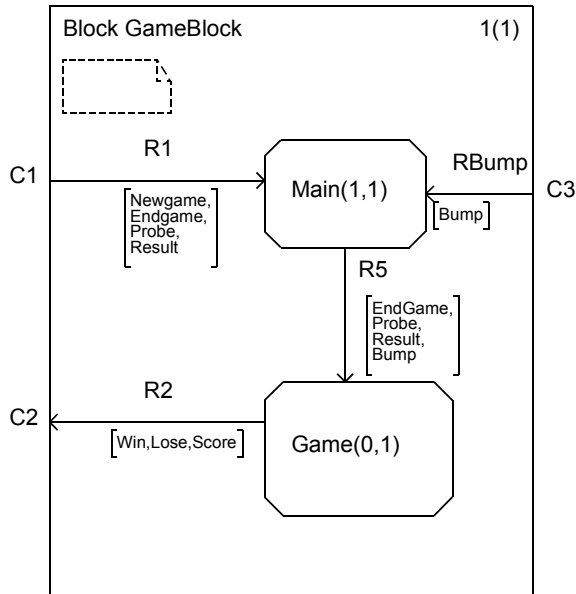
In Windows, the installation directory is assumed to be `C:\IBM\Rational\SDL_TTCN_Suite6.3` throughout this tutorial. If you cannot find this directory on your PC, you should ask your system manager or the person responsible for the SDL Suite environment at your site for the correct path to the installation directory.

3. Start the SDL Suite and open the system file `demongame.sdt` in this new directory with the Organizer. (You will find copies of the diagrams building up the complete system).

You should recognize the system `DemonGame`, with the modifications as described above.

Some Preparatory Work

Nearly all versions of the diagrams shown in the following exercises are available in the directory you created above. You can either draw a diagram to learn how to use SDL-92 in the SDL Editor, or copy (or connect to) the pre-made version of the diagram if you do not wish to do this.



*Figure 152: The block GameBlock, redesigned
The exact layout of your diagrams may differ slightly from the above.*

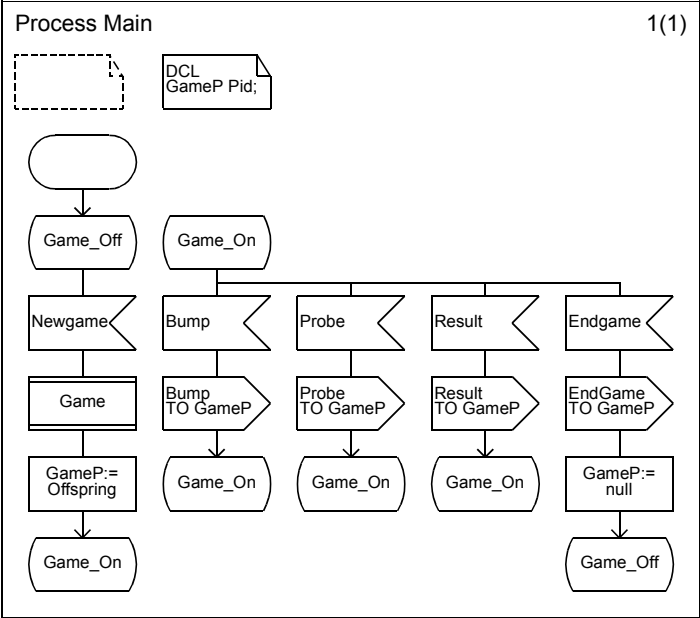


Figure 153: The process Main, redesigned
The exact layout of your diagrams may differ slightly from the above.

Creating a Process Type from a Process

What You Will Learn

- To change a process diagram to a process type
- To refer to and instantiate a process type
- To interconnect the process type with a block and other processes (types), using gates
- To define transitions as virtual

Changing into a Process Type

To facilitate the introduction of new features, we will start by generalizing the process Game, by changing it to a process type, that you later on will be in a position to specialize or redefine.

1. Open the process Game and change the diagram type from process to process type, simply by selecting the diagram heading symbol and editing the text in it to say “Process Type Game”.

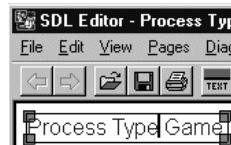


Figure 154: Changing the diagram type

2. From the SDL Editor's *File* menu, save the diagram Process Type Game **on a new file**, e.g. `new_game.spt` using the *Save As* command. Edit the file name in the file selection dialog and click *OK*. (The existing `*.spt` files are copies of the complete system that comes with the examples in the SDL Suite.)
3. Raise the Organizer window.

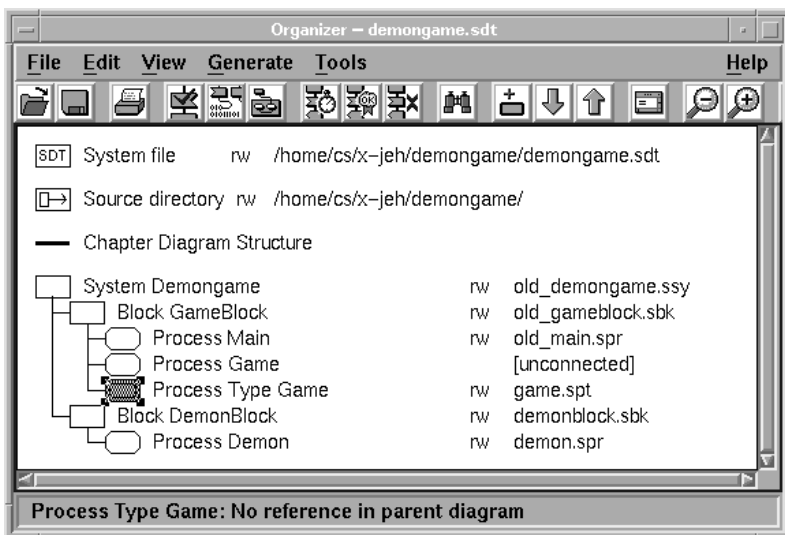


Figure 155: Invalid reference as shown in the Organizer

You may notice that the reference symbol has been changed to Process Type Game, and marked as having no reference in the parent diagram. Also, the old process Game is marked as *unconnected*. Do not bother about that for the moment – it will be replaced by an instantiation symbol, which will be explained later.

4. Open the diagram block GameBlock. Change it so that the process reference Game is changed to an instantiation of the process type Game. The syntax is: “Game(0,1):Game” (You are allowed to add newlines to have the text fit into the symbol.)
 - Before you have started text editing, the text cursor is not flashing. Pressing <Delete> at this stage deletes the whole selected symbol. Once text editing has started, the text cursor is flashing and pressing <Delete> only deletes a character.
5. As soon as you deselect the symbol, one text rectangle appears for each connection point to the signal routes. Name the connections points for instance G2 and G5:

Creating a Process Type from a Process

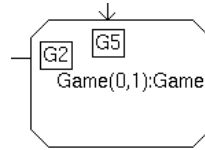


Figure 156: Naming the connection points

6. Also add a process type reference symbol with the name Game. The block diagram should now look like this:

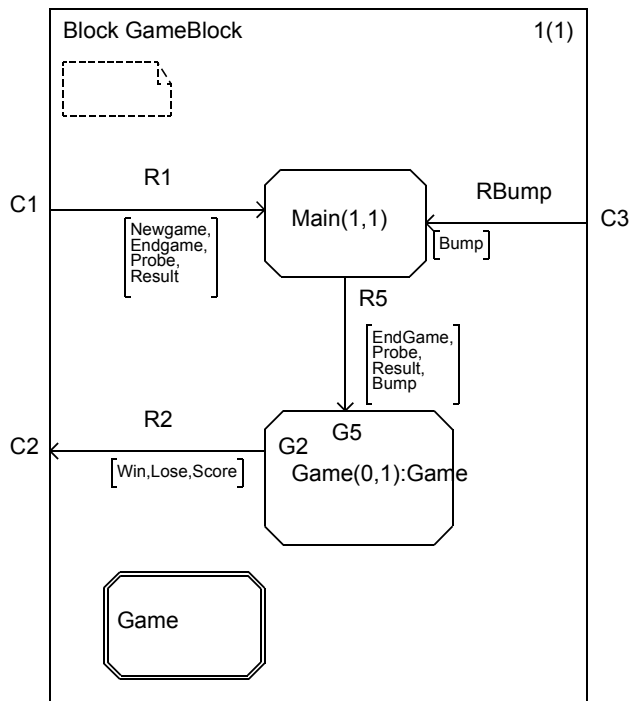


Figure 157: The resulting block

7. Save the block diagram on a new file, e.g. new_gameblock.sbk (use *Save As* as before).

Inserting Gates and Virtual Transitions

You will now finish the process type Game. You are recommended to do this by following the editing instructions described below. If you prefer, you can instead connect to the finished version of the diagram (see [“Connecting to the Finished Diagram” on page 238](#)), but you should in any case read through the text below.

Editing the Process Type Diagram

1. Go back to the process type Game in the SDL Editor. The connection to the signal routes must be defined using gate symbols, named in accordance to the connection points you just defined.
2. Gate symbols are to be connected to the frame symbol. If you want to connect gates to the left or top of the frame, you must first select the frame and drag it down and/or right.

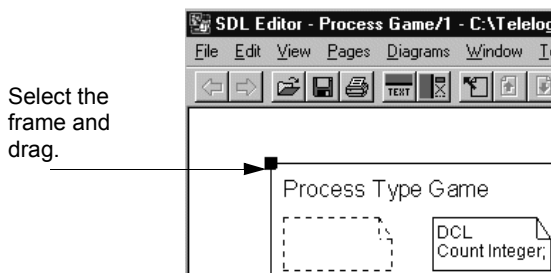


Figure 158: Adjusting the frame symbol

3. For each of the gates G2 and G5, add a gate symbol and fill in the name and the signal list.
 - The gate symbol is the one who looks like an arrow. Remember that the Status Bar displays the type of a symbol when you point to or select it in the symbol menu.
 - To direct a gate **to** the frame, you must add a gate, then *Redirect* it. (Gates can also be made bidirectional.)
 - You may use the *New Window* command to bring both the Process Type Game and the Block GameBlock into view at the same time, then *Copy* and *Paste* the text between the diagrams.
 - You may also take advantage of the *Signal Dictionary* window, and *Insert* the signals from the block GameBlock (*Up*).
4. Also make the start transition as well as the input of the signals Probe and Bump virtual by adding the text “VIRTUAL” before the name of the signal.
 - By doing this, you will later on be able to change the properties of the game in a smooth way.

The changes to the resulting diagram should now look something like this:

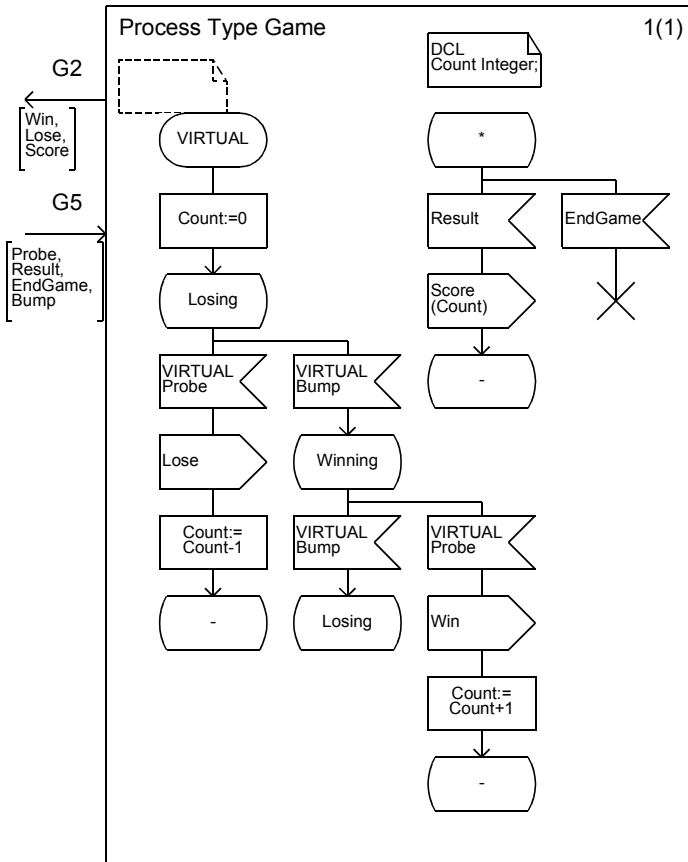


Figure 159: The resulting process type Game

Connecting to the Finished Diagram

The finished Game process type diagram is also available as the file `game.spt`. If you instead of drawing the diagram wish to use this file, do as follows:

1. In the SDL Editor, close the Game process type diagram.
2. In the Organizer, select the diagram Process Type Game, and then select *Connect* from the *Edit* menu.

Creating a Process Type from a Process

3. Select the option *To an existing file*. Change the filename to `game.spt`, or select this file by using the folder button.
4. Click *Connect* and check the new file connection in the Organizer.

The Organizer Structure

1. Save everything. The resulting Organizer list should now resemble:

— Chapter Diagram Structure

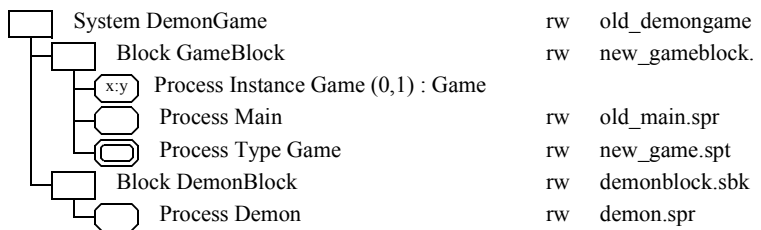


Figure 160: Resulting Organizer view

Note the presence of an *instantiation symbol*, looking like a normal symbol, but with the generic “X:Y” (meaning instance:type) notation in it. The instantiation symbol denotes that the type is actually instantiated somewhere in the diagram.

Instantiation symbols in the Organizer **cannot** be used for navigating into the system hierarchy with a double click, since they do not refer to diagrams. (You can use them from within the SDL Editor with the support from the *Type Viewer* tool, which you will practice on later in this tutorial).

2. Terminate by analyzing the system. Correct any syntactic or semantic errors that are reported.

Redefining the Properties of a Process Type

What You Will Learn

- To have a process type inherit properties from another process type
- To redefine transitions in a process type

The Process Type JackpotGame

So far, you have redesigned the original functionality of the system DemonGame, using a slightly different design. Next step will be to add a feature that allows you to win the “jackpot”, with a probability of 10%. The jackpot is arbitrarily set to increase the score by 10. A simple implementation of this could be to create a pseudo random number generator that returns a sequence of numbers from 0 to 9, and to check the random number upon the reception of the signal Probe.

It should also be possible to specify what kind of game to start at runtime, meaning that we need an additional input signal from the environment, NewJackpotGame, that will start the JackpotGame; that new signal requires additions to the process Main and the system diagram.

The JackpotGame is implemented as a process type that inherits the properties of the process type Game, and adds the random number feature by redefining the transitions that handle the signal Bump. The pseudo random generator is activated upon each reception of the signal Bump. See below.

Redefining the Properties of a Process Type

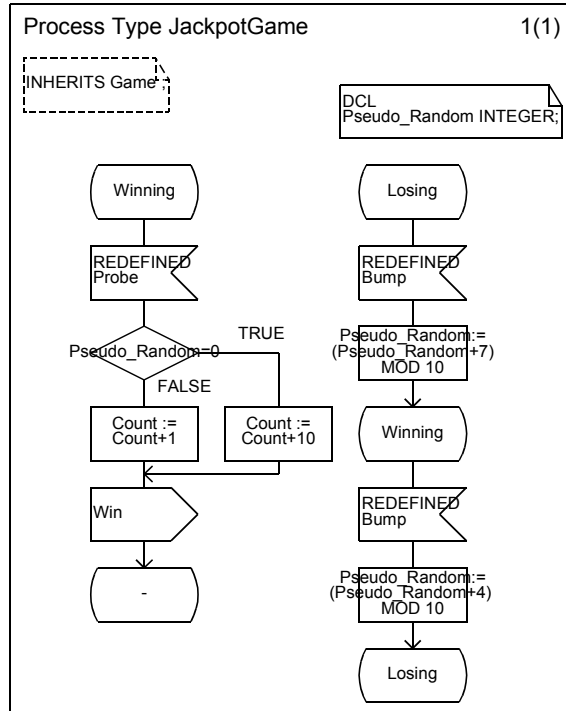


Figure 161: The process type *JackpotGame*

- Create the diagram above and save it on a new file (new_jackpotgame.spt for instance). Then close the diagram in the SDL Editor (*Close Diagram* from the *File* menu).
 - This diagram is also available as the file `jackpotgame.spt`, if you wish to make a copy (or use it as is) instead of drawing the diagram.

Changes to the Block GameBlock

To make the process type `JackpotGame` available from the parent block, you simply add a process reference symbol and a process instantiation symbol, as you did before with the process type `Game`. You also add a signal `NewJackpotGame` to the signal list to the process `Main`.

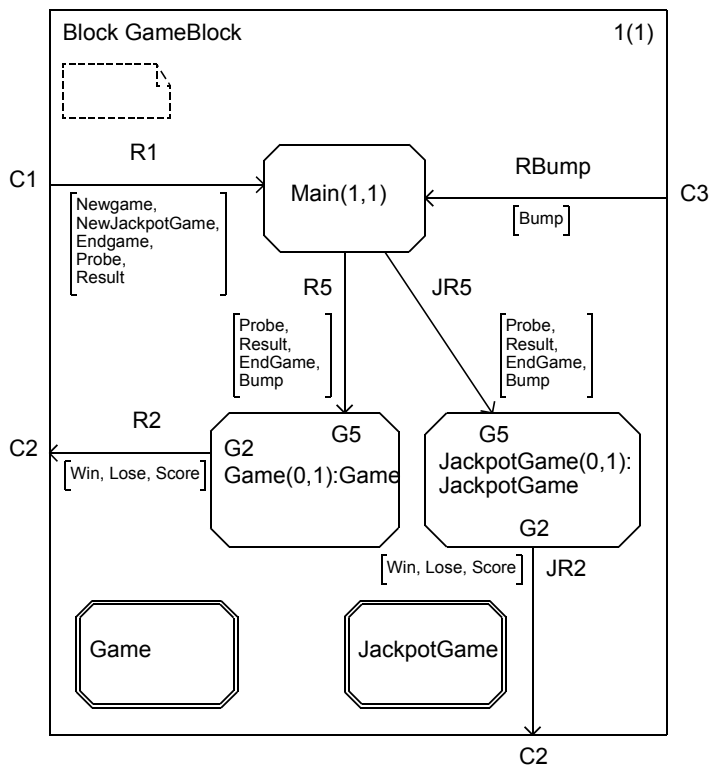


Figure 162: The block `GameBlock`

- Update the existing diagram `GameBlock` according to above and save it on file.
 - This version of the diagram is also available as the file `gameblock2.sbk`. To use it instead of drawing the diagram, close the `GameBlock` diagram in the SDL Editor, and connect the `GameBlock` diagram in the Organizer to the new file. (Use *Connect* in the *Edit* menu and the option *To an existing file*.)

Changes to Process Main and System DemonGame

The process Main and the system DemonGame need to be extended with the declaration of the signal NewJackpotGame and the code to receive the signal and create an instance of the game JackpotGame:

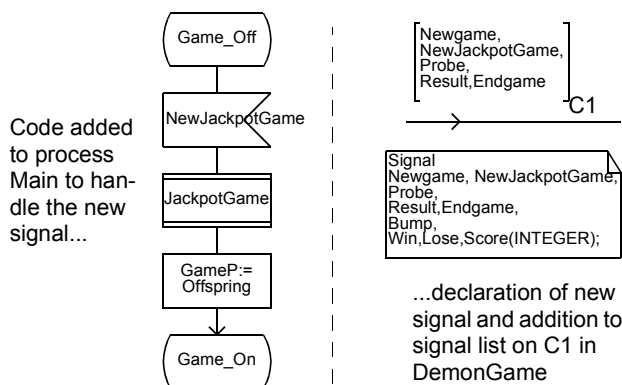


Figure 163: The extensions to process Main and system DemonGame

1. Update the diagrams Main and DemonGame according to the figure above and save them on file. You may want to save the diagrams on new files, e.g. `new_demongame.ssy` and `new_main.spr`.
 - This version of the Main diagram is also available as the file `main2.spr`, if you wish use it instead of editing the diagram. In the Organizer, connect the diagram to the new file (from the *Edit* menu).
 - **The DemonGame diagram has to be edited manually** – do not re-connect it to an existing file.
2. In the Organizer, make sure that the process type diagram JackpotGame is connected to the file `new_jackpotgame.spt` that you created earlier. (If not, use *Connect* in the *Edit* menu and the option *To an existing file*.)

The resulting Organizer list should now look like this:

— Chapter Diagram Structure

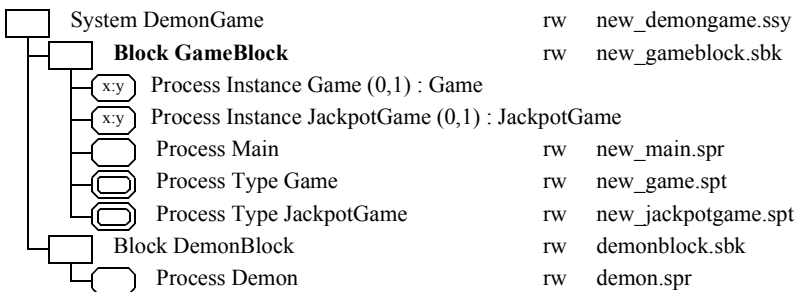


Figure 164: JackpotGame added to Organizer list

Simulating the JackpotGame

To understand the resulting system, you may want to spend a few minutes simulating it.

1. First analyze the system and generate a simulator, as you learned from the tutorial on the simulator. Then open the generated simulator in the Simulator UI.
2. We suggest that you check the following features:
 - It should be possible to start one instance of Game **or** of JackpotGame at run-time using the NewGame/NewJackpotGame signals, but not to have two games running at the same time. (Use the command **output-via** to send the signals NewJackpotGame, Newgame and EndGame via C1, in order to start and stop the game).
 - Even if we do not have any game started, the signal Bump no longer causes any dynamic error, since there is always a receiver (Main).
 - Turn the graphical MSC trace on, to visualize how the signalling is done. Also turn the graphical SDL trace on. Verify that the execution takes place in the graphs for both the process types Game and JackpotGame, even if you have started a JackpotGame! (You may have to execute at symbol level to catch this.)

Redefining the Properties of a Process Type

3. Play the game in a realistic way.
 - First, create a button in the Simulator UI with the name *Probe*, that sends the signals *Probe* and then *Result*, then resumes the execution with the command *Go*. Each time you click this button, the *Score* is returned. (The button definition should contain `output-to Probe Main; output-to Result Main; go`)

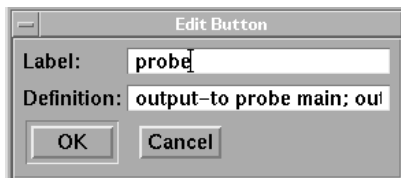


Figure 165: Defining the button *Probe*

- Then, set the trace for the system to 1, meaning that only signals to/from the environment are traced. If required, turn all graphical traces off, in order to speed up the execution:
`set-gr-trace 0; stop-msc-log`
 - Send the signal *NewJackpotGame* and run the simulator:
`output-to NewJackpotGame Main; go`
 - Click repeatedly the *Probe* button and watch the trace. You should win 10 points every now and then.
4. Stop the execution with the *Break* button.

Adding Properties to a Process Type

What You Will Learn

- To inherit a process type and add properties
- To use dashed gates.

The Process Type DoubleGame

Even with a “jackpot” feature, winning “a lot” with the DemonGame takes some time... Suppose now that you would like to add a function that doubles the “stake” of the game, whenever you want, so that you have the possibility to win more.

A way to do this is to:

1. Create a process type DoubleGame, that inherits the properties of the process type Game, with the following additions:
 - Declaration of a variable Stake of type integer.
 - Initialization of Stake to 1, by redefining the start transition.
 - Reception of a signal DoubleStake that doubles the value of Stake.
 - Redefinition of the transitions Winning and Losing to add/deduct the current Stake from the score Count.
2. The resulting graph is depicted below. Create it in the same way as you have learned from the previous exercises, and save it on the file `new_doublegame.spt`. Then close the diagram in the SDL Editor (*Close Diagram* from the *File* menu).
 - This diagram is also available as the file `double.spt`, if you wish to make a copy (or use it as is) instead of drawing the diagram.

Adding Properties to a Process Type

Note:

The diagram contains a *dashed* gate symbol G5, where the signal DoubleStake is conveyed. You use dashed gates to refer to gates that are already defined in the supertype (the type that you inherit from), to distinguish from situations where you have to add a new gate.

To dash a gate:

- Make sure the gate is selected.
- Select the *Dash* command from the *Edit* menu of the SDL Editor (this command toggles between *Dash/Undash*).

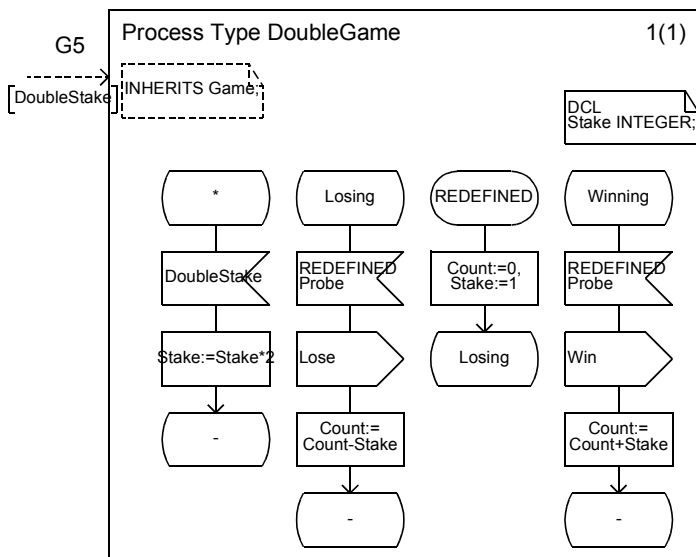


Figure 166: The process type DoubleGame

3. Add a process type reference symbol DoubleGame, and a process instantiation symbol with the text “DoubleGame (0,1):Double-Game” to the block diagram GameBlock; see below.
 - This version of the diagram is also available as the file `gameblock3.sbk`. To use it instead of drawing the diagram, close the GameBlock diagram in the SDL Editor, and connect the GameBlock diagram in the Organizer to the new file.

4. Also add the signals NewDoubleGame and DoubleStake to the signal list on the signal route R1, and DoubleStake to the signal list on the signal route R1, and DoubleStake to the signal list on the process DoubleGame; see below.

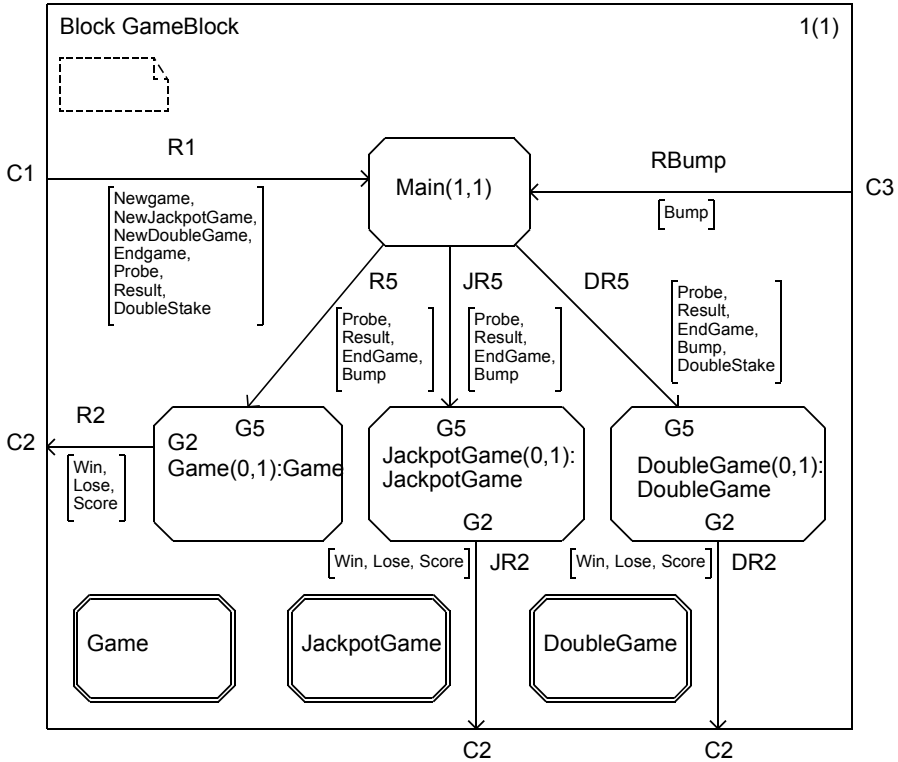


Figure 167: The resulting block GameBlock

5. Add the new signals NewDoubleGame and DoubleStake at the system level (in the DemonGame diagram), both in the signal declaration, and in the signal list on the channel C1.

– **You have to make these changes yourself.**

Adding Properties to a Process Type

6. Extend the process Main with the code to receive the signal DoubleStake, and the code to receive the signal NewDoubleGame and create an instance of the game DoubleGame; see below.
 - This version of the Main diagram is also available as the file `main3.spr`, if you wish use it instead of editing the diagram. In the Organizer, connect the diagram to this file.

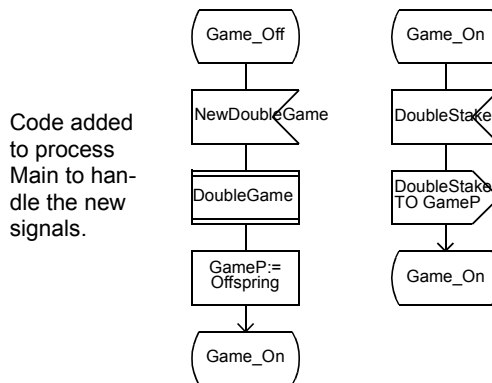


Figure 168: New code in process Main

7. If needed in the Organizer, connect the process type diagram DoubleGame to the file `new_doublegame.spt` that you created earlier.

Simulating the DoubleGame

You may simulate the DoubleGame in a similar way as the JackpotGame (the DoubleGame is started with the signal NewDoubleGame).

1. To play the game in a realistic way, also add a button *Double* to the Simulator UI, with the text “Double” and the command `Output-to DoubleStake Main; go`
2. Try for instance the following tactic: whenever your score is negative, double the stake.

Combining the Properties of Two Process Types

What You Will Learn

- To work with the *Type Viewer* (the “class browser” in the SDL Suite)
- To inherit process types in more than one level

So far, you have created a basic version of the game (the supertype process type *Game*), and extended it as two subtypes (the process types *JackpotGame* and *DoubleGame*). To assist you in understanding the inheritance and instantiation of types, the SDL Suite is provided with a “class browser”, the *Type Viewer*.

Working with the Type Viewer

1. Open the *Type Viewer* with the command *Type Viewer* from the Organizer’s *Tools > SDL* menu.

The *Type Viewer* is started and displays two windows: the main window, where all types are listed, and the *Type Trees*, where the inheritance and instantiation of the types is visualized.

Combining the Properties of Two Process Types

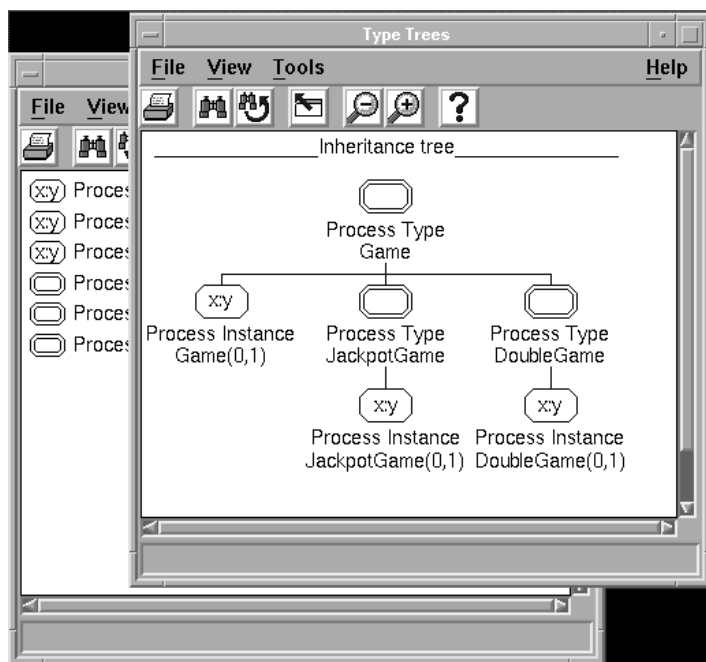


Figure 169: The two windows of the Type Viewer

The main window displays a list of all types and instances that exist in your current system. When selecting an object in the main window, the Type Trees window is updated to show the inheritance tree for that type.

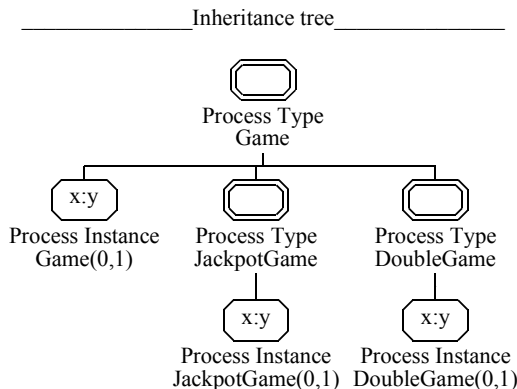


Figure 170: The inheritance tree for the process type Game

Figure 170 shows an inheritance tree for the process types Game, JackpotGame and DoubleGame. We have one level of inheritance, as depicted above. You can also note that the Type Viewer keeps track about the types that have been instantiated somewhere in the SDL system.

- You may go to the source SDL graphs and find the declarations and instantiations of the types by double clicking the symbols in the Type Viewer.

How to Work-Around the Lack for Multiple Inheritance

Say that you would like to design a new game where both the “jackpot” and the “double” features are supported. As SDL-92 does not support multiple inheritance, we cannot simply create a SuperGame that inherits JackpotGame and DoubleGame. Instead, we will have to inherit from, i.e. reuse, the JackpotGame or the DoubleGame, and then redefine/add some of the properties. (The idea is to rewrite as little code as possible).

Which one should we reuse as is? The code for the DoubleGame seems to be still valid for the SuperGame. So, let us inherit that process type, and redefine some of the properties in accordance to the JackpotGame.

Combining the Properties of Two Process Types

To create the SuperGame:

1. Open the process type JackpotGame in the SDL Editor, and *Save As* on a new file, e.g. `new_supergame.spt`
 - This diagram is also available as the file `supergame.spt`, if you wish to make a copy (or use it as is) instead of drawing the diagram. In that case, continue with step 6. below.
2. Rename the diagram to process type SuperGame.
3. Change the inheritance from “INHERITS Game” to “INHERITS DoubleGame”.
4. Update the contents of the graph, in order to:
 - Change the branch Winning/Probe so that you add Stake instead of 1 to Count when winning, and reward you with 10 times the value of Stake when winning the jackpot.
 - Redefine the transition Losing/Probe so that you deduct Stake instead of 1 from Count.

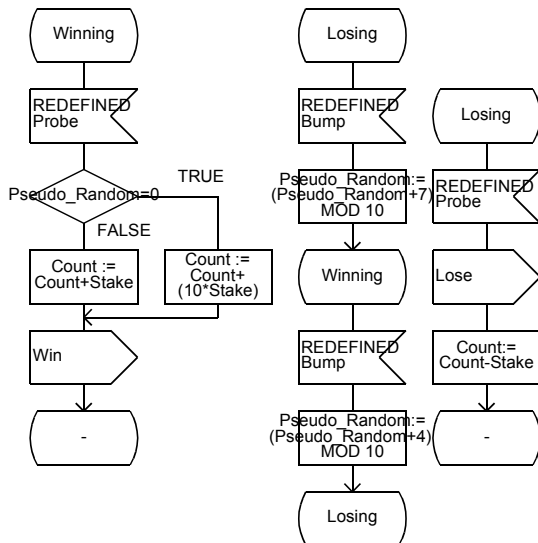


Figure 171: The changes to the process type SuperGame

5. If the Process Type JackpotGame has become unconnected in the Organizer, re-connect it to the file used earlier (`new_jackpotgame.spt`).
6. Also add a process type reference symbol with the name SuperGame in the diagram GameBlock. In the Organizer, then connect the newly added process type diagram SuperGame to the file `new_supergame.spt` that you created earlier.
7. You may check the impact of the changes above in the Type Viewer. Save everything and then select *Update* from the Type Viewer's *File* menu (since the Type Viewer does not automatically update its content when you make changes to a diagram). Your inheritance tree should now look like this:

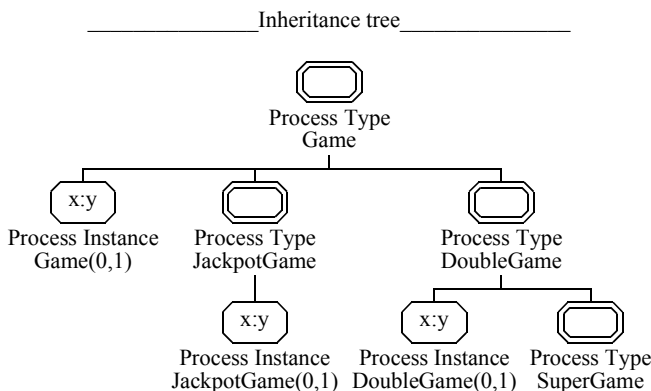


Figure 172: The process type SuperGame, added

8. If you want to be able to play the SuperGame, you must also add a process instantiation symbol “SuperGame(0,1):SuperGame” in the GameBlock, and add a signal NewSuperGame that starts the game (in a similar fashion as you did in the JackpotGame and the DoubleGame). Do not forget to update the system diagram.
 - These versions of the diagrams are also available as the files `gameblock.sbk` and `demongame.ssy`, if you wish use them instead of editing the diagrams. In the Organizer, connect the diagrams to the new files. A complete and final system file, `demongame_sdl92.sdt`, is also available, with connections to the final SDL diagrams.

Using Packages and Block Types

What You Will Learn

- To create a package diagram
- To use a package in a system
- To refer to and instantiate a block type
- To define a process type as virtual

Package – a Reusable Component

Packages are used to make type definitions available in different systems, and to make components reusable. You will take advantage of the package concept by developing two versions of the DemonGame, one that has only the basic “Probe” feature, and one that also includes the “Jackpot” and “DoubleStake” features.

The idea here is to develop a package “BasicFeatures” that is used in the basic version and that is reusable to 100% in the advanced version.

Using packages to their full extent in this example requires not only the process Game to be transformed to a process type (as you have done in the previous exercises, when creating the JackpotGame, DoubleGame and SuperGame), but also to transform the process Main and the block GameBlock to reusable process type and block type, respectively.

You have probably noticed that the process type Main also requires to be extended for each feature that we add (“jackpot”, “double”, etc.), so it would be a good idea to make a reusable type of it. This has already been prepared for you, so your task will be to add the required “glue” to build the two packages.

1. Start by copying the files `gameblock.sbt` and `main.spt` from the directory
`$telelogic/sdt/examples/demongame/sdl92/packages` (**on UNIX**), or
`C:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\examples\demon game\sdl92\packages` (**in Windows**) to the same directory you created earlier for this tutorial.
 - All diagrams in the remaining exercises are available in the above directory. You may choose to copy them if you do not

want to draw all diagrams, and then connect the created diagram symbols in the Organizer to the corresponding files.

Creating a Package

To create a package:

1. Select the *Add New* command from the Organizer's *Edit* menu. In the *Add New* dialog, specify document type as **SDL Package**, and document name as **BasicFeatures**.

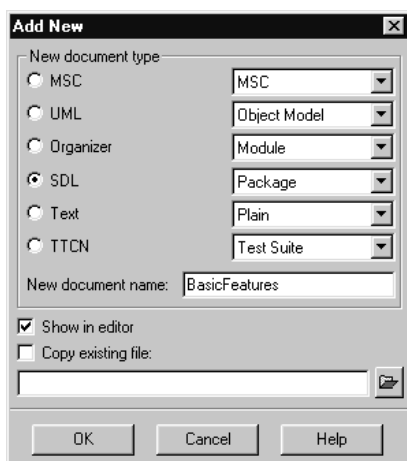


Figure 173: Adding a new package *Basic Features*

As you click *OK*, a new diagram structure is created in the Organizer with the package diagram as root diagram. (The Organizer supports managing multiple structures in the same system file.)

The newly created package should contain the generic properties for the DemonGame; namely:

- The declaration of the signal interface between the “basic” DemonGame and the environment, as well as a process type *Main* that supports the signal interface.
- The definition of the process type *Game* with the basic functionality.
- A block type that contains the process types.

2. With the SDL Editor, move the declaration of the signals from the system diagram to the package diagram. Also add the process type reference symbol Game and the block type reference symbol BasicGameBlock to the package diagram. See below.

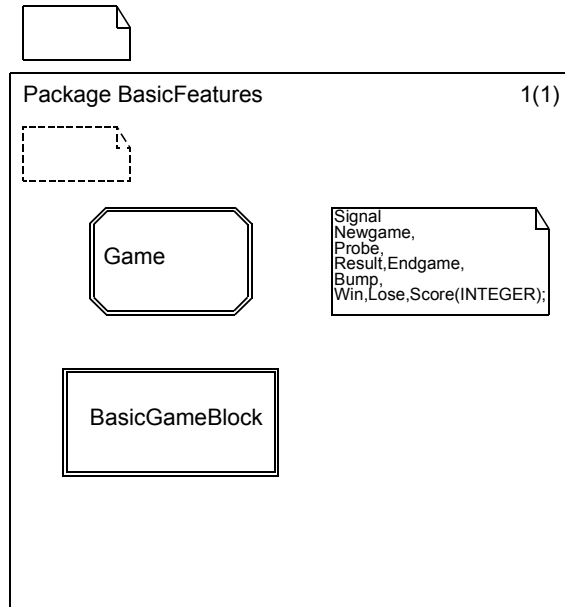


Figure 174: The package BasicFeatures

3. Save the package diagram on a file, e.g. `basicfeatures.sun`
4. With the Organizer, connect the block type BasicGameBlock to the recently copied file `gameblock.sbt` (the diagram is depicted in [Figure 175](#)).
 - Note that the process type Main is declared as VIRTUAL. This is essential since we are going to add properties to Main, and need to address signals from the environment **to Main**, without changing its name to e.g. “SuperMain” (compare to how you did for the process type Game that was specialized into JackpotGame, etc.). By defining a process type as VIRTUAL, we can later add properties without changing its name, using the

keyword REDEFINED (you will practice that in a few moments, in [“Redefined Process Type Main” on page 263](#)).

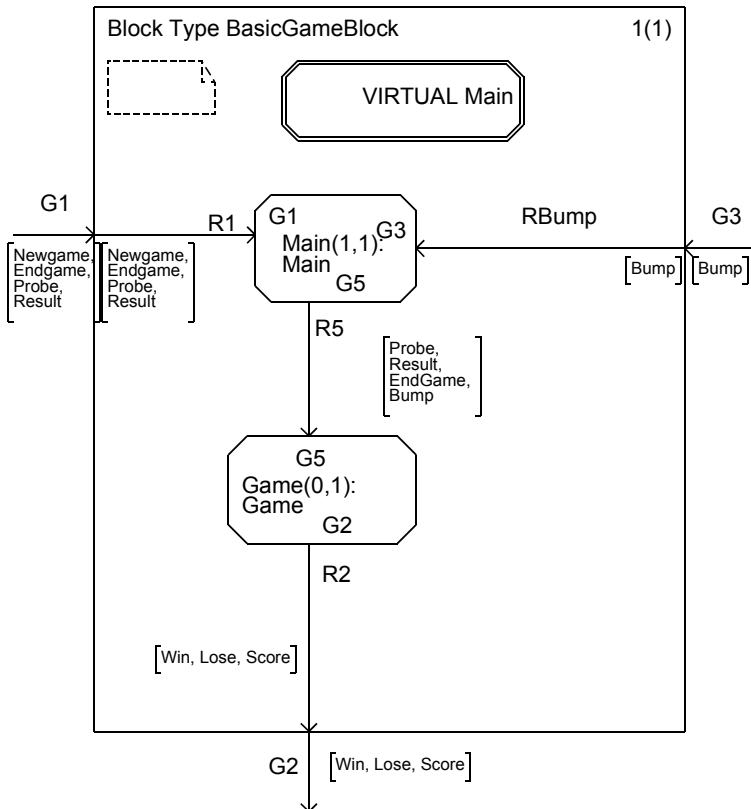


Figure 175: The block type BasicGameBlock

5. You should also connect the process type Main to the copied file `main.spt` (This may already have been done if you had *Expand Substructure* turned on in the *Connect* dialog in the previous step.)

Using a Package

To use a package, you add a USE statement to the package reference symbol (looking like a text symbol immediately outside the frame symbol).

1. To create a version of the DemonGame that has the basic features, add a USE statement to the system diagram. Also remember to instantiate the block type BasicGameBlock (which is contained in the package). See below.

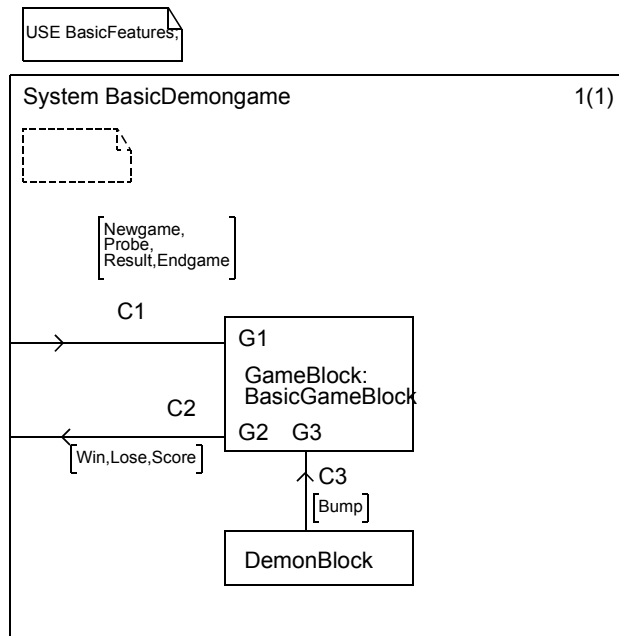


Figure 176: USE of packages

- You may want to save the system diagram on a new file, e.g. `basicdemongame.ssy`
2. In the Organizer, *Disconnect* the old GameBlock from the system structure. The resulting Organizer view should now be something like this (the order of appearance of symbols may differ):

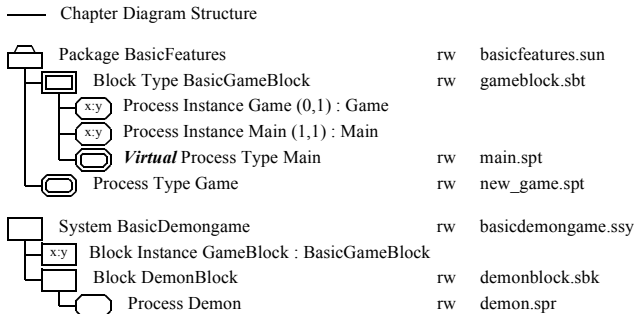


Figure 177: System BasicDemongame using package BasicFeatures

3. Terminate the exercise by analyzing the resulting system.

Note:

You may analyze the package (by selecting the package symbol in the Organizer as input to the Analyzer before ordering the *Analyze* command), in which case a **partial** semantic analysis will be done on the package. (The Analyzer will not check the consistency between the package and the system that uses it.)

A complete semantic analysis requires the system diagram to be selected before ordering *Analyze*.

Reusing Packages

When developing the version of the DemonGame that has all features, you create a package AdvancedFeatures that contains the additional features and that will reuse the package BasicFeatures.

What You will Learn

- To reuse a package in another package
- To inherit a block type
- To redefine a process type

The Package AdvancedFeatures

1. Create the package AdvancedFeatures in a similar fashion as the package BasicFeatures (see [Figure 173 on page 256](#)).
 - You should now have two package structures in the Organizer, BasicFeatures and AdvancedFeatures.

The package AdvancedFeatures must do the following:

2. Use the package BasicFeatures.
3. Add the declarations of the new signals.
4. Add references to the process types JackpotGame, etc.
5. Add a reference to a block type AdvancedGameBlock (which inherits the block type BasicGameBlock and in turn refers to a redefined process type Main).
6. Save the package diagram on the file `advancedfeatures.sun`

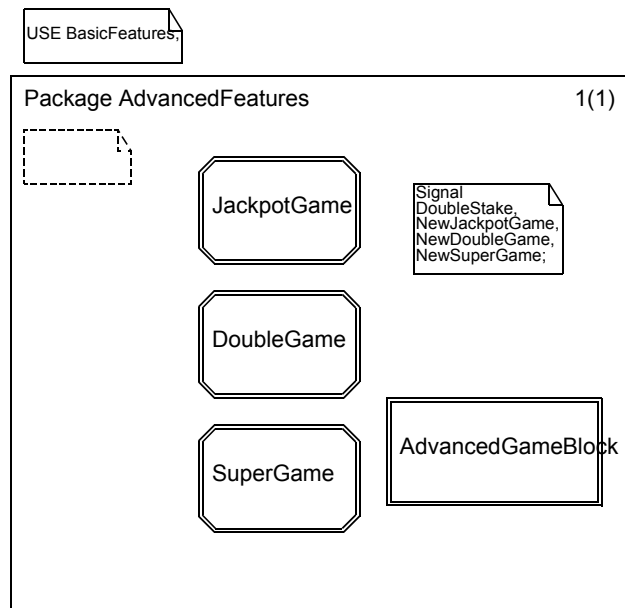


Figure 178: The package AdvancedFeatures

Block Type AdvancedGameBlock

The diagram contains a reference to a REDEFINED process type Main, and a dashed instantiation symbol Main.

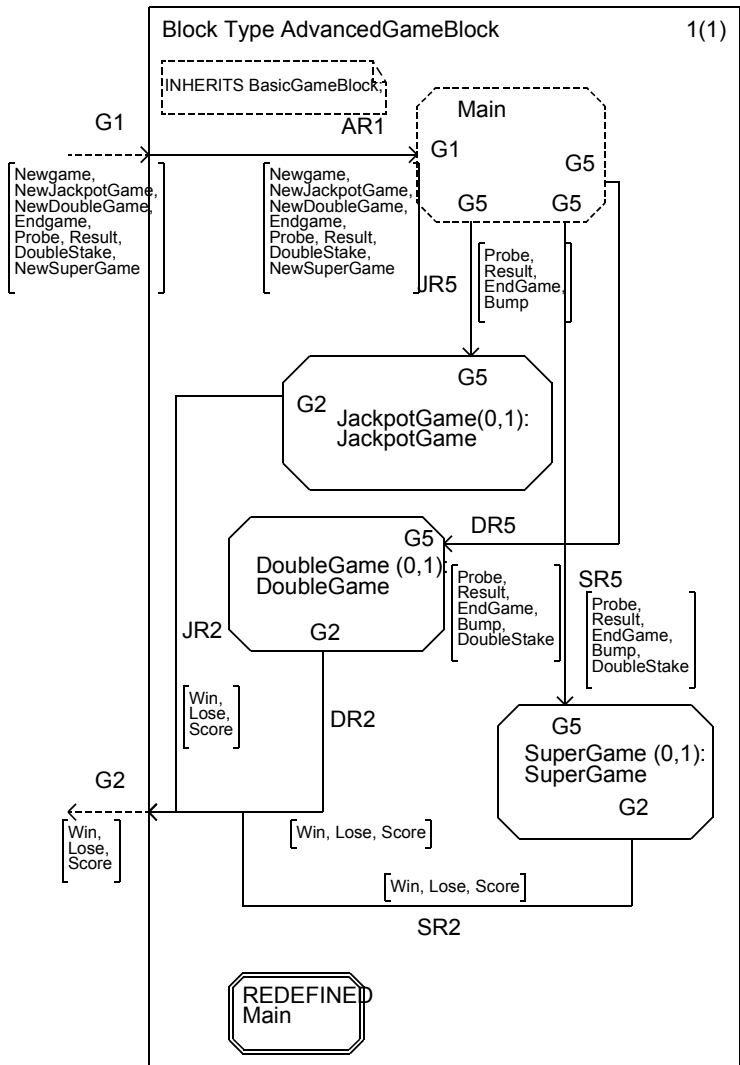


Figure 179: Block type AdvancedGameBlock

- The block type `AdvancedGameBlock` is already provided on the file `advancedgameblock.sbt`. Copy that file from the directory `$stelelogic/sdt/examples/demongame/sdl92/packages` (**on UNIX**), or `C:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\examples\demon game\sdl92\packages` (**in Windows**), and use the Organizer to connect the diagram to the file.

Redefined Process Type Main

The REDEFINED process type `Main` inherits implicitly from the `VIRTUAL` process type `Main` in the package `BasicFeatures`, and adds the code to receive the signals that command the new features.

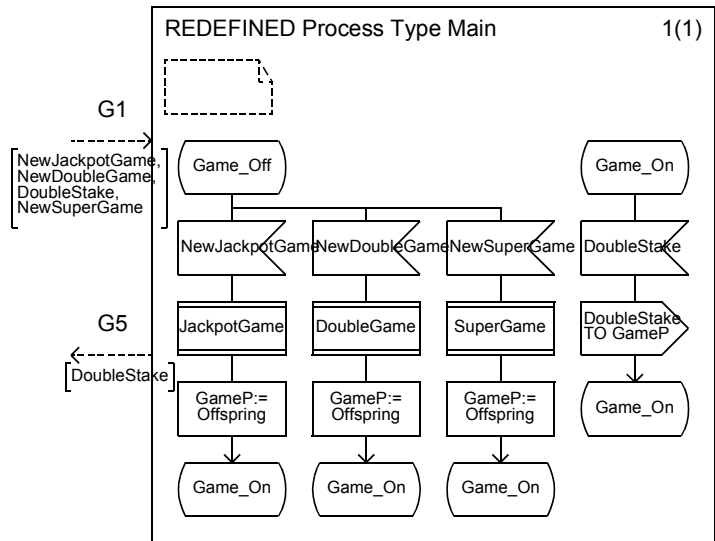


Figure 180: The redefined process type `Main`

- The REDEFINED process type `Main` is also provided on the file `advancedmain.spt` in the directory `$stelelogic/sdt/examples/demongame/sdl92/packages` (**on UNIX**), or `C:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\examples\demon game\sdl92\packages` (**in Windows**). Copy the file and use the Organizer to connect the diagram to the file.

Creating the System AdvancedDemonGame

Creating the system is now fairly simple.

1. Add a New SDL system in the Organizer. Say you name the system AdvancedDemonGame and save it as `demongameadvanced.ssy`
2. With the SDL Editor, *Copy* the contents of the system BasicDemonGame and *Paste* them into the new system.
3. Have the system USE AdvancedFeatures in addition to BasicFeatures.
4. Change the reference from the block type BasicGameBlock to AdvancedGameBlock.
5. Update the signal list C1 with the new signals JackpotGame, etc. The system is now complete. Analyze it and simulate it if you find it meaningful.

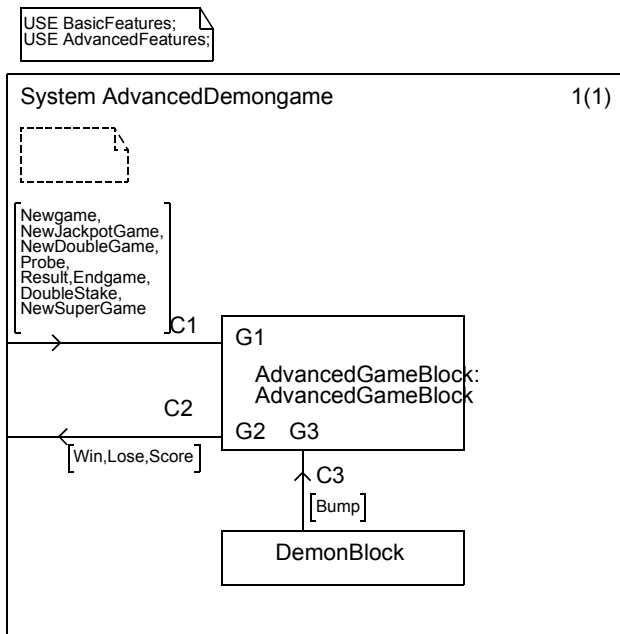


Figure 181: The system AdvancedDemonGame

Conclusion

The SDL-92 session of this tutorial has shown how to design a (small) SDL system so that the result becomes reusable components, which in turn reduces the effort needed to maintain and extend the functionality.

The tutorial also illustrates the need to design the system properly in order to introduce the OO paradigm in a smooth way.

To verify that you have assimilated the SDL-92 tutorial, you should now be ready to add new features on your own, without having to re-write the whole system.

More Exercises...

As a “menu” of new features that can be introduced, we suggest that you try to extend the `AdvancedDemonGame` with the following:

1. Memorization of “highest score ever” since system start (there should be only one highest score, common for all types of games).
2. A “hall of fame” that memorizes the name of the player that reaches the “highest score ever”. (The name is assumed to be provided by the environment).
3. A “gameover” function that checks if the current score is less than an arbitrary value of, say -100, and disables the game so that the player needs to restart it entirely.

Good luck!

Note:

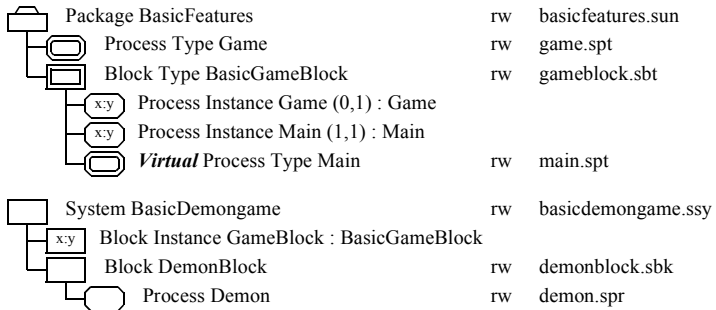
A suggestion for a solution for the exercises above can be found in the directory:

`$telelogic/sdt/examples/demongame/sdl92/exercises` (**on UNIX**), or

`C:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\examples\demon game\sdl92\exercises` (**in Windows**)

Appendix: Diagrams for the DemonGame Using Packages

Chapter Diagram Structure (basic)



Chapter Diagram Structure (advanced)

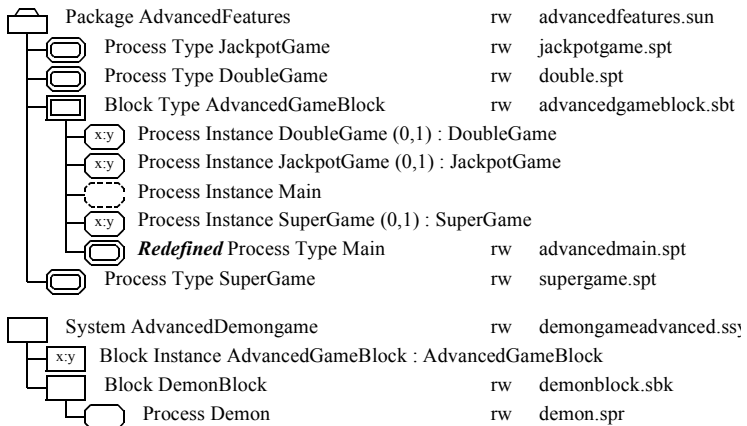


Figure 182: Hierarchical structure

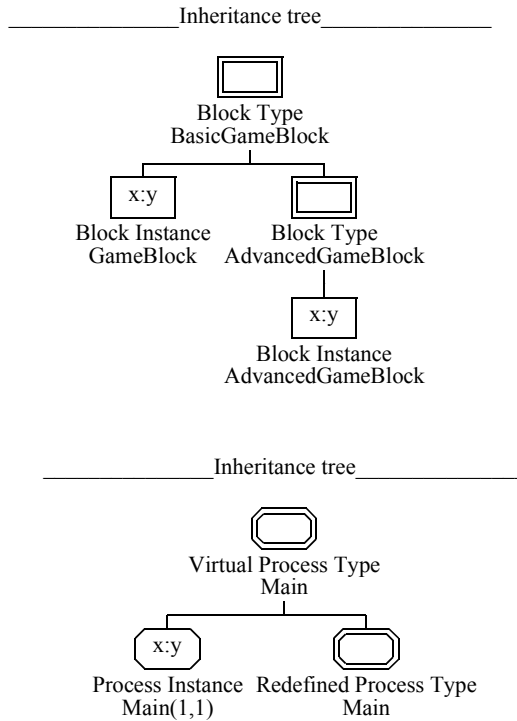


Figure 183: Inheritance tree for the block type and process type Main

(The inheritance tree for the process type Game is displayed in [Figure 170 on page 252](#).)

Cmicro Targeting Tutorial

This tutorial takes you through the first steps of targeting. Currently this tutorial is designed for using a a Microsoft Visual C compiler in Windows, and gcc or cc on UNIX.

Prerequisites / Abbreviations Used

Before you run this tutorial, you should be familiar with the SDL Suite tools, especially the Organizer, the SDL Analyzer and the SDL Simulator. If you have not already done so, you are recommended to go through the previous tutorials in this volume.

The following notations and directories concern the rest of this tutorial:

- `<installation>` denotes the installation directory, which is called `<systemdrive>:\IBM\Rational\SDL_TTCN_Suite6.3` **in Windows** and `$telelogic` **on UNIX**.
- In this tutorial there is a mixed use of the path separation characters `'/'` and `'\'`, as several steps **in Windows** and **on UNIX** only differ in these.
- Although “directories” are sometimes called “folders” **in Windows**, this tutorial always uses the expression “directory”.

You will find this tutorial placed in the directories:

```
<installation>\sdt\examples\cmicrotutorial\wini386  
<installation>/sdt/examples/cmicrotutorial/sunos5
```

- The Cmicro Library can be found in the directories:

```
<installation>\sdt\sdt_dir\wini386\cmicro  
<installation>/sdt/sdt_dir/sunos5sdt_dir/cmicro
```

For a description of the Cmicro Library please view [chapter 66, *The Cmicro Library, in the User's Manual*](#).

Introduction

General

This tutorial is divided into three sections. In the first section you will take a small SDL system and generate it with the SDL Analyzer. You will learn about configuration possibilities using the Targeting Expert and how to create environment functions. Finally, you will build the target application.

In the second section you will test it and learn something about how to use the SDL Target Tester.

In the third section you will learn how to remove the Target Tester source from the target application.

Integrations

Targeting may be implemented using the following methods:

- Bare integration:

The SDL system is running on a bare target, scheduled by the Cmicro Kernel without any other operating system (OS).

- Light integration:

The SDL system (scheduled by the Cmicro Kernel) is running as one task in an OS, possibly using functions of the OS.

- Tight integration:

All the SDL process instance sets (and other tasks) are scheduled by the OS of the target.

In this tutorial you will be doing a light integration as the target application is executed as an independent OS task, where the SDL processes are scheduled by the Cmicro Kernel.

Target Tester Communication

The communication between the Target Tester and the target application is done using sockets (localhost, port 9000 as default) in this tutorial.

Prerequisites to the Example

The Pager System

The SDL system that will be used for this tutorial is a pager system. A pager is a small hand-held device used for contacting people. It contains a radio receiver which is capable of receiving signals on a certain frequency consisting of short messages and telephone numbers.

The pager has also a sort of databank with a limited capacity for storing messages as well as a keypad and a display which serve as the interface to the user. The user has the option of scrolling through, reading and deleting the messages that are displayed on the small screen.

The keypad consists of three buttons; one for scrolling to the right, one for scrolling to the left and one for deleting. The pager emits a sound when a new message has arrived and also when the user makes an error or tries to do something which is not allowed. For example, trying to delete a message when the databank is empty or scrolling too far in a certain direction would be instances of illegal actions. Naturally, the pager can only hold a certain amount of messages and therefore at some point eventually fills up.

When the pager has reached its capacity a warning message is given for 2 seconds before the received message is displayed.

The SDL Overview shows the pager system divided into blocks and processes.

Prerequisites to the Example

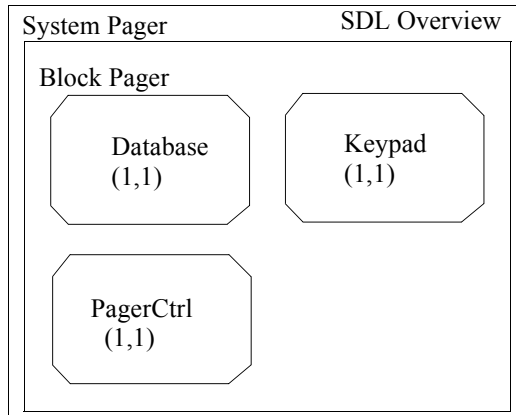


Figure 184: An overview of the system Pager

Process	Description
Database	The process Database manages the array of messages that makes up the pager's memory. It can store messages, retrieve them and delete them while maintaining order in the databank.
PagerCtrl	PagerCtrl basically handles all the input and output of the system. It receives input from the user via the keypad, messages from the radio receiver and information from the database regarding the status of saving and deleting.
Keypad	The process Keypad converts the input from the user into a signal and sends it to PagerCtrl.

Delivered Files

The files needed for this tutorial can be found in the directory:

```
<installation>/sdt/examples/cmicrotutorial/<platform>/pager
```

The project directory `pager` includes a sub directory called `system`. The directory `system` contains the SDL/GR files of the Pager system.

Furthermore, there is a directory prepared in parallel to the `system` directory. Here you can find an environment file `env.c` which can be used if you are not interested in programming the environment on your own.

Targeting

Preparations - File Structure

1. Create a new empty directory `<cmicrotutorial>` in your home directory or on your local hard disk. This directory will be denoted by `<MyTutorial>` in the following.
2. Copy the directory
`<installation>/sdt/examples/cmicrotutorial/<platform>/pager`
(including all files and subdirectories) to your new `<MyTutorial>` directory and remove all write protections.
3. In the Organizer, open the Pager system (`Pager.sdt`) found in `<MyTutorial>/pager/system`.

Using the Targeting Expert

1. Select the system symbol in the Organizer view.
2. Start the Targeting Expert from the *Generate* menu. The Targeting Expert will generate a default partitioning diagram model (see [“Partitioning Diagram Model File” on page 2973 in chapter 59, The Targeting Expert](#)) and will check the directory structure.
3. As the target directory specified in the Organizer does not exist yet, you will be prompted if it should be created. Press the *Yes* button.

A sub-directory structure is added in the target directory afterwards by the Targeting Expert. For further information see [“Target Sub-Directories Structure” on page 2998 in chapter 59, The Targeting Expert](#).

Note:

If the Targeting Expert is started the very first time a welcome window is displayed. Just press the *Close* button and proceed. The welcome window will be shown any time you start the Targeting Expert again until you select the “Do not show again at startup” check box.

The work flow of the Targeting Expert is divided into four steps.

- [Step 1: Select the Desired Component](#)
- [Step 2: Select the Type of Integration](#)
- [Step 3: Configure the Build Process](#)
- [Step 4: Make the Component](#)

The very first time you are using the Targeting Expert, an assistant is automatically started showing you how to proceed. When you have closed the assistant, you can always re-start it by choosing the menu option *Help > Assistant*.

The Help Viewer will be displayed and show the appropriate manual page if you click on one of the numbered boxes.

Step 1: Select the Desired Component

- Click on the component in the partition diagram model.
The complete SDL system is (per default) generated into the component “component”.

Hint:

The component can be given any name you like if the system is deployed using the Deployment Editor. For more detailed information on how to deploy a system, see [chapter 40, *The Deployment Editor*, in the User's Manual](#).

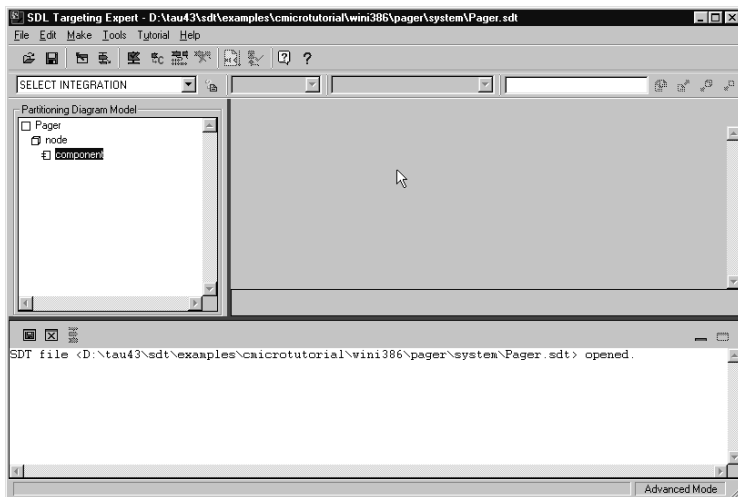


Figure 185: The Targeting Expert's main window

Background Information

- More details about partitioning diagram models can be found in [“Partitioning Diagram Model File” on page 2973 in chapter 59, The Targeting Expert.](#)
- For further information concerning the selectable entries in the partitioning diagram model, please see [“Targeting Work Flow” on page 2928 in chapter 59, The Targeting Expert.](#)

Step 2: Select the Type of Integration

1. Press the left most combo box in the integration tool bar of the main window (or click the component entry in the partitioning diagram model using the right mouse button). This is shown in [Figure 186](#).



Figure 186: Popup menu in the Targeting Expert

2. A tree structure containing all the pre-defined integrations is shown in the popup menu displayed. Select *Light Integrations* > *Application TEST*.

The SDL system is now checked for correctness and the automatic configuration is done.

Hint:

Although only the automatic configuration files `sdl_cfg.h` and `ml_mcf.h` are needed the SDL to C compiler also generates the files: `component.c`, `component.ifc`, and `component.sym`.

After the SDL to C compiler has finished, the Targeting Expert:

- generates the file `env.c` and lists all the SDL signals from and to the environment (see event log). A manual adaptation is needed for each signal before the generated files and the library files can be compiled. This is necessary as the Targeting Expert only generates a skeleton with the used signals.

How to edit the `env.c`, is described in [“Edit the Environment File” on page 279](#).

- generates default Target Tester options (file `sdtmt.opt`)
- generates a default manual configuration (file `ml_mcf.h`)

Hint:

After the integration has been selected the Targeting Expert automatically sets the default compiler. The default compiler’s name is taken from the preferences.

If a different compiler is required than the one set as default in the preferences, it is possible to change this in the integration tool bar’s combo box.

Background Information

If you want to do an integration not given in the SDL Suite distribution, please select the entry *<user defined>* in the integration popup menu. Then you are able to do all settings needed for the used hardware. You are also able to set up your own integration accessible in the integration popup menu.

The contents of the files:

- `component.c`
Describes the SDL system’s behavior in C functions.
- `component.ifc`
The header file for the environment functions. E.g. it provides PIDs and type definitions for signals.
- `component.sym`
Provides information on SDL symbols. Necessary for tracing the SDL system with the SDL Target Tester.
- `sdl_cfg.h`
The automatic configuration file for the Cmicro Kernel. For instance, if you use a timer, the file contains a define, which turns the timer implementation on.

Edit the Environment File

In this section you will learn how to fill in the environment functions in the file `env.c`.

Hint:

There is also a prepared `env.c`. You can copy it from directory:

`<MyTutorial>/pager/prepared` into

`<MyTutorial>/pager/target/pager._0/Application_TEST.`

Do not forget to remove the write protection!

Note the differences between the prepared and generated file.

Note:

In the following section the Targeting Expert starts a text editor. Per default the built-in editor is used. This can be changed in the *Tools* > *Customize* menu.

1. Select the menu *Edit* > *Edit Environment File* to open the file `env.c`

Caution!

In the file `env.c`, you should only edit code between the lines:

```
/* BEGIN User Code ... */
/* END User Code ... */
```

The reason is:

If the Targeting Expert needs to generate the file a second time, the code in these sections will be read in and copied to the new file. Only the code between the mentioned lines will be unchanged.

Do NOT edit lines with the text:

```
/* BEGIN User Code ... */
/* END User Code ... */
```

2. Find the lines from the global section:

```
/* BEGIN User Code (global section)*/
/* It is possible to define some global variables here */
/* or to include other header files. */
```

This tutorial describes a console application. It will use the screen and the keyboard to communicate with the user. It is necessary to include the used header file(s).

For a better style some defines are used. Further some global variables and functions have to be implemented. The functions are called, if there should be something simulated in the environment, like a display. After the line

`/* or to include other header files. */`
the following code needs to be inserted:

```
#if defined(MICROSOFT_C)
#include <conio.h>
#else
#include <stdio.h>
#endif

#define key_was_pressed 1
#define key_not_pressed 0

int KeySignalPresent = 0;
char LastKeyPressed;
```

xInitEnv()

3. We like to have a welcome message displayed when the system is started. This can be done like this in the function `xInitEnv()`

```
printf("----- Welcome to Pager system-----\n\n");
printf("get message : 0 to 4\n");
printf("scroll right: r\n");
printf("scroll left : l\n");
printf("delete      : d\n\n");
```

The code must be inserted between

```
/* BEGIN User Code (init section) */
and
/* END User Code (init section) */
```

xInEnv()

4. Now you have to handle the data from the environment. In this tutorial it means you have to handle the input from the keyboard!

Caution!

Do not use blocking functions in the environment file.

The environment is polled with every cycle of the Cmicro Kernel. That is the reason why it is not allowed to use blocking functions like `getchar()`. These kind of functions stop the kernel making it unable to process events in the SDL system, for example expired timers.

The possible handling of the data:

If a key has been pressed the digits 0-4 are recognized as a message and the letters 'r', 'l' and 'd' are the commands for scrolling and deleting.

Please go to the code position of the function `xInEnv()` with the following lines:

```
/* BEGIN User Code (variable section)*/  
/* It is possible to define some variables here */  
/* or to insert a functionality which must be polled */
```

Below these lines insert the following code:

```
char my_inkey;  
  
KeySignalPresent = key_not_pressed;  
  
#if defined(XMK_UNIX)  
    my_inkey = 0;  
    if((my_inkey = getchar_unlocked()) != 0)  
    {  
        KeySignalPresent=key_was_pressed;  
    }  
#elif defined(MICROSOFT_C)  
    if (kbhit())  
    {  
        my_inkey= getch();  
        KeySignalPresent=key_was_pressed;  
    }  
#endif  
  
if (KeySignalPresent==key_was_pressed)  
{  
    if ((my_inkey == 'r') ||  
        (my_inkey == 'l') ||  
        (my_inkey == 'd') ||  
        ((my_inkey>='0') && (my_inkey<='4')))  
        LastKeyPressed = my_inkey;  
    else  
        LastKeyPressed=0;  
}  
else  
    LastKeyPressed = 0;
```


5. Find the following lines of code in the function `xInEnv()`

```
/* BEGIN User Code <ScrollRight>_1 */
if (i_have_to_send_signal_ScrollRight)
/* END User Code <ScrollRight>_1 */
```

In step 2 we implemented the variable `LastKeyPressed`. In step 4 we assigned it the value of `my_inkey` which has the value of the last key pressed. Modify the `if()` statement into:

```
if (LastKeyPressed == 'r')
```

6. Find the following line of code in the function `xInEnv()`

```
GLOBALPID(Who_should_receive_signal_ScrollRight,0));
```

The process type ID which should receive the signal `ScrollRight` needs to be inserted. To get an overview of the process type IDs look at the dialog window that has pop-ed up by the Targeting Expert. All the used process type IDs are given here.

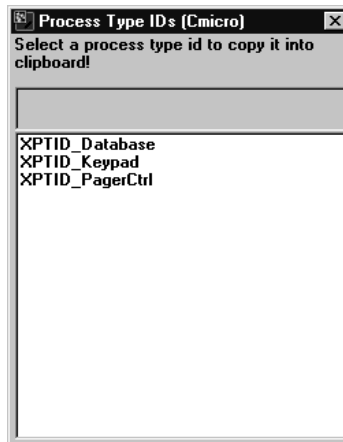


Figure 187: Process type ID dialog

Select the `XPTID_Keypad` entry in this dialog to copy it into the clipboard. Then paste it into the `env.c` as shown below.

```
GLOBALPID(XPTID_Keypad,0));
```

The function returns, and the signal is treated.

7. Find the following lines in the function `xInEnv()`

```
/* BEGIN User Code <ScrollLeft>_1 */
```

```
    if (i_have_to_send_signal_ScrollLeft)
/*    END User Code <ScrollLeft>_1 */
```

Modify the `if ()` statement to:

```
if (LastKeyPressed == 'l')
```

8. Find the following line of code in the function `xInEnv()`

```
GLOBALPID(Who_should_receive_signal_ScrollLeft,0);
```

Copy the `XPTID_keypad` as described in step 6.

```
GLOBALPID(XPTID_Keypad,0);
```

9. Find the following lines of code in the function `xInEnv()`

```
/* BEGIN User Code <Delete>_1 */
    if (i_have_to_send_signal_Delete)
/*    END User Code <Delete>_1 */
```

Modify the `if ()` statement to:

```
if (LastKeyPressed == 'd')
```

10. Find the following line of code in the function `xInEnv()`

```
GLOBALPID(Who_should_receive_signal_Delete,0);
```

Copy the `XPTID_keypad` as described in step 6.

```
GLOBALPID(XPTID_Keypad,0);
```

11. Because we do not use a real target hardware, but simulate the Pager system, we have to predefine some messages.

Find the following lines of code in the function `xInEnv()`

```
/* BEGIN User Code <ReceivedMsg>_1 */
if (i_have_to_send_signal_ReceivedMsg)
/*    END User Code <ReceivedMsg>_1 */
```

Modify the `if ()` statement to:

```
if ((LastKeyPressed>='0')&&(LastKeyPressed<='4'))
```

This `if`-statement checks whether the key hit on the keyboard was one of the defined keys or not.

12. Go to the next empty “User Code” section and insert following lines:

```
char *p;
xmk_var.Param1.MyText = (SDL_Charstring)NULL;

switch (LastKeyPressed)
{
    case '0':
        p = " Hello user";
        xmk_var.Param1.TelNumber = 12345;
        break;
    case '1':
        p = " How do you feel doing targeting?";
        xmk_var.Param1.TelNumber = 555555;
        break;
```

```

    case '2':
        p = " Targeting is all so easy!";
        xmk_var.Param1.TelNumber = 987654;
        break;
    case '3':
        p = " I only wanted to check if it works.";
        xmk_var.Param1.TelNumber = 45454;
        break;
    case '4':
        p = " ... and it works very fine!";
        xmk_var.Param1.TelNumber = 911911;
        break;
    default :
        break;
}
xAss SDL_Charstring(&(xmk_var.Param1.MyText), p,
XASS_AC_ASS_FR);

```

In this part the messages to the equivalent numbers 0-4 are stored. With the `switch` statement it is decided which one is handed over to the environment.

Note:

This way of “receiving” messages is of course just a helper function because we do not use a real interface here!

The line `xmk_var.Param1.MyText = (SDL_Charstring)NULL;`

means that the element `MyText` of the parameter `message` which is a parameter of the signal `ReceivedMsg` is set to null.

The Signal `ReceivedMsg` and the parameter `message` are declared in the SDL system.

The line `xAss SDL_Charstring(&(xmk_var.Param1.MyText), p, XASS_AC_ASS_FR);` allocates memory for the pointer `p`.

Note:

If an SDL charstring is mapped to an array of char in C, the first character in this array (index 0) is for internal use only, i.e. the text message should start at index 1. This is done by having a space in front of the text in the implementation shown above.

- Find the following line of code in the function `xInEnv()`

```
GLOBALPID(Who_should_receive_signal_ReceivedMsg,0));
```

Modify the statement as showed below.

```
GLOBALPID(XPTID_PagerCtrl,0));
```

xOutEnv()

Caution!

The function `xOutEnv()` provides a pointer named: `xmk_TmpDataPtr`. The data referenced by this pointer is valid only as long as the function `xOutEnv()` is processed.

If you need to treat the data after leaving the function, copy it to variables defined by you.

14. Find the following code section:

```
case CurrentMsg :
{
    /* BEGIN User Code <CurrentMsg>_1 */
    /* Use (yPDP_CurrentMsg)xmk_TmpDataPtr to access
    the signal's parameters */
    /* ATTENTION: the data needs to be copied. Otherwise it */
    /* will be lost when leaving xOutEnv */
    /* END User Code <CurrentMsg>_1 */

    /* BEGIN User Code <CurrentMsg>_2 */
    /* Do your environment actions here. */
    xmk_result = XMK_TRUE; /* to tell the caller that
*/
/* signal is consumed
*/
/* END User Code <CurrentMsg>_2 */
}
```

This code fragment handles the signal `CurrentMsg`. The chosen message is displayed on the screen (telnumber, message, current message position and the total number of messages). Insert the following code after: `/* Do your environment actions here. */`

```
printf("\r\n");
printf(" \rCurrentMessage: %6d %s (%d/%d)",
      ((yPDP_CurrentMsg)xmk_TmpDataPtr)->Param3.TelNumber,
      ((yPDP_CurrentMsg)xmk_TmpDataPtr)->Param3.MyText+1,
      ((yPDP_CurrentMsg)xmk_TmpDataPtr)->Param1,
      ((yPDP_CurrentMsg)xmk_TmpDataPtr)->Param2);
xFree(&(((yPDP_CurrentMsg)xmk_TmpDataPtr)->Param3.MyText));
```

The line

```
xFree(&(((yPDP_CurrentMsg)xmk_TmpDataPtr)->Param3.MyText));
```

free the memory allocated in the kernel when sending the signal.

15. Find the following code in the function `xOutEnv()`:

```
case ServiceMsg :
{
    /* BEGIN User Code <ServiceMsg>_1 */
    /* Use (yPDef_Close*)xmk_TmpDataPtr to access
    the signal's parameters */
    /* ATTENTION: the data needs to be copied.
    Otherwise it */
```

```

/*                                will be lost when leaving xOutEnv */
/*  END User Code <ServiceMsg>_1 */

/* BEGIN User Code <ServiceMsg>_2 */
/* Do your environment actions here. */

xmk_result = TRUE; /* to tell the caller that */
/* signal is consumed */
/*  END User Code <ServiceMsg>_2 */
}

```

The code fragment handles the signal `ServiceMsg`. The handling of the data is same as in the step before. So insert the following code after: `/* Do your environment actions here. */`

```

printf("\r\nServiceMessage: %s",
);
printf( "\r\nServiceMessage: %s",
        ((yPDP_ServiceMsg)xmk_TmpDataPtr)->Param1+1);
xFree(&((yPDP_ServiceMsg)xmk_TmpDataPtr)->Param1);

```

16. Find the following code section in the function `xOutEnv()`:

```

case ShortBeep :
{
    /* BEGIN User Code <ShortBeep>_1 */
    /*  END User Code <ShortBeep>_1 */

    /* BEGIN User Code <ShortBeep>_2 */
    /* Do your environment actions here. */
    xmk_result = XMK_TRUE; /* to tell the caller that */
                          /* signal is consumed */
    /*  END User Code <ShortBeep>_2 */
}
break ;

```

The code fragment handles the signal `ShortBeep`. A beep sounds when the pager receives a message or you do something which is not allowed. After the code `/* BEGIN User Code <ShortBeep>_1 */` insert

```
putchar(07);
```

17. Find the following code section in the function `xOutEnv()`:

```

case LongBeep :
{
    /* BEGIN User Code <LongBeep>_1 */
    /*  END User Code <LongBeep>_1 */

    /* BEGIN User Code <LongBeep>_2 */
    /* Do your environment actions here. */
    xmk_result = XMK_TRUE; /* to tell the caller that */
                          /* signal is consumed */
    /*  END User Code <LongBeep>_2 */
}
break ;

```

Insert the following code after

```

/* BEGIN User Code <LongBeep>_1 */

putchar(07);

```

```
putchar(07);
```

Closing the Environment

In this tutorial there is no need to close the environment. In other cases, e.g. microprocessor hardware, it is probably necessary to do so.

Have a look in the file `env.c`, find code like this in the function `xCloseEnv()`:

```
/* BEGIN User Code (close section) */  
/* Do the actions here to close your environment */  
/* END User Code (close section) */
```

Insert any code you need to have here.

Step 3: Configure the Build Process

1. Press the items below the `Application TEST` in the partitioning diagram. If it is necessary to add or remove settings for your job, you can edit the settings.
2. Click *Save* to close the dialog.

For this section of the tutorial there it is not necessary to modify anything, though we will do some modifications later in section [“Run Target get EXE without Tester” on page 294](#).

Background Information

Short description of the different areas:

- *Compiler / Linker / Make*

In this area it is possible to configure all the settings used for the Compiler, Linker and Make tools.

Something special regarding *Additional Compiler*. For example, you have to use an ANSI C- compiler to compile the generated files, and it is necessary to link the objects with the object from one file, which needs to be compiled with a C++ compiler. In this instance you could enter the regarding file and the used compiler in the section *Additional Compiler*.

For more information see: [“Configure Compiler, Linker and Make” on page 2932 in chapter 59, The Targeting Expert](#).

- *Target Library*

In this area it is possible to set defines and values to scale the target library. For more information see [“Configure and Scale the Target Library” on page 2948 in chapter 59, The Targeting Expert.](#)

All settings will be stored in the file `ml_mcf.h`.

- *Target Tester*

In this area it is possible to set defines and values to scale the functionality of the tester. For more information see [“Configure the SDL Target Tester \(Cmicro only\)” on page 2949 in chapter 59, The Targeting Expert.](#)

All settings will be stored in the file `ml_mcf.h`.

- *Host Connection*

In this area it is possible to set the parameter of the connection to the host. For example, it contains the description of the message coding and the name of the executable, etc. The configuration of the *Host Connection* is always stored in the file `sdtmt.opt`. This file is mandatory for the SDL Target Tester. For more information see: [“Configure the Host \(Cmicro only\)” on page 2950 in chapter 59, The Targeting Expert.](#)

Step 4: Make the Component

1. In the dialog which is displayed by default (when the `Application TEST` is selected) you have to select two check boxes. *Analyze / Generate Code* and *Environment functions*.
2. Click on the button *Full Make* to start the code generation and make.

After the SDL to C compiler has finished the code generation, the Targeting Expert will re-generate the `env.c` (and keep your modifications). Afterwards it generates a makefile with the given settings and the code will be compiled and linked.

The Targeting Expert then starts the SDL Target Tester.

Use of the SDL Target Tester

Differences between SDL Simulator and SDL Target Tester

The difference compared to the SDL Simulator is that the generated SDL system is running on a target hardware and is sending messages to the host system on which the SDL Target Tester's host part is running.

In this Cmicro tutorial, the SDL Target Tester's host part is running on the host as well as the generated SDL system (target).

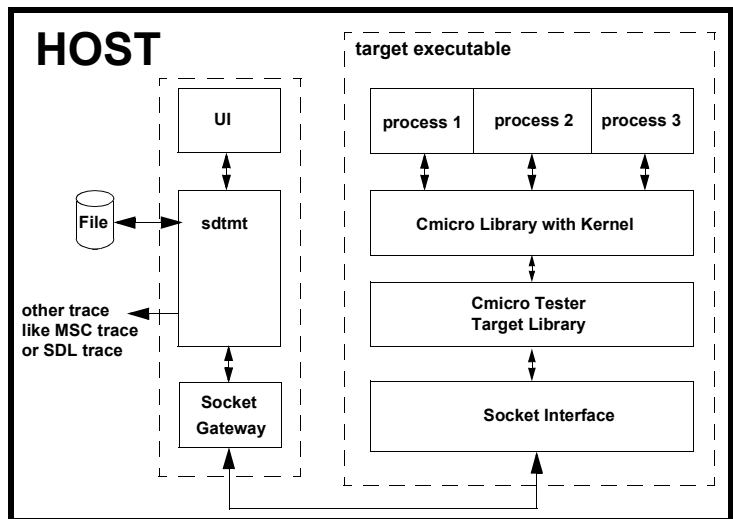


Figure 188: Communication links between the processes

Restrictions in this Tutorial

It is possible to use all the features supported by Cmicro (e.g. signal priorities, error checks, tester) except the preemptive Cmicro Kernel. This is due to the concept behind Cmicro which is designed for small targets on a stand-alone hardware (bare integration).

Background Information

- To get information on how it works with a real target hardware, see [chapter 66, *The Cmicro Library, in the User's Manual*](#).
- If you want to edit the makefile, or have a look into it, please use the menu *Edit > Makefile* in the Targeting Expert. For more information please view [chapter 59, *The Targeting Expert, in the User's Manual*](#).

Testing the Pager System

Running the SDL Target Tester

The SDL Target Tester could be started automatically from the Targeting Expert or by:

- selecting *Tools > SDL > SDL Target Tester* in the menu of the Targeting Expert.
- pressing the quick button in the quick button bar.

After it has been started the following steps have to be done (in the given order)

1. Start the communication with the executable by pushing the button *StartGateway* in the *Communication* group or with the menu *Execute > StartGateway*.
2. Go back to the Targeting Expert and select the menu entry *Tutorial > Start target (Windows)* or respectively *Tutorial > Start target (UNIX)*

Hint:

The menu *Tutorial* is a configurable menu not available if working on other SDL systems. Please refer to [“Configurable Menus” on page 2907 in chapter 59, *The Targeting Expert, in the User's Manual*](#) for more information on how to create your own menu entries.

Note:

In Windows the target application is started in a separate command prompt window.

On UNIX the target application is started in a separate xterm.

3. Now the communication between SDL Target Tester and target is established. The SDL Target Tester displays the message `start` with “Go Forever”. The corresponding button is located in the *Execute* group. Press the button.

All messages of the target application are displayed in the UNIX shell or DOS command prompt where it was started. The pager will display its start-up message which was implemented in [“xInitEnv\(\)” on page 280](#).

4. You can “receive” messages by pressing the keys *0*, *1*, *2*, *3*, *4* and scroll with the keys *r* and *l*. To delete a message press *d*. See [“xInEnv\(\)” on page 280](#) on how the environment was implemented.
5. The pager displays the last received message, the number of the current message and the total number of received messages. Now you can scroll, delete and receive new messages. If you receive a new message while the maximum amount of messages(3) is reached, the pager saves the message temporary and displays the warning:
Memory full, please free memory to get new messages for about 2 seconds.
6. Press the *d* key on your keyboard, the last message is then deleted and the received message is displayed.

Note:

To exit the target application press `CTRL+C` on the keyboard.

SDL Target Tester Commands

The SDL Target Tester has button groups. Each button represents a Tester command. You can use the buttons or the command line at the bottom of the Tester to enter a command. If you enter `help` in the command line you will see a list of SDL Target Tester commands.

In the following some Tester commands are explained briefly. For more information, see [chapter 67, The SDL Target Tester](#).

Tracing the SDL System -> MSC Editor

Similar to the SDL Simulator GUI it is possible to generate MSC traces while testing the system with the SDL Target Tester.

- In the *Trace* group you can select the *Start MSC* button, for example, to start the MSC trace.
- Now you can select between output via display or output to a file.

Tracing the SDL System -> SDL Editor

It is possible to trace the target system with the SDL Editor.

- You can start the trace with the command line of the SDL Target Tester by typing `start-sdle`, or by using the button *Start SDLE*.

The MSC trace and SDL trace functions are powerful tools for understanding the system.

Target Information

To get more information about target configuration, you must open the *Configuration Group* in the button area of the SDL Target Tester and press *Target* to get the current target configuration.

To get information about the kernel, you have to open the *Examine Group*. If you press *Queue* you will get information about the current state of the internal queue of your system. You will see the peak hold and the amount of signals of your current system. By pressing the other buttons in the *Examine Group*, you will get more information of the running system.

Memory

With the `?memory` command you can see how the current memory state is.

1. Start the Pager system as described in [“Running the SDL Target Tester” on page 290](#).
2. Type `?memory` (the short command `?m` can also be used) on the command line. You can check the memory pool size, the current memory fill and so on.

Now we will see how the memory is handled in the Pager system. Notice that the current amount of blocks in pool is four and the peak hold is five.

3. Switch to the target application and press a key (1-4) to get a message.

4. Go back to the Target Tester and execute the `?m` command again. This time you can see that two more blocks are allocated. If you delete the last message the memory should be freed again and show four blocks.

Breakpoints and Queues

To debug the system you can use the *Breakpoints* button group. You can set a breakpoint on a signal input or a process state. If a breakpoint was reached you can continue the system with the button *Continue*.

1. Restart the target as described in [“Running the SDL Target Tester” on page 290](#).
2. Expand the *Breakpoints* group and select *Break input*.
3. Now you can choose a process ID. In this example we take the *Keypad ID*.
4. A signal ID list is shown afterwards, select the *delete* signal.
5. Breakpoint on *input is set* is shown in the text area now, switch to the target application window and insert some messages first, then press *d*. Nothing happens.
6. Switch to the Tester again. Now you can use the `?memory` command or look how the *Queue* looks like (`?queue`).
7. Press *Continue* after you have examined the system state.

As the system will be halted every time the signal *delete* is to be consumed in process *Keypad*, it is probably useful to delete the breakpoint(s) by entering the command `BA`.

Run Target EXE without Tester

It is also possible to run the target executable without starting the Target Tester. Following has to be done first:

1. Exit the target application and the Target Tester if not already done.
2. Switch back to the Targeting Expert.

In the following there is a description what has to be done to remove the Target Tester source code form the target application.

1. Press the entry `Target Tester` below `Application TEST` in the Partitioning Diagram Model. As you can see all the selected Target Tester flags are disabled, i.e. it is not possible to switch them off because the pre-defined integration selected prevents doing so.

To get access to these flags, the Targeting Expert provides a so called “Advanced Mode”. (For details see [“Advanced Mode” on page 2926 in chapter 59, The Targeting Expert, in the User’s Manual](#)).

2. Select the menu entry *Tools > Customize* and a dialog pops up. Select the check box *Advanced Mode* and press the dialog’s *OK* button.

Caution!

The Advanced Mode is now switched on each time you enter the Targeting Expert again. Make sure you switch off the Advanced Mode to take advantage of the restrictions given with all the other pre-defined integration settings.

3. In the dialog displayed when the `Application TEST` entry is selected in the Partitioning Diagram Model, you have to select the *Execution* tab. Press the *None* radio button in the *Test application* group box.
This is done to disable the execution of the Target Tester after the target application has been successfully build.
4. Select `Target Tester` in the Partitioning Diagram Model and de-select the *Use the Target Tester* check box. Press *OK* in all following windows. (Depending flags are switched off). After the last

dialog has been executed, all check boxes on the *Tester* tab should be un-selected now.

5. Press the *Save* button below the dialog.

Hint:

A new manual configuration file `ml_mcf.h` is generated with all the Target Tester flags undefined.

6. Now press the *Make* button. (All the files are compiled again because the `ml_mcf.h` has been modified.)

When compile and link is completed, the target can be started via the menu *Tutorial > Start target* and can be used now without SDL Target Tester.

Tutorial: Using ASN.1 Data Types

This tutorial describes how to use ASN.1 types and values in the SDL Suite. You will learn how to import and use ASN.1 modules in your SDL diagrams, how to generate code and how to encode and decode your ASN.1 types using BER or PER encoding.

The tutorial contains all steps from creating ASN.1 data types to the implementation of the ASN.1 data types in your source code.

To illustrate the functionality and the work flow, small examples are presented throughout the tutorial. The SNMP protocol is used as a base to illustrate how ASN.1 could be applied on a typical SNMP stack.

In order for you to fully take advantage of this tutorial, you should be familiar with the SDL Suite and the basics of ASN.1.

Additional information regarding ASN.1 types and its usage together with the SDL Suite can be obtained in:

- [chapter 2, *Data Types, in the Methodology Guidelines*](#)
- [chapter 13, *The ASN.1 Utilities, in the User's Manual*](#)
- [chapter 57, *Building an Application, in the User's Manual*](#)
- [chapter 58, *ASN.1 Encoding and De-coding in the SDL Suite, in the User's Manual*](#)

Introduction

The Abstract Syntax Notation One (ASN.1) is a notation language that is used for describing structured information that is intended to be transferred across some type of interface or communication medium. It is especially used for defining communication protocols.

As ASN.1 is widely popular, the SDL Suite allows you to translate ASN.1 data types to SDL and to encode/decode ASN.1 data types.

By using ASN.1 data types in the implementation of your application, you will optimize your development process. The following list displays some of the advantages of ASN.1:

- ASN.1 is a standardized, vendor-, platform- and language independent notation.
- A vast number of telecommunication protocols and services are defined using ASN.1. This means that pre-defined ASN.1 packages and modules are available and can be obtained from standardization organizations, RFCs, etc. For instance, the ASN.1 data types defining SNMP are available in RFC 1157.
- When ASN.1 data types are transmitted over computer networks, their values must be represented in bit-patterns. Encoding and decoding rules determining the bit-patterns are already defined for ASN.1. The SDL Suite supports BER and PER encoding.
- ASN.1 enables extensibility. This means that it simplifies compatibility of systems that have been designed and implemented large time frames apart.
- The SDL Suite and the TTCN Suite can share common data types by specifying these in a separate ASN.1 module.

Implementation of ASN.1

When importing ASN.1 data types to your SDL system, you need to translate the ASN.1 definitions to SDL. The SDL Suite does this for you using a tool called ASN.1 Utilities. This tool is automatically invoked when you analyze your SDL system.

However, having the ASN.1 data types translated to SDL is not enough to include them in your application. If you are going to transfer application-generated information on computer networks, the values of the

data types must be encoded. When transferring signals in or out of your SDL system, you must also create the interface between the environment and the system.

Thus, the process of implementing ASN.1 data types can be divided into three separate steps:

1. Creating the abstract syntax
2. Creating the transfer syntax
3. Compiling the application

The definitions of the abstract syntax and the transfer syntax is presented below.

Abstract Syntax

The basic idea is to transport some type of information between two nodes using protocol messages. The abstract syntax is defined as the set of all possible messages that can be transported. To create the abstract syntax you must:

- design some form of data structure defined in a high-level programming language, for instance ASN.1.
- define the possible set of values that the data structure can take.

Transfer Syntax

The transfer syntax is the set of bit patterns that represents the abstract syntax messages with each bit pattern representing just one value. The rules determining the bit-patterns are called the encoding rules.

Creating the Abstract Syntax

When creating the abstract syntax you must perform the following tasks:

- Adding ASN.1 modules to your project
- Importing the ASN.1 modules in your SDL diagrams.
- Assigning values to the data types.

Adding ASN.1 Modules to your Project

An ASN.1 module is a file containing the ASN.1 data type definitions. If you are implementing a standard communication protocol, it is very likely that pre-defined ASN.1 modules are available. The modules can be obtained from standardization organizations, RFCs, etc. In [Example 7 on page 324](#) and the following examples ASN.1 modules that define data structures for SNMP protocol are presented, RFC1155-SMI based on RFC 1155, RFC1157-SNMP based on RFC 1157 and RFCxxxx-MIBs containing references to different objects from various Managed Information Bases.

Should a pre-defined module not be available for your type of application, you must create your own module. ASN.1 modules can be created in a simple text editor. It must contain a header and a footer. Definitions should be inserted inside, for example:

```
MyModule DEFINITIONS AUTOMATIC TAGS ::=
BEGIN

-- Definitions should be inserted here, for example,

T-Age ::= INTEGER ( 0 .. 150 )

T-City ::= ENUMERATED { moscow, malmo, new-york }

MyType ::= SEQUENCE {
    name VisibleString,
    surname VisibleString,
    age T-Age OPTIONAL,
    city T-City DEFAULT moscow
}

-- Other definitions etc.

END
```

Creating the Abstract Syntax

The header of the module can contain global settings for the module. In the example above we use AUTOMATIC TAGS which can be omitted but is a recommended flag if you need to generate encode/decode procedures for this module.

For more details please see adequate ASN.1 literature for instructions and guidelines on creating ASN.1 modules.

Regardless how you obtain the ASN.1 modules, you must add the module to your project before the SDL Suite can include the data types.

Follow the instructions below to add the ASN.1 module to your project.

1. Save your ASN.1 module in a subdirectory to your project. Make sure that you append the **.asn** file extension to the saved module.
2. Open the Organizer and select the chapter where you want to include the module. This is done by clicking the chapter marker, for instance the *Other Documents* marker, see [Figure 189 on page 301](#).

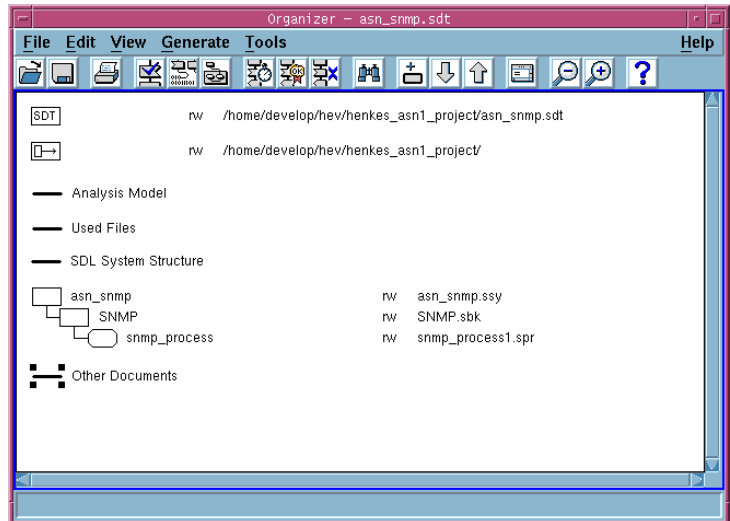


Figure 189: Selection of chapter marker

3. From the *Edit* menu, select the *Add Existing...* command. The Add Existing window opens.

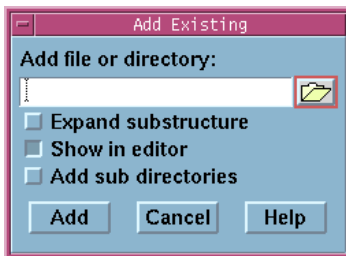


Figure 190: The Add Existing window

4. Click the folder image button in order to find your ASN.1 module. The Select file to add window opens.
5. Select the directory you want to search and change the search filter, by typing *.asn in the *Filter* field. Click the *Filter* button. The available ASN.1 modules are now displayed in the *Files* window. Select module and click the *OK* button. The Select file to add window closes.
6. The selected module is now displayed in the Add Existing window. Just click the *OK* button to add the module to your system. The module should now be visible in the Organizer in your selected chapter.

The ASN.1 modules are now added to your project.

Example 1: Adding ASN.1 modules to SDL project ---

In the SNMP example, the three modules RFC1155_SMI, RFC1157_SNMP and RFCxxxx-MIBs have been added to the project.

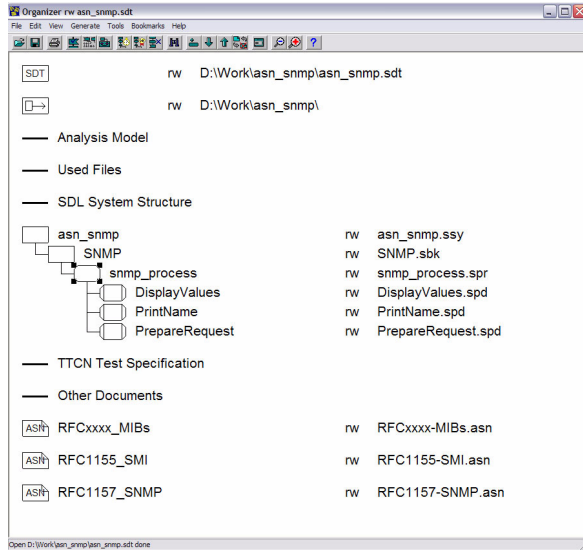


Figure 191: View of added ASN.1 modules

Importing ASN.1 Modules

After the modules have been added to the project, they must be made available to the SDL system. This is done by importing the modules to the SDL system file. When the modules have been imported, the ASN.1 data types can be used as regular SDL types.

Follow the instructions below to import the modules in the SDL diagrams:

1. From the Organizer, open the system file, <system_name>.ssy.
2. Add use of the added modules in the package reference frame, which is located outside the system frame. (See [Figure 192](#)).
3. Save the diagram.

Example 2: Importing ASN.1 modules

In the SNMP example, the three modules RFC1155_SMI, RFC1157_SNMP and RFCxxxx-MIBs have been imported.

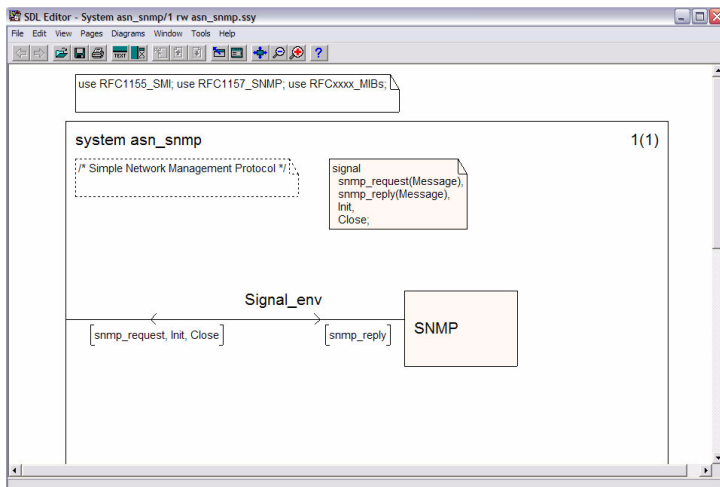


Figure 192: View of the imported ASN.1 modules

4. ASN.1 modules are imported when you Analyze an SDL system that uses these modules.
5. For each ASN.1 module an SDL package with the same name is created. It contains corresponding SDL definitions. SDL packages are saved into *.pr file in the target folder, for example:

```
use RFC1155_SMI/
newtype ObjectName,
newtype ObjectSyntax,
newtype NetworkAddress,
newtype IPAddress,
newtype TimeTicks;
/*#SDTREF(TEXT,D:\Work\asn_snmp\RFC1157-SNMP.asn)*/

package RFC1157_SNMP; /*#ASN.1 'RFC1157-SNMP'*/

syntype Message_INLINE_0 = Integer endsyntype;

synonym version_1 Message_INLINE_0 = 0;

/*#SDTREF(TEXT,D:\Work\asn_snmp\RFC1157-SNMP.asn,10,1)*/
newtype Message struct
    version Message_INLINE_0;
```

```
community Octet_string;
data PDUs;
endnewtype;

...

/*#SDTREF(TEXT,D:\Work\asn_snmp\RFC1157-SNMP.asn,123,1)*/
newtype VarBind struct
  name ObjectName;
  value ObjectSyntax;
endnewtype;

/*#SDTREF(TEXT,D:\Work\asn_snmp\RFC1157-SNMP.asn,130,1)*/
newtype VarBindList
  String (VarBind, emptystring)
endnewtype;

endpackage RFC1157_SNMP;
```

6. SDL analogues of ASN.1 types and values from generated packages can be referenced from the SDL system.

Assigning Values to the Data Types

When the modules are imported to the SDL system, you are free to declare signal parameters and variables of ASN.1 data types. The parameters and variables are treated as regular SDL parameters and variables, and you assign values to them in the same manner as you normally do.

For the SNMP example signals `snmp_request(Message)` and `snmp_reply(Message)` are declared and they carry information that corresponds to the Message ASN.1 type. Value for the type Message is constructed in the SDL process, see [Example 3](#).

Example 3: Assigning values to variables

```
dcl reply, request Message;
dcl vb VarBind;
dcl data PDUs;

synonym sysDescr_val Object_identifier = sysDescr // (. 0 .);
vb := (. sysDescr_val, simple : empty : NULL .);
data!get_request := (. 1001, 0, 0, (. vb .) .);
request := (. version_1, comPublic, data .);
```

Message, VarBind, PDUs, sysDescr and version_1 are types and values imported from ASN.1 modules. sysDescr is defined in the ASN.1 module RFCxxxx-MIBs.asn and it points to an object type corresponding to a textual description of the network entity. sysDescr // (. 0 .) points to the value of the sysDescr object type and denotes a string with the real description of the network entity. By sending a get_request with sysDescr // (. 0 .) in the 'name' field and

an empty 'value' field, network entity supporting SNMP should return the same `get_request` with the 'value' field holding the string with the description.

When your variables have been assigned values, you have created the abstract syntax.

When the variable `request` has been declared, you can use it in the SDL diagram as a regular SDL variable. [Figure 193 on page 306](#) shows how `request` is used as the argument of a signal that is sent from the SDL system to the environment.

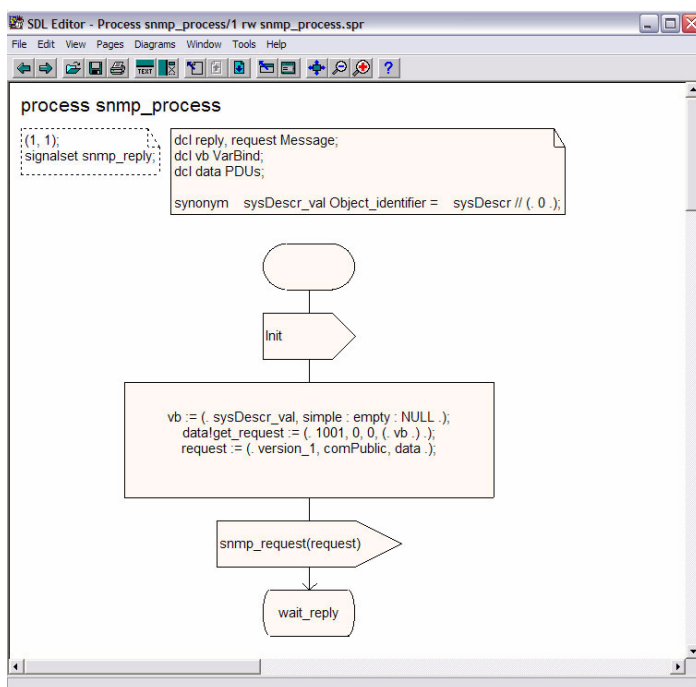


Figure 193: Usage of the request variable

Creating the Transfer Syntax

The SDL Suite offers several ways to create the transfer syntax. The available coding access interfaces are:

- Basic SDL interface
- Extended SDL interface
- C code interface

In this tutorial, only the C code interface will be covered. For a complete description of ASN.1 encoding and decoding, please see [chapter 58. *ASN.1 Encoding and De-coding in the SDL Suite, in the User's Manual.*](#)

Using the C code interface, the transfer syntax can be created either using the Organizer's make dialog or using the Targeting Expert. Both methods are presented in this tutorial. When using the Targeting Expert, you can select to use the Cadvanced SDL to C Compiler or the Cmicro SDL to C Compiler when creating the transfer syntax. Both methods will be covered as well.

This section starts with a short introduction and the actual instructions are presented in:

- [“Generating Template Files - the Organizer” on page 311](#)
- [“Generating Template Files - Targeting Expert” on page 318](#)

Introduction

To be able to transfer the abstract syntax between two nodes in network, you must first create the transfer syntax. The transfer syntax representation is then transmitted in a protocol buffer.

When creating the transfer syntax you must perform the following tasks:

- Generating template files
- Editing the generated template files

The template files must be generated in order for you to include the ASN.1 data types in the compilation and code generation processes. The template files extract information from your SDL system and create a skeleton. Often these template files do not contain sufficient informa-

tion to meet the demands of the application and therefore you must edit the templates. The template files that are generated cover the following areas:

- the environment functions
- the make process

However, the SDL Suite needs additional information in order to create the environment file. Before the generation you must determine which encoding/decoding schemes to use and you must create type nodes files.

Note: Environment File

There are several ways to create the environment file. This tutorial shows how to auto-generate the file. However, you can also make your own file from scratch. This procedure is more advanced and is only partially covered.

Note: Type Nodes

The type nodes are auto-created by ASN.1 Utilities and must not be edited.

Note: The Make process

The template makefile is only created if you are using the Make dialog. The default makefile of the Targeting Expert handles all necessary make functionality.

Environment Functions

The environment is defined as all devices or functions that are needed by the application but not specified within the SDL system. By sending signals to the environment, the SDL system wants certain tasks to be performed. This could be for instance:

- reading or writing information to a file
- sending or receiving messages across the network
- reading or writing information on hardware ports or sockets

However, the SDL system only controls events that occur within the system. It does not specify how signals leaving the system are handled by the environment. This is why you must provide an interface between

the SDL system and the environment. This interface is made up by the environment functions.

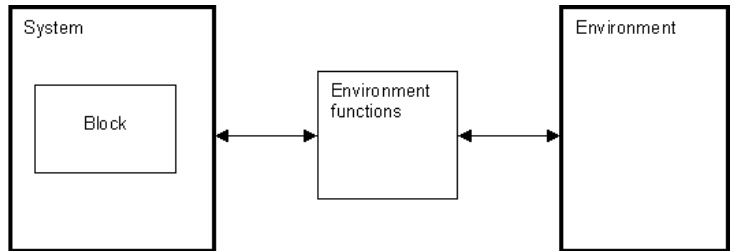


Figure 194: The environment functions

The SDL Suite is rather helpful and can generate a template environment file that includes a skeleton of the environment functions. The environment file is written in C code and by editing this file you can specify the behavior of signals from the SDL system and of signals going in to the SDL system.

An environment header file or system interface header file can also be created. This file contains all type definitions and other external definitions that are necessary in order to implement the environment functions.

Note: Environment files

There are several ways to create the environment file. You can:

- auto-create the file. This procedure is covered in this tutorial.
- make your own file from scratch. This is a more advanced procedure and is only partially covered in this tutorial.

Encoding/Decoding

When creating the transfer syntax, the messages that will be transferred must be encoded and the incoming messages must be decoded. The type of encoding rules to apply is specified in the environment file. This means that the encoding/decoding function calls must be included in the environment file.

The SDL Suite supports the standard BER and PER encoding/decoding schemes, but it also allows you to use a user specified encoding scheme. ASCII encoding is available in the SDL Suite as well, but it does not support encoding of ASN.1 types.

Type Nodes

To include the ASN.1 data types in your application, they must be translated into a form that the SDL Suite understands. Within the SDL Suite, this translation is handled by the ASN.1 Utilities.

The ASN.1 Utilities tool is invoked automatically when the SDL system is analyzed and it allows you to:

- perform syntactic and semantic analysis of your ASN.1 modules
- generate SDL code from the ASN.1 modules
- generate type information for encoding and decoding using BER or PER

This means that when you are using the ASN.1 utilities, you create type nodes. A type node is a static variable that describes the properties and characteristics of an ASN.1 data type, including tag information needed by BER/PER encoders and decoders. The variable is named `yASN1_<type_name>`.

All nodes are generated in files named `<asn1module_name>_asn1coder.c` and declarations to access them in files named `<asn1module_name>_asn1coder.h`.

Note:

The type nodes are auto-created by ASN.1 Utilities and must not be edited.

Make Process

Note: Make dialog only

This section is only valid if you build and analyze your project using the Organizer's make dialog.

The default makefile in the SDL Suite, determines the relationship between source files, header files, object files and libraries in your project.

However, the default makefile does not include the generated files in the make process. To include the environment files and the type node files in the make process, you must generate a template makefile that will be appended to the default makefile, see [Figure 195](#). The template makefile can be generated by the SDL Suite.

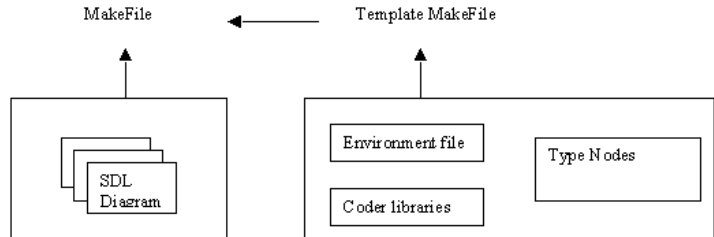


Figure 195: The make process

Generating Template Files - the Organizer

Follow the instructions below to generate environment files, type node files and the template makefile using the Organizer's Make dialog:

1. Click the SDL system symbol in the Organizer.
2. From the *Generate* menu, select the *Make...* command. The SDL Make window opens.
3. Specify your options in the make dialog according to the following list:
 - Select *Analyze & generate code*
 - From the *Code generator* drop-down list, select *Cadvanced*
 - Select *Generate environment header file*
 - Select *Generate environment functions*
 - Select *Generate ASN.1 coder*, to invoke ASN.1 Utilities.
 - From the *Use standard kernel* drop-down list, select *Application*

Note:

Make sure that you de-select the *Compile & link* option as you only want to generate the template files.

- Specify your target directory where the generated files will be stored.

[Figure 196](#) shows the Make dialog with the selected options.

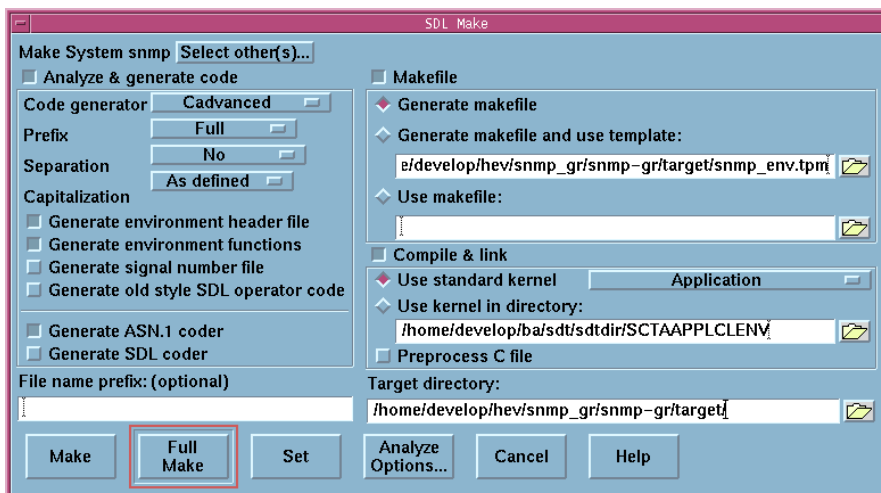


Figure 196: The Make dialog - generating template files

- Press the *Full Make* button.

Note:

Encode and decode calls are only generated if the *Generate ASN.1 coder* option is enabled in the make dialog.

In your target directory, you will now find the generated files including:

- `<system_name>_env.c`
This is the environment skeleton file.
- `<system_name>.ifc`
This is the environment header file.

- `<system_name>_env.tpm`
This is the template makefile.
- `<asn1module_name>_asn1coder.c`
`<asn1module_name>_asn1coder.h`
These files are the type nodes created by the ASN.1 utilities.

Editing the Generated Files - the Organizer

As the generated files only consist of skeleton functions, you must edit the files to suit the functionality of your application.

Note:

Make a habit of making a copy of the environment file and the template makefile after they have been edited. Otherwise the edits will be overwritten, if the files are re-generated from the Make dialog by mistake.

1. Edit the environment file `<system_name>_env.c` file using any text editor. In the skeleton file, macros are included but they are not defined. To define the required functionality, either create a `<system_name>_env.h` file and define all macros there, or replace the macros with the required code directly in the `<system_name>_env.c` file. [Example 4 on page 314](#) shows the updated SNMP environment file.
2. Save the environment file.
3. Edit the template makefile `<system_name>_env.tpm` if necessary.
4. Save the template makefile.
5. Make copies of the edited files and save the copies in a different folder.

Notes:

- In order to transfer the information on the network, you must add socket commands to an appropriate header file.
- If you want to use more than one encoding scheme, for instance BER and PER, you must enter the appropriate encoding function calls in the environment file

Example 4: Environment Functions

The SNMP `get_request` message should be encoded by BER DEFINITE and then sent to the network entity by UDP protocol to port 161 which is the default port for the SNMP requests. BER encode and decode function calls are automatically generated to the environment functions. This code should then be updated with the socket function calls and with choosing correct BER dialect.

The following code is part of the environment file for the Windows platform (`#include <WinSock.h>`) and displays the function that handles the out signals.

```
char*   data;
int     datalen, i;
tBuffer Buf = 0;
struct  sockaddr_in manager_addr, agent_addr;

XENV_OUT_START
/* Signals going to the env via the channel Signal_env */

/* Signal snmp_request */
IF_OUT_SIGNAL(snmp_request, "snmp_request")
/* Encoding message to the buffer */
BufInitBuf( Buf, bms_SmallBuffer );
ERSetRule( Buf, er_BER | er_Definite );
BufInitWriteMode( Buf );
BEREncode( Buf, (tASN1TypeInfo *) &yASN1_Message, (void
*) &((ypDef_snmp_request *) (*SignalOut)) ->Param1);
BufCloseWriteMode( Buf );

/* Sending message to the network */
BufInitReadMode( Buf );
datalen = BufGetDataLen( Buf );
data = BufGetSeg( Buf, datalen );
agent_addr.sin_family = AF_INET;
agent_addr.sin_port = htons(161);
agent_addr.sin_addr.s_addr = inet_addr("192.168.0.20"); /*
ip address of any network entity, a good entity for test is
network printer */
i = sendto( manager_sock, data, datalen, 0, (struct
sockaddr *) &agent_addr, sizeof (agent_addr) );
BufCloseReadMode( Buf );
BufCloseBuf( Buf );
RELEASE_SIGNAL
END_IF_OUT_SIGNAL(snmp_request, "snmp_request")

/* Signal Init */
IF_OUT_SIGNAL(Init, "Init")
WSADATA wsdata;
WORD wVersionRequested;
wVersionRequested = MAKEWORD(2,2);
/* Registering in the socket library */
if( WSASStartup( wVersionRequested, &wsdata ) != 0 )
    exiterr("Init", "WSASStartup");

/* Creating manager socket */
if ( (manager_sock = socket( AF_INET, SOCK_DGRAM,
IPPROTO_UDP )) == -1 )
    exiterr("Init", "socket");
```

Creating the Transfer Syntax

```
manager_addr.sin_family = AF_INET;
manager_addr.sin_port = htons(162);
manager_addr.sin_addr.s_addr = INADDR_ANY;
/* Binding manager socket to <local IP>:162 */
if ( bind(manager_sock, (struct sockaddr *) &manager_addr,
sizeof (manager_addr)) != SOCKET_ERROR )
    exiterr("Init", "bind");
RELEASE_SIGNAL
END_IF_OUT_SIGNAL(Init,"Init")
/* Signal Close */
IF_OUT_SIGNAL(Close,"Close")
OUT_SIGNAL1(Close,"Close")
XENV_BUF(BufInitWriteMode(Buf));
OUT_SIGNAL2(Close,"Close")
XENV_BUF(BufCloseWriteMode(Buf));
RELEASE_SIGNAL
END_IF_OUT_SIGNAL(Close,"Close")
}
```

Encoder and Decoder function calls

As stated earlier, it is not necessary to auto-create the environment file. By copying another environment file or by writing it from scratch, you can customize the environment file for your needs. If you do so you must use the correct syntax of the encoding and decoding functions.

The syntax of the BER function calls is:

```
BER_ENCODE (Buffer, &Typenode, &Signalparameter)
BER_DECODE (Buffer, &Typenode, &Signalparameter)
```

The syntax of the PER function calls is:

```
PER_ENCODE ( Buffer, &Typenode, &Signalparameter)
PER_DECODE ( Buffer, &Typenode, &Signalparameter)
```

Example 5: Encoding and Decoding function calls

The following function calls are being used in the SNMP example:

```
BER_ENCODE(Buf, (tASN1TypeInfo *)&yASN1_Message,
(void *)&((yPDef_snmp_request *) (*SignalOut))->Param1));

BER_DECODE(Buf, (tASN1TypeInfo *)&yASN1_Message,
(void *)&((yPDef_snmp_reply *)SignalIn->Param1))
```

After the snmp message has been sent to an active network entity to port 161, the SDL system should change the state and start waiting for a reply with the values requested, see [Figure 197](#).

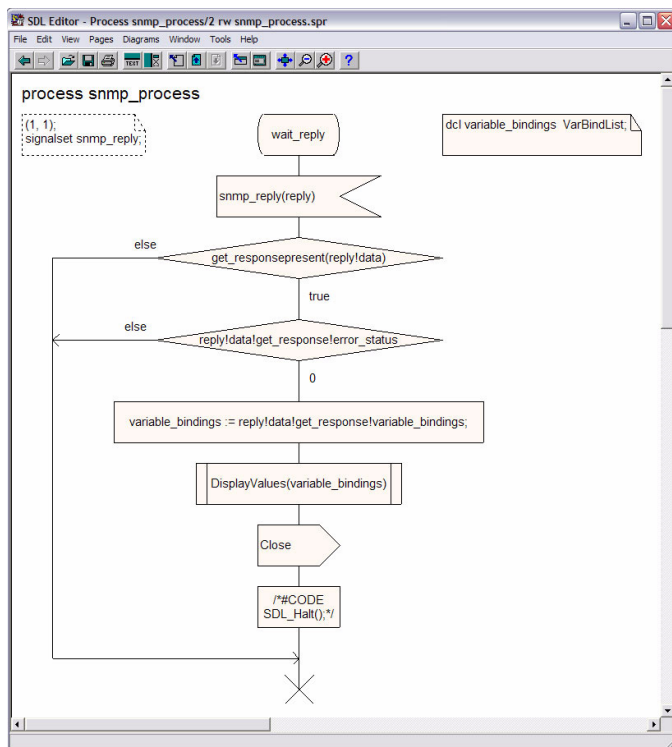


Figure 197: Waiting for the response

Decoding incoming signals

Before the SDL system can receive and use the information that is encapsulated in the incoming environment signals, a number of tasks must be performed in the environment file. Most of them are automatically generated to the environment file by the SDL Suite, but some must be handled manually.

The following list defines the steps involved in the decoding process. You must perform step 1 manually, while steps 2 through 4 are generated by the SDL Suite:

1. Extract the encoded information from the protocol-specific packet and transfer it to a data buffer. This should be implemented in C code in the environment file. In our case this is receive bytes from the socket and save them into the buffer.
2. Allocate memory for the signal structure. Special functions for that are automatically generated into the environment file by SDL Suite.
3. Call BER_DECODE function. The function is defined in the decoding library and handles the actual decoding process.
4. The decoded signal is sent to the SDL system. This is performed by the SDL_Output function.

The following code is part of the environment file for the Windows platform (`#include <WinSock.h>`) and displays the function that handles incoming signals.

```
/* Signal snmp_reply */
if ( manager_sock != -1 )
{
    datalen = recv( manager_sock, reply, sizeof(reply), 0 );
    if ( datalen == SOCKET_ERROR )
        exiterr("snmp_reply", "recv");
    else
    {
        BufInitBuf( Buf, bms_SmallBuffer );
        BufInitWriteMode( Buf );
        BufPutSeg( Buf, reply, datalen );
        BufCloseWriteMode( Buf );
        ERSetRule( Buf, er_BER | er_Definite );
        BufInitReadMode(Buf);
        IN_SIGNAL1(snmp_reply, "snmp_reply")
        BERDecode(Buf, (tASN1TypeInfo *)&yASN1_Message,
            (void *)&((ypDef_snmp_reply *)SignalIn)->Param1);
        IN_SIGNAL2(snmp_reply, "snmp_reply")
        BufCloseReadMode(Buf);
        BufCloseBuf(Buf);
    }
}
```

Generating Template Files - Targeting Expert

Follow the instructions below to generate environment files, type node files and the template makefile using the Targeting Expert.

1. From the *Generate* menu, select the *Targeting Expert* command. The SDL Targeting Expert window opens.
2. From the drop-down menu located above the Partitioning Diagram Model frame, select *Light Integrations* and the desired SDL to C Compiler. It is possible to use either Cadvanced or Cmicro. The pre-defined alternative specifies the type of compiler needed for the generation.
3. Select the *SDL to C Compiler* tab.
4. In the *General* box, select *Analyze/generate code*.
5. In the *Environment* box, select:
 - *Environment functions*
 - *Environment header file*
6. Select the *Communication* tab. In the *Coders* box, select the *Generate ASN.1 coder functions* check box.
7. Press the *Full Make* button. This generates the environment file.

Note:

Encode and decode calls are only generated if the *Coder functions...* option is enabled.

In your target directory, you will now find the generated files including:

- `<system_name>_env.c` (Cadvanced)
`env.c` (Cmicro)
This is the environment skeleton file.
- `<system_name>.ifc`
This is the environment header file.
- `<asn1module_name>_asn1coder.c`
`<asn1module_name>_asn1coder.h`
These files are the type nodes created by the ASN.1 utilities.

Editing the Generated Files - Targeting Expert

As the generated files only consist of skeleton functions, you must edit the files to suit the functionality of your application.

Note:

Make a habit of making a copy of the environment file after it has been edited. Otherwise the edits will be overwritten, if the file is re-generated by mistake.

1. Rename the environment file.
2. Edit the environment file `<system_name>_env.c` file according to your needs. In the skeleton file, macros are included but they are not defined. To define the required functionality, either create a `<system_name>_env.h` file and define all macros there, or replace the macros with the required code directly in the `<system_name>_env.c` file.
3. Save the environment file.

Notes:

- In order to transfer the information on the network, you must add socket commands to an appropriate header file.
- If you want to use more than one encoding scheme, for instance BER and PER, you must enter the appropriate encoding function calls in the environment file

Example 6: Environment functions - Cmicro

The following code is part of the environment file skeleton and displays the function that handles the out signals.

```
switch (xmk_TmpSignalID)
{
    case snmp_request :
    {
        /* BEGIN User Code */
        /* Use (yPDP_snmp_request)xmk_TmpDataPtr to access the signal's
parameters */
        /* ATTENTION: the data needs to be copied. Otherwise it */
        /* will be lost when leaving xOutEnv */
        /* This section can be used to encode outgoing data with the
selected coder functions.
        ** Please remove the comments and send the data with your
communications interface!
        ** (<SendViaCommunicationsInterface( data, datalen )> must be
replaced)
```

```

char* data;
int datalen;

BufInitWriteMode( Buf );
XENV_ENC( PER_ENCODE( Buf, (tASN1TypeInfo *)
&yASN1_z_RFC1157_SNMP_0_Message,
                                (void *) &((yPDef_snmp_request *)xmk_TmpDataPtr) -
>Param1));
    BufCloseWriteMode( Buf );
    BufInitReadMode( Buf );
    datalen = BufGetDataLen(Buf);
    data = BufGetSeg( Buf, datalen );
    <SendViaCommunicationsInterface( data, datalen )>;
    BufCloseReadMode( Buf );
    /*
    /* Do your environment actions here. */
    xmk_result = XMK_TRUE; /* to tell the caller that */
                        /* signal is consumed */
    /* END User Code */
    }
    break ;

case Init :
{
    /* BEGIN User Code */
    /* Do your environment actions here. */
    xmk_result = XMK_TRUE; /* to tell the caller that */
                        /* signal is consumed */
    /* END User Code */
    }
    break ;

case Close :
{
    /* BEGIN User Code */
    /* Do your environment actions here. */
    xmk_result = XMK_TRUE; /* to tell the caller that */
                        /* signal is consumed */
    /* END User Code */
    }
    break ;

default :
    xmk_result = XMK_FALSE; /* to tell the caller that */
                        /* signal is NOT consumed */
                        /* and to be handled by */
                        /* the Cmicro Kernel ... */
    break ;
}

```

Compiling Your Application

After you have created the transfer syntax, you are ready to compile and build your application, including the edited environment file and the template makefile. Follow the appropriate instructions as presented in:

- [“Using the edited files - Organizer” on page 321](#)
- [“Using the edited files - Targeting Expert” on page 322](#)

Using the edited files - Organizer

Please follow the instructions below:

1. Click the SDL system symbol in the Organizer
2. From the *Generate* menu, select the *Make...* command. The SDL Make window opens.
3. Change your options in the make dialog according to the following list:
 - Select *Analyze & generate code*
 - From the *Code generator* drop-down list, select *Advanced*
 - De-select *Generate environment header file*
 - De-select *Generate environment functions*
 - De-select *Generate ASN.1 coder*
 - Select the *Makefile* button
 - Select *Generate makefile and use template* and enter the generated template makefile . . . /<new_name>_env.tmp in the text field.
 - Select *Compile & link*
 - From the *Use standard kernel* drop-down list, select *Application*.

[Figure 198 on page 322](#) shows the Make dialog with the selected options.

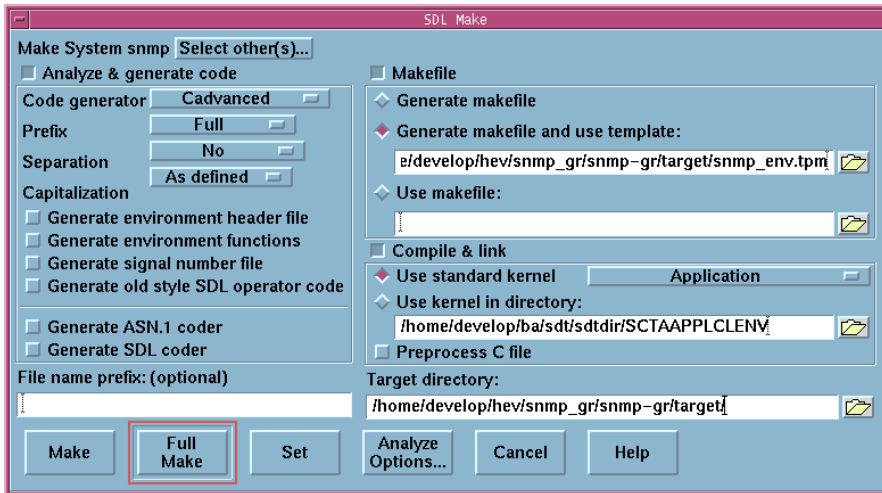


Figure 198: The Make dialog - Compiling

4. Specify your target directory where the generated files will be stored.
5. Press the *Full Make* button.

Using the edited files - Targeting Expert

Please follow the instructions below:

1. Select the *SDL to C Compiler* tab.
2. In the *Environment* box, de-select the *Environment functions* option and the *Environment header file* option.
3. Press the *Full Make* button.

Running Your Application

Your application acts as snmp agent. It connects to some network entity, sends request to snmp port and receives reply with the requested information.

If you have chosen the network printer as your network entity and implemented additional debug print, you might get a log like this for the sysDescr request:

```
snmp_request : Sending request #1001
snmp_request :
snmp_request : datalen = 40
snmp_request : BER encoded = 30 26 2 1 0 4 6 70 75 62 6c 69 63 a0 19 2 1 2 2 1
0 2 1 0 30 e 30 c 6 8 2b 6 1 2 1 1 1 0 5 0
snmp_request : Sending this message to SNMP agent 192.168.0.20:161...
snmp_request : 40 bytes sent
snmp_reply :
snmp_reply : Receiving data from socket...
snmp_reply : Received 131 bytes reply from SNMP agent
snmp_reply : BER encoded = 30 81 80 2 1 0 4 6 70 75 62 6c 69 63 a2 73 2 1 2 2 1
0 2 1 0 30 68 30 66 6 8 2b 6 1 2 1 1 1 0 4 5a 48 50 20 45 54 48 45 52 4e 45 54
20 4d 55 4c 54 49 2d 45 4e 56 49 52 4f 4e 4d 45 4e 54 2c 52 4f 4d 20 42 2e 32 35
2e 30 31 2c 4a 45 54 44 49 52 45 43 54 2c 4a 44 31 32 30 2c 45 45 50 52 4f 4d 20
56 2e 32 38 2e 37 32 2c 43 49 44 41 54 45 20 30 37 2f 32 30 2f 32 30 30 34
snmp_reply : Received in SDL process snmp_process
snmp_reply : Received 1 values for request # 1001

snmp_reply : Value #1 = HP ETHERNET MULTI-ENVIRONMENT,ROM
B.25.01,JETDIRECT,JD120,EEPROM V.28.72,CIDATE 07/20/2004
```

Appendix A

Example 7: The RFC1157-SNMP ASN.1 Module

```

RFC1157-SNMP DEFINITIONS ::= BEGIN
IMPORTS
    ObjectName, ObjectSyntax, NetworkAddress, IpAddress, TimeTicks
    FROM RFC1155-SMI;

-- top-level message

Message ::=
    SEQUENCE {
        version      INTEGER { version-1(0) }, -- version-1 for this RFC
        community    OCTET STRING,             -- community name, for example,
        "private" or "public"
        data          PDUs --ANY--             -- e.g., PDUs if trivial
        authentication is being used
    }

comPublic OCTET STRING ::= '7075626C6963'H
comPrivate OCTET STRING ::= '70726976617465'H

-- protocol data units

PDUs ::=
    CHOICE {
        get-request      GetRequest-PDU,
        get-next-request GetNextRequest-PDU,
        get-response     GetResponse-PDU,
        set-request       SetRequest-PDU,
        trap              Trap-PDU
    }

-- the individual PDUs and commonly used data types will be defined later

GetRequest-PDU ::= -- Used to retrieve a piece of management information
    [0] IMPLICIT
        SEQUENCE {
            request-id      RequestID,
            error-status     ErrorStatus,      -- always 0
            error-index      ErrorIndex,       -- always 0
            variable-bindings VarBindList
        }

GetNextRequest-PDU ::= -- Used iteratively to retrieve sequences of management
information
    [1] IMPLICIT
        SEQUENCE {
            request-id      RequestID,
            error-status     ErrorStatus,      -- always 0
            error-index      ErrorIndex,       -- always 0
            variable-bindings VarBindList
        }

GetResponse-PDU ::= -- Used by the agent to respond with data to requests from
the manager

```

Appendix A

```
[2] IMPLICIT
    SEQUENCE {
        request-id          RequestID,
        error-status         ErrorStatus,
        error-index          ErrorIndex,
        variable-bindings    VarBindList
    }

SetRequest-PDU ::= -- Used to initialize and make a change to a value of the
network element
[3] IMPLICIT
    SEQUENCE {
        request-id          RequestID,
        error-status         ErrorStatus,          -- always 0
        error-index          ErrorIndex,           -- always 0
        variable-bindings    VarBindList
    }

Trap-PDU ::= -- Trap == Asynchronous event report from the agent running on the
managed system (sent without being asked)
-- Used to report an alert or other asynchronous event about a
managed system.
-- Asynchronous event reports are called notifications in later
versions of SNMP
[4] IMPLICIT
    SEQUENCE {
        enterprise          OBJECT IDENTIFIER,      -- type of object
generating trap, see sysObjectID in [5]
        agent-addr          NetworkAddress,         -- address of object
generating trap
        generic-trap        INTEGER {              -- generic trap type
            coldStart(0),
            warmStart(1),
            linkDown(2),
            linkUp(3),
            authenticationFailure(4),
            egpNeighborLoss(5),
            enterpriseSpecific(6)
        },
        specific-trap        INTEGER,               -- specific code, present
even if generic-trap is not enterpriseSpecific
        time-stamp          TimeTicks,              -- time elapsed between
the last (re)initialization of the network entity and the generation of the trap
        variable-bindings    VarBindList           -- "interesting"
information
    }

-- request/response information

RequestID ::=
    INTEGER

ErrorStatus ::=
    INTEGER {
        noError(0),
        tooBig(1),
        noSuchName(2),
        badValue(3),
        readOnly(4),
        genErr(5)
    }
```

```
ErrorIndex ::=
    INTEGER

-- variable bindings

VarBind ::=
    SEQUENCE {
        name      ObjectName,
        value      ObjectSyntax
    }

VarBindList ::=
    SEQUENCE OF
        VarBind

END
```

Appendix A

Example 8: The RFC1155-SMI ASN.1 Module

```
RFC1155-SMI DEFINITIONS ::= BEGIN
EXPORTS -- EVERYTHING
    internet, directory, mgmt, experimental, private, enterprises,
    ObjectName, ObjectSyntax, SimpleSyntax,
    ApplicationSyntax, NetworkAddress, IpAddress,
    Counter, Gauge, TimeTicks, Opaque;

-- the path to the root

internet      OBJECT IDENTIFIER ::= { iso(1) org(3) dod(6) 1 }
directory     OBJECT IDENTIFIER ::= { internet 1 }
mgmt          OBJECT IDENTIFIER ::= { internet 2 }
experimental  OBJECT IDENTIFIER ::= { internet 3 }
private       OBJECT IDENTIFIER ::= { internet 4 }
enterprises   OBJECT IDENTIFIER ::= { private 1 }

-- names of objects in the MIB

ObjectName ::= OBJECT IDENTIFIER

-- syntax of objects in the MIB

ObjectSyntax ::= CHOICE {
    simple          SimpleSyntax,
    application-wide ApplicationSyntax
}

SimpleSyntax ::= CHOICE {
    number    INTEGER,
    string    OCTET STRING,
    object    OBJECT IDENTIFIER,
    empty     NULL
}

simple-value SimpleSyntax ::= empty : NULL

ApplicationSyntax ::= CHOICE {
    address    NetworkAddress,
    counter    Counter,
    gauge      Gauge,
    ticks      TimeTicks,
    arbitrary  Opaque
    -- other application-wide types, as they are
    -- defined, will be added here
}

-- application-wide types

NetworkAddress ::=
    CHOICE {
        internet    IpAddress
    }

IpAddress ::=
    [APPLICATION 0]          -- in network-byte order
    IMPLICIT OCTET STRING (SIZE (4))

Counter ::=
    [APPLICATION 1]
    IMPLICIT INTEGER (0..4294967295)
```

```
Gauge ::=
  [APPLICATION 2]
    IMPLICIT INTEGER (0..4294967295)

TimeTicks ::=
  [APPLICATION 3]
    IMPLICIT INTEGER (0..4294967295)

Opaque ::=
  [APPLICATION 4]          -- arbitrary ASN.1 value,
    IMPLICIT OCTET STRING  -- "double-wrapped"

END
```

Appendix A

Example 9: The RFCxxxx-MIBs ASN.1 Module

```
RFCxxxx-MIBs DEFINITIONS ::= BEGIN
    IMPORTS
        mgmt, enterprises, NetworkAddress, IpAddress, Counter, Gauge, TimeTicks
        FROM RFC1155-SMI;

mib          OBJECT IDENTIFIER ::= { mgmt 1 } -- 1.3.6.1.2.1

system       OBJECT IDENTIFIER ::= { mib 1 }  -- 1.3.6.1.2.1.1
interfaces   OBJECT IDENTIFIER ::= { mib 2 }
at           OBJECT IDENTIFIER ::= { mib 3 }
ip           OBJECT IDENTIFIER ::= { mib 4 }
icmp         OBJECT IDENTIFIER ::= { mib 5 }
tcp          OBJECT IDENTIFIER ::= { mib 6 }
udp          OBJECT IDENTIFIER ::= { mib 7 }
egp          OBJECT IDENTIFIER ::= { mib 8 }
printmib     OBJECT IDENTIFIER ::= { mib 43 } -- 1.3.6.1.2.1.43

-- sysDescr      OBJECT-TYPE
-- SYNTAX         DisplayString (SIZE (0..255))
-- ACCESS         read-only
-- STATUS         mandatory
-- DESCRIPTION    "A textual description of the entity. This value should include
the full name and version
--                identification of the system's hardware type, software
operating-system, and networking
--                software. It is mandatory that this only contain printable
ASCII characters."

sysDescr      OBJECT IDENTIFIER ::= { system 1 }
DisplayString ::= OCTET STRING

END
```

Tutorial: Using SDL-2000 features

This tutorial describes how to use the SDL-2000 features in the SDL Suite.

In order for you to fully take advantage of this tutorial, you should be familiar with the SDL Suite and the SDL Editor functionality.

This tutorial is independent from the Demon game example presented in the tutorial on the editors and the Analyzer.

You can find additional information on how to use the SDL-2000 features in [*“Working with Classes” on page 1944 in chapter 43, Using the SDL Editor, in the User’s Manual.*](#)

Purpose of this Tutorial

The purpose of this tutorial is to make you familiar with the SDL-2000 support in SDL Suite.

Introduction

Support in the SDL Suite

The SDL Suite supports the following SDL-2000 related features:

- Graphical design of data types (using class symbols)
- Textual algorithms
- Case sensitivity
- Operators without parameters
- Operators without return results.

Why SDL-2000?

The useful features in UML are included in SDL to facilitate the transport from UML to SDL.

Graphical Design of Data Types

SDL Suite offers the following graphical data type design features:

- [Class Symbols](#)
- [Association Lines](#)
- [Aggregation Lines](#)
- The *Browse & Edit Class* dialog, see [Editing a Diagram](#).
- The *Class Information* dialog, see [Viewing a Class](#).

Class Symbols

With class symbols you can graphically design data types. The SDL editor handles class symbols much the same way as other SDL symbols.

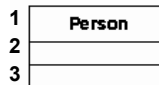


Figure 199 Class symbol

A class symbol has three separate text fields:

- name field (1)
- attribute field (2)
- operator field (3)

The properties of a class symbol can be set in the preference manager in the same way as for other SDL symbols, except for case sensitivity, see [“Case Sensitivity” on page 340](#). The size of the class symbol is adjusted to the size of the text, and you cannot change them manually. Class symbols cannot be collapsed or expanded.

Association and Aggregation Lines

There are two types of lines:

- Association lines

- Aggregation lines

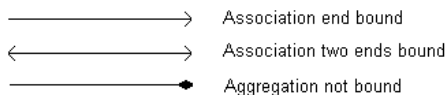


Figure 200

Association Lines

An association links two types using UML notation. The types are:

- Block types
- Process types
- Data types
- Interfaces

Association lines can be both redirected and bidirected. Single association lines (unidirectional associations) have a name and a role name. Double association lines (bidirectional associations) have a name and two role names.

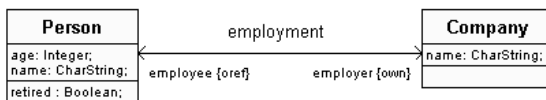


Figure 201: Example of an association line

Aggregation Lines

Aggregations use the same format as associations.

Aggregations are used to indicate that a class is a subset of another class or a part-of relationship, e.g. a steering wheel is a subset of or a part of a car. Aggregations can only be single directed.

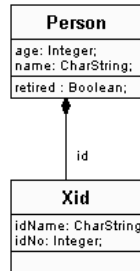


Figure 202: Example of an aggregation line.

Creating an SDL Structure

In this section you will create a small example and perform a number of actions to learn what possibilities and limitations there are in the SDL-2000 support in the SDL Suite.

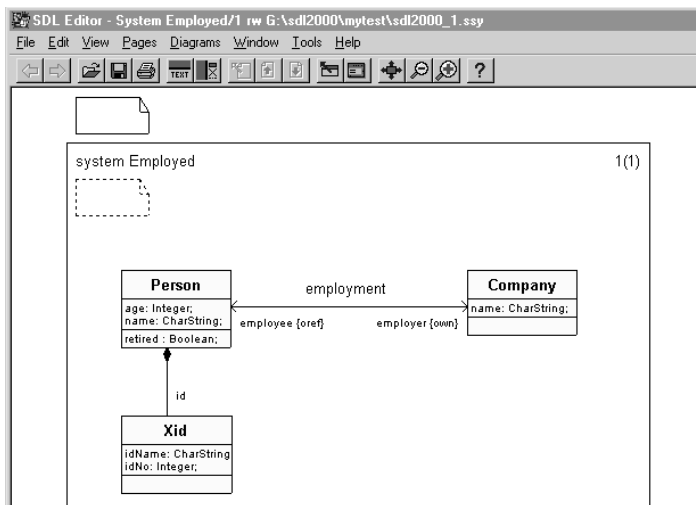


Figure 203:SDL diagram with class symbols

Working with Class Symbols

1. Start the SDL Editor.
2. Place a class symbol in the diagram.

You place symbols the same way as other SDL symbols, see [“Placing Block Reference Symbols” on page 59 in chapter 3, Tutorial: The Editors and the Analyzer](#). You can move symbols but not resize them.

3. Name the class symbol **Person**.

All class symbols with the same class name are treated as a single, merged class. See [“Limitations” on page 340](#) for more information on naming issues.

4. Fill out the attribute and operator fields.

5. Place two more class symbols in the diagram and name them **Company** and **xid**.
6. Fill out the attribute and operator fields.

Working with Lines

1. Click the class symbol **Person**. Two handles appears.
2. Select the square association handle and drag it to the class symbol **Company**. The editor draws a line while you are dragging. You drag lines the same way as lines in other SDL diagrams, see [“Drawing Channels between Blocks” on page 61 in chapter 3, Tutorial: The Editors and the Analyzer.](#)

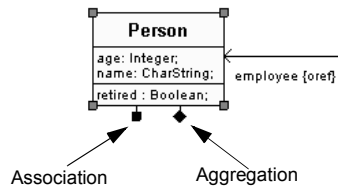


Figure 204

3. Click the mouse button when you have reached the class symbol **Company**. The line is connected at both ends.
4. Name the association line and the role.

The name of an association is often a verb phrase and the role often has a noun as a name.

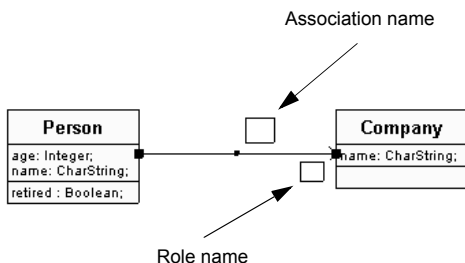


Figure 205

5. Bidirect the line by selecting *Bidirect* from the *Edit* menu.
6. Give the second role a name.
7. Select the diamond shaped aggregation handle and drag it to the class symbol `xid`.
8. Name the aggregation line and the role.

Moving Symbols and Lines

You move symbols and lines the same way as in other SDL diagrams, see [“Moving and Resizing Symbols” on page 61 in chapter 3, Tutorial: The Editors and the Analyzer.](#)

Saving the Diagram

1. Save the diagram by the menu choice *Save*, by clicking the *Save* button or by `<Ctrl> s`.

The file is saved with the extension `.ssy`.

Editing a Diagram

The *Browse & Edit Class* dialog lets you view and edit the complete definition of a class to ensure consistency between the attributes and operators.

The dialog consists of two parts. The above part is where you browse classes and all occurrences of each class. In the below part you can edit the name, attributes and operators of a class.

All symbols that belong to the class you select in the *Browse & Edit Class* dialog will be affected by the changes you make in the dialog. If you want to edit the content in only one class symbol, you have to select the single symbol and edit the content in that symbol. For detailed information on how to edit class information, see [“Browse & Edit Class Dialog” on page 1945 in chapter 43, Using the SDL Editor.](#)

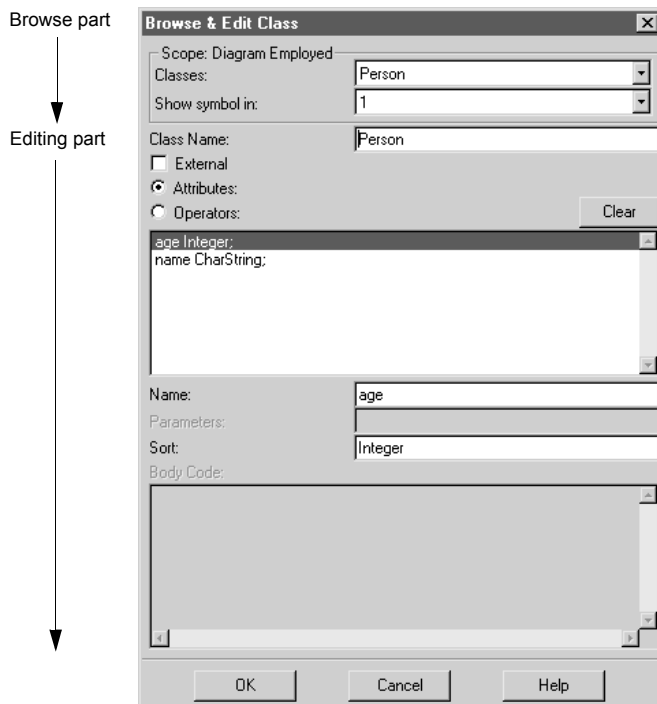


Figure 206: The Browse & Edit Class dialog

Case Sensitivity

Case sensitivity applies to class symbols and it cannot be turned off in the Preference Manager, though case sensitivity for other symbols in the SDL editor can be switched off.

Class sensitivity applies to both the name field and the attribute field as well as the operator field.

If two classes have the same name but different cases (for example `Person` and `person`), they will be treated as two separate classes in the *Browse & Edit Class* dialog and will **not** be merged into a single class when generating code.

For a detailed description, see [“Set-Case-Sensitive” on page 2488 in chapter 54, *The SDL Analyzer*](#).

Limitations

You can edit a class in this dialog when you have selected a single class symbol, but the dialog is not available if you select more than one class symbol.

If the class name contains incorrect syntax the dialog will not be displayed.

You cannot add syntax errors to your classes as the values you enter are syntactically checked.

Incorrect text in a symbol is not displayed in the dialog.

Edit the Diagram

1. Double-click the class `xid` or select the class `xid` and then select *Class..* on the *Edit* menu.

The *Browse & Edit Class* dialog is displayed with the class name and its attributes or operators filled in.

If there are more than one class with the same name, attributes or operators for all classes with the same name will be shown in the dialog.

Editing a Diagram

2. Select class `Person` from the *Classes* drop down menu. The content of the attributes and operators fields changes to that of the class `Person`.
3. Change the name of the class `Person` to **Employed** in the *Class Name* field and click *OK*.

The class name has changed to `Employed` in the diagram.

4. Open the *Browse & Edit Class* dialog again and select either of the radio buttons *Attributes* or *Operators* to edit or view the attributes or operators of the class `Employed`.

The list of the attributes or operators is displayed in the field below the radio buttons.

5. You can now edit the information in the *Name*, *Parameters*, *Sort* and *Body Code* fields.

The *Parameters* and *Body Code* fields are for operators only.

6. Click *OK*.

The dialog is closed and all appropriate class symbols in the current SDL diagram are updated.

Viewing a Class

The Class Information dialog lets you view the complete textual (PR) definition of a class in read-only format. If you want to edit the diagram, see [“Editing a Diagram” on page 339](#).

For detailed information on how to view class information, see [“Class Information” on page 1944 in chapter 43, *Using the SDL Editor*](#).

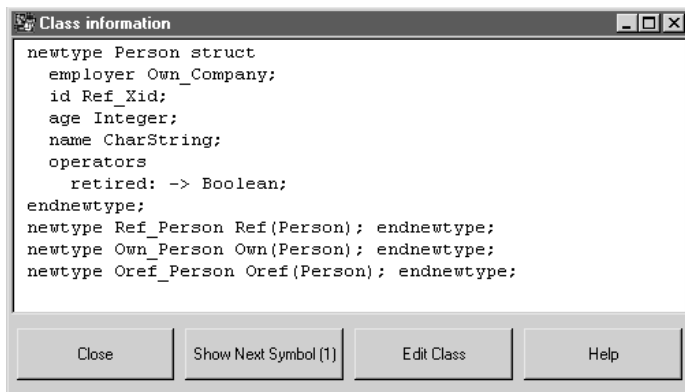


Figure 207: The Class Information dialog

Limitations

The dialog is not available if you select more than one class symbol.

View the Definition

1. Rename two or more classes with identical names (for testing of the Class information dialog).
2. Select one of the classes and then select *Class Information* on the *Window* menu.

The *Class Information* dialog, with the PR code for all classes with the same name, is displayed.

3. Select the *Show Next Symbol* button.

The next symbol, which is described by the line on which the cursor is placed, is displayed.

The number in brackets shows the number of symbols that are described by this line in the current diagram. You can browse through all symbols, one by one, with the *Show Next Symbol* button.

4. If you decide that you want to edit the class, select the *Edit Class* button and the *Browse & Edit Class* dialog for that class is displayed.

Textual Algorithms

Textual Algorithms

SDL Suite supports textual descriptions of algorithms. Algorithms can be expressed within a Task symbol, e.g. if-then-else, loops and decisions. For a detailed description of this feature, see [“Algorithms in SDL” on page 137 in chapter 3, *Using SDL Extensions*](#).

Operators without Parameters and Operators without Return Results

SDL Suite supports operators with no parameters or no return value. For example, an alternative to an assignment statement could be that an operator application statement invokes a non value returning operator.

For a detailed description of operators and parameters, see [“Operators” on page 81 in chapter 2, *Data Types*](#).

Limitations

The following limitations are worth noticing:

- Inheritance lines are not supported.
- Comments added to a class symbol will not appear in the generated PR code.
- CIF code generation for class symbols is not supported.
- If the same association has been drawn in several places, all instances have to be edited to change the association.
- Navigating from a class symbol is possible only by using *Tools>Navigate*, since double-clicking a class symbol will open the *Browse & Edit Class* dialog.

An Example of Using Class Symbols

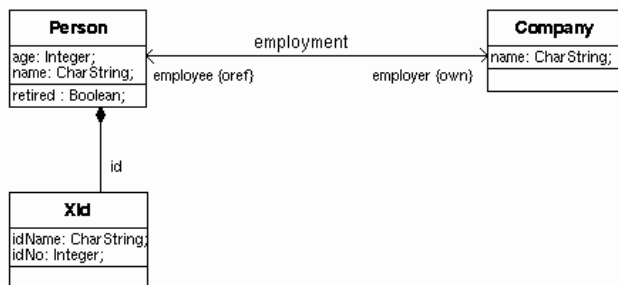


Figure 208 Graphical representation of a data structure

The operator definition must be added in the *Body Code* part of the *Browse & Edit Class* dialog. It might look like this:

```

operator retired returns Boolean { /*start of user defined
body code*/
return age > 64;
/* End of user defined body code */}
  
```

The above graph will then generate the following PR code.

```

newtype Person struct
  employer Own_Company;
  a_Company Ref_Company;
  id Ref_Xid;
  
```

An Example of Using Class Symbols

```
age Integer;
name CharString;
operators
  retired: -> Boolean;
  operator retired returns Boolean { /*start of user defined
body code*/
return age > 64
/* End of user defined body code */}
endnewtype;
newtype Ref_Person Ref(Person); endnewtype;
newtype Own_Person Own(Person); endnewtype;
newtype Oref_Person Oref(Person); endnewtype;
newtype XId struct
  idName CharString;
  idNo Integer;
endnewtype;
newtype Ref_XId ref(XId); endnewtype;
newtype Own_XId own(XId); endnewtype;
newtype Oref_XId oref(XId); endnewtype;
newtype Company struct
  employee Oref_Person;
  a_Person Ref_Person;
  name CharString;
endnewtype;
newtype Ref_Company Ref(Company); endnewtype;
newtype Own_Company Own(Company); endnewtype;
newtype Oref_Company Oref(Company); endnewtype;
```


Tutorial: Threaded Integration

The Threaded Integration model provides a powerful template for integrating generated C code with an operating system. In this tutorial, you will practice building applications using this model. You will get hands-on experience by building the SDL system “Mobile”, which is an implementation of a small GSM system.

This tutorial requires basic knowledge of the SDL Suite editors, the Organizer and the Targeting Expert. Brief knowledge of SDL and integration with operating systems will also help your understanding. Before working with this tutorial, it is recommended that you read through [chapter 64, *Integration with Operating Systems*](#).

To get the most out of this tutorial, read the entire chapter. As you progress, perform the exercises on your computer as described.

Introduction

The purpose of this tutorial is to make you familiar with the Threaded Integration model. After reading the tutorial, you should have a basic understanding of how to build a “threaded” executable and how to integrate it with external code.

This tutorial is designed as a guided tour through the build tools in the SDL Suite. You will get acquainted with the Deployment Editor and the Targeting Expert.

Note: Platform differences

This tutorial, and the others that are possible to run on both the Solaris, Linux and Windows platforms, are described in a way common to both platforms. In case there are differences between the platforms, this is indicated by texts like “on Solaris”, “on Unix”, “Windows only”, etc. When such platform indicators are found, please pay attention only to the instructions for the platform you are running on.

Normally, screen shots will only be shown for one of the platforms, provided they contain the same information for both platforms. This means that **the layout and appearance of screen shots may differ** slightly from what you see when running the SDL Suite in your environment. Only if a screen shot differs in an important aspect between the platforms will two separate screen shots be shown.

Prerequisites

Windows

- C compiler
- Resource compiler

Solaris / Linux

- C compiler
- Motif (Library for GUI widgets) version 2.1 or later

Description of Example System

The SDL System

Functional Description

The example system is a model of a GSM system. The implementation is restricted to the high-level layers of the GSM standard. The following functionality is implemented:

- User PIN code verification (PIN code is one digit only)
- Tracking of mobile phones through VLR and HLR databases
- Freedom of movement for mobile phones by base station switching
- Control of the IMEI code of a mobile phone
- Billing service to keep track of elapsed time and cost for phone calls

Implementation Description

The system is implemented in a modular way. Each functional entity is implemented as a block type. The block types are organized in a package, called “GSM”.

The SDL system uses the GSM package and instantiates its block types. The “Mobile” system has four MobileStation block instances, four BaseTransceiverStation block instances and two MobileSwitchingCenter block instances. The MobileStation instances are named as mobile phone owners. The names are Marie, John, ParisPizza and Lyon-Pizza. The BaseTransceiverStation instances are named Lyon11, Lyon12, Paris11 and Paris12, indicating their location.

The Target Application

Deployment

The Mobile system is partitioned into five executable files. Each MobileStation block instance is built as an executable and the remaining block instances (Proxy, BaseTransceiverStation, BaseStationController, MobileSwitchingCenter and Database) execute together in one executable.

The Graphical User Interface

A graphical user interface (GUI) is provided for the MobileStation block. It is delivered as C source code to be compiled and linked when building each of the four mobile phone executable files. The interface resembles that of a typical mobile phone. A simple menu system, containing a phone book and a billing report, is provided. The user can switch base station from the option menu in the bottom of the window. The GUI is shown in [Figure 209](#).



Figure 209: The Mobile Phone GUI

TCP/IP Communication

The executable files of the target application communicate by sending signals via TCP/IP. The communication is handled by the SDL Suite TCP/IP communication module, which is delivered as C source code. It is compiled and linked into each of the executables.

To be able to locate the receiver of a signal, the TCP/IP module needs the host name and TCP port number of the receiving executable. This information must be supplied through a routing function, which is manually implemented in C.

The mobile system is delivered with default routing functions. This will enable you to run all executables on your computer without giving any routing information manually.

[Figure 210](#) shows a UML component diagram of the executable files in the deployed mobile system. Each executable file sets up a TCP/IP server, listening on a specific TCP port number. This number is used by re-

Description of Example System

note executables for addressing SDL signals. The TCP/IP server is illustrated by the interface on each component. The dashed arrows indicate a flow of signals from one component to another.

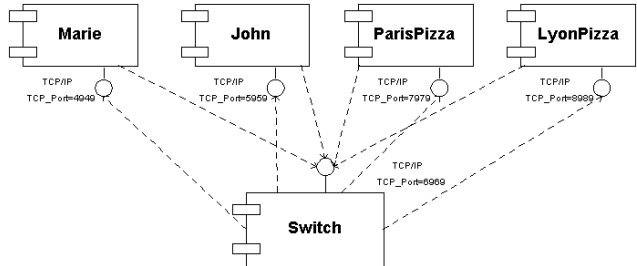


Figure 210: A Component Diagram Showing the Deployed Mobile System

Preparations

Copy the Example System

In order to allow experimentation, you should copy the example system from the SDL Suite installation into a working directory of your preference:

UNIX

Copy all files from the directory
`$telelogic/sdt/examples/mobile`
to a working directory, e.g. `~/mobile`.

Windows

Copy all files from the directory
`C:\IBM\Rational\SDL_TTCN_Suite6.3\sdt\examples\mobile`
to a working directory, e.g.
`C:\IBM\Rational\SDL_TTCN_Suite6.3\work\mobile`.

Open the System

Open the system file `Mobile.sdt` from the Organizer.

Drawing a Deployment Diagram

In order to build a threaded executable, you must supply information about the threads of the application. For each thread, you must specify which SDL instance sets that should run inside it. Finally, you may want to partition your SDL system into several executable files.

This information is modeled using Deployment Diagrams. These are edited using the Deployment Editor. The Deployment Diagram provides a way to model deployment of SDL systems independently of the target platform.

What You Will Learn

- To start the Deployment Editor
- To deploy an SDL system onto executable files and threads

Starting the Deployment Editor



- Double-click the deployment symbol “network_depl” in the Organizer. The Deployment Editor is launched with the selected diagram.





Deploying an SDL System

The Deployment Diagram features:

- Partitioning of SDL systems
- Organization of SDL instance sets within executable files
- Organization of SDL instance sets within threads (Threaded Integration only)

A deployment diagram has five symbols: the node, the component, the thread, the object and the comment symbols. Each symbol is described in [Table 1](#).

Table 1 Deployment Diagram Symbols

Symbol	Name	Description
	Node	A computational resource, i.e. a computer
	Component	An executable file which contains SDL instance sets
	Thread	A point of execution. The symbol is used for threaded integrations only
	Object	An SDL instance set, i.e. a system, block or process instance set

The symbols are connected using compositions. When a symbol is selected in a diagram, a handle is shown at its bottom. Click the handle to create a composition link. Click the symbol you wish to connect to.

The name of each symbol can be edited from the diagram area. Some of the symbols contain additional information, which is edited from the *Symbol Details* dialog box.

The *network_depl* deployment diagram is shown in [Figure 211](#).

Drawing a Deployment Diagram

network_depl

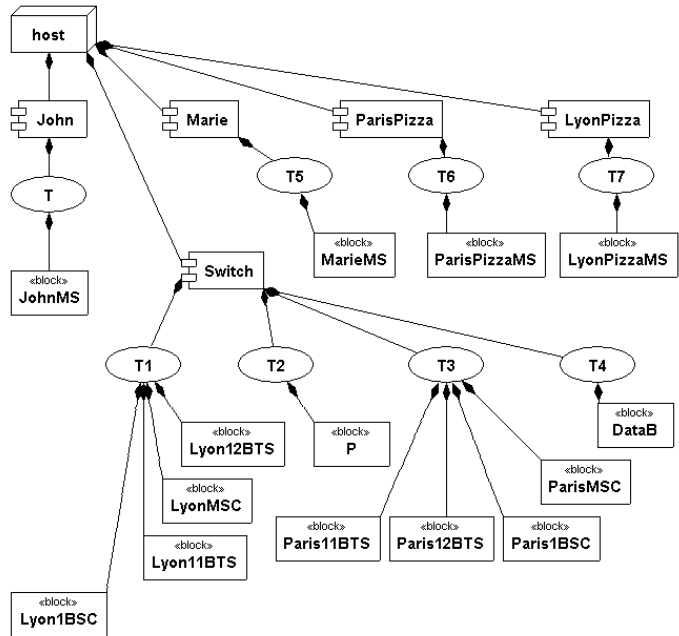


Figure 211: The network_depl Deployment Diagram

The diagram shows five components, each representing an executable file. All components are attached to one node. Each component has one or many threads. Each thread has one or many objects attached to it, reflecting the SDL instance sets that execute inside it.

For instance, the “JohnMS” component contains one thread, “T”. The thread has one object attached to it, called “JohnMS”. The object reflects an SDL instance set in the network SDL system. Look at the Organizer window and compare the block instance sets shown in the system with the objects in the deployment diagram.

The Symbol Details Dialog Box

The dialog box is opened in one of the following ways:

- Double-click a diagram symbol

- Right-click a diagram symbol. Select *Symbol Details...* in the pop-up menu.

Component

Double-click the component named “John”. The symbol details dialog box will appear. The dialog box is shown in [Figure 212](#).

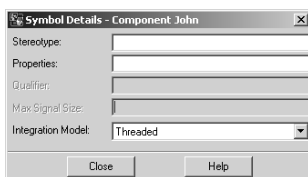


Figure 212: The Symbol Details dialog box for Component “John”

The integration model can be selected from a drop-down list in the dialog box. The selected integration model controls the code generation parameters for the executable file that is generated from the component. Three integration models can be selected: Light, Threaded and Tight.

For the “John” component, the selected integration model is “Threaded”. The information filled in the other text boxes is not used for code generation.

Thread

Click the thread symbol “T”. The dialog box will change its content to reflect that of the thread. The new appearance of the dialog box is shown in [Figure 213](#).

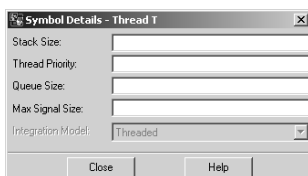


Figure 213: The Symbol Details Dialog Box for Thread “T”

For a thread, four parameters can be set for performance-tuning of the generated executable:

Drawing a Deployment Diagram

- Stack Size
- Thread Priority
- Queue Size
- Max Signal Size

The values of these parameters are specific to the target operating system. If no values are given, default values are used. All threads in the *network_depl* deployment diagram use default values.

Object

Click the object symbol “JohnMS”. The dialog box will change its content to reflect that of the object. The new appearance of the dialog box is shown in [Figure 214](#).

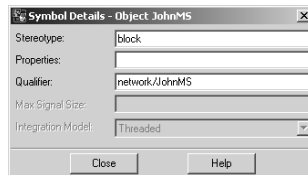


Figure 214: The Symbol Details Dialog Box for Object “JohnMS”

For an object, two parameters are mandatory to fill in:

- Stereotype
- Qualifier

The stereotype is the type of SDL instance set. The possible values are “system”, “block” and “process”. “JohnMS” is a block, which is reflected in the stereotype text box.

The qualifier is used to identify the SDL instance set. Locate the “JohnMS” block in the Organizer window. The block instance set is located directly under the “network” system. This renders the qualifier “network/johnMS”.

The *network_depl* diagram is complete. The next step is to build executable files from the diagram.

More information on the Deployment Editor is available in [chapter 40, *The Deployment Editor, in the User's Manual*](#).

Using the Targeting Expert

What You Will Learn

- To start the Targeting Expert
- To build executable files from a deployment description
- To configure C code generation from the Targeting Expert GUI
- To configure your compiler and linker from the Targeting Expert GUI

Starting the Targeting Expert

The deployment diagram should be used as input to the Targeting Expert.

Right-click the *network_depl* diagram symbol in the Organizer and select *Targeting Expert* from the pop-up menu.

The deployment diagram will be analyzed. If any errors are found, the Organizer Log pops up and shows an error message. You can locate the error in the deployment diagram by clicking on the “Show error” button in the Organizer Log toolbar.

The Targeting Expert is launched.

Selecting Target Platform

The initial appearance of the Targeting Expert window is shown in [Figure 215](#).

Using the Targeting Expert

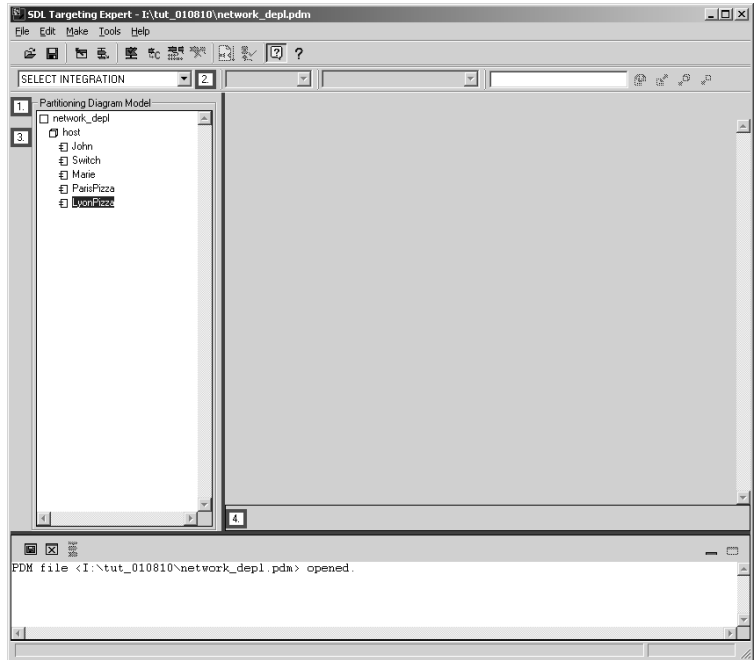


Figure 215: The Targeting Expert When First Opened

The left part of the window, called *Partitioning Diagram Model*, shows a filtered view of the deployment diagram. The nodes and components are represented.

For each component, do the following:

1. Select the component in the *Partitioning Diagram Model* window.
2. Click the drop-down list containing "SELECT INTEGRATION". A menu with available integrations is opened. As the component has "Threaded" as integration (selected in the Deployment diagram), this is the only available selection.

3. Select the platform in the sub-menu.
 - **Windows:** Select *Win32 Threaded (CAdvanced)*
 - **Solaris:** Select *Solaris Threaded (CAdvanced)*
 - **Linux:** Select *Linux Threaded (CAdvanced)*
4. A dialog-box will pop up, asking if the SDL system should be automatically generated. Click *Yes*.

The Targeting Expert window now looks as in [Figure 216](#).

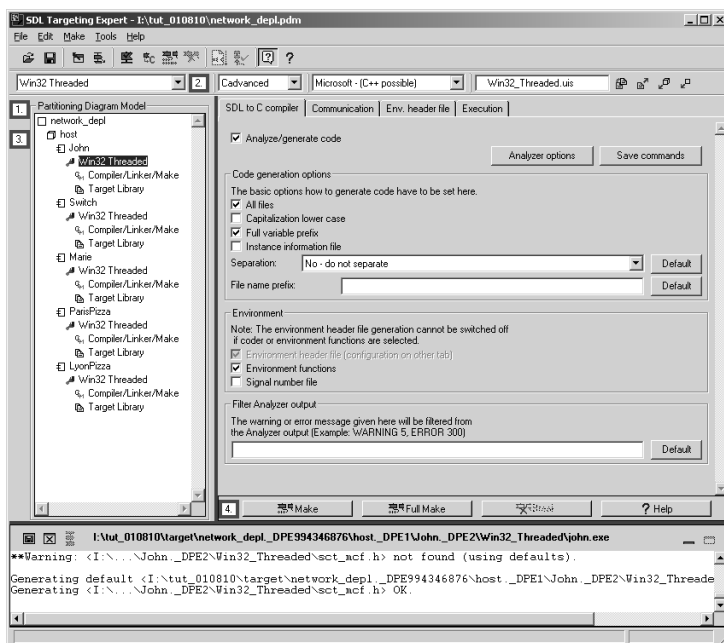


Figure 216: Components with Selected Integration Models

Configuring C Code Generation

The Targeting Expert GUI for C code generation has four tabs. These are *SDL to C Compiler*, *Communication*, *Environment Header File* and *Execution*. All these tabs contain widgets for configuring the code generation for an executable.

Using the Targeting Expert

Each of the executable files that will be built have communication through external code. All executables use the TCP/IP communication module and the mobile phone executables (John, Marie, ParisPizza and LyonPizza) use a GUI. The external code is connected to the SDL system through environment functions. The generation of environment functions must be configured manually.

The Targeting Expert features a wizard for easy configuration of the TCP/IP module. When activating the wizard, all the necessary environment functions will be generated by default.

The TCP/IP wizard dialog box is shown in [Figure 217](#).

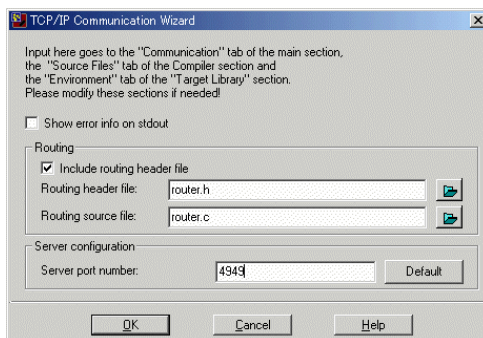


Figure 217: The TCP/IP Wizard

The TCP/IP module needs routing information to send signals to the correct recipient. A C header file with the routing function declaration must be included. The routing function definition must be given in a C source file. The TCP port number for incoming signals must also be given. This number is used by routing functions in remote executables sending signals to this executable.

For each of the components, do the following:

1. Click the name of the integration in the left sub-window.
 - **Windows:** Click *Win32 Threaded*
 - **Solaris:** Click *Solaris Threaded*
 - **Linux:** Click *Linux Threaded*

2. Select the *Communication* tab.
3. Click in the *TCP/IP* check box in the *Signal Sending* group. The TCP/IP Wizard is opened.
4. Click in the check box *Include routing header file*.



5. Click the file button to the right of the header file text box. A file selection dialog box is opened.
6. Select the routing header file in your working directory.
 - If you build the John, Marie, ParisPizza or LyonPizza executable, select *router.h*.
 - If you build the Switch executable, select *switchrouter.h*.



7. Click *Open*.
8. Click the file button to the right of the source file text box. A file selection dialog box is opened.
9. Select the routing source file in your working directory.
 - If you build the John, Marie, ParisPizza or LyonPizza executable, select *router.c*.
 - If you build the Switch executable, select *switchrouter.c*.
10. Click *Open*.
11. Enter the server port number of the executable you are building. Enter a port number according to [Table 2](#).

Table 2 TCP Server Port Numbers for the Executables

Executable	TCP Port Number
Marie	4949
John	5959
ParisPizza	7979
LyonPizza	8989
Switch	6969

12. Click *OK* in the TCP/IP wizard dialog box.

Using the Targeting Expert

The code generation configuration is now finished.

Compiling and Linking

The Targeting Expert *Compiler/Linker/Make* section contains six tabs: *Compiler*, *Source Files*, *Additional Compiler*, *Linker*, *Library Manager* and *Make*. The section is shown in [Figure 218](#).

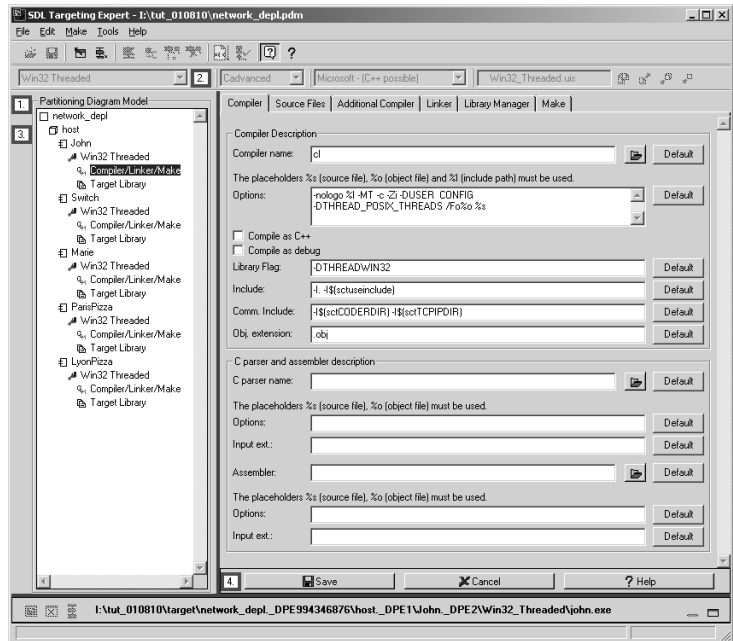


Figure 218: The Targeting Expert Compiler/Linker/Make Section

Compiling and Linking the *Switch* Executable

To configure the compilation and linking of the *Switch* executable, do the following:

1. Click *Compiler/Linker/Make* for *Switch* in the Partitioning diagram Model window.
2. In the *Include* text box, add `-I../..../` After the addition, the text box will look as shown in [Figure 219](#).

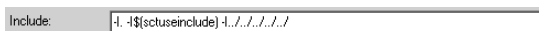


Figure 219: The Include Text Box with an Additional `-I` flag

3. Click the *Save* button.
4. Click *Target Library* for *Switch* in the Partitioning diagram Model window.
5. In the *Kernel* tab, click in the following check boxes:
 - *Show errors on stdout*. Setting this option renders log messages if errors are encountered during execution.
 - *Text trace*. Setting this option renders textual traces on stdout during execution.
6. Click the *Save* button.
7. Click the *Full Make* button. The SDL system is analyzed, code is generated and a makefile is generated and executed.

You will now have an executable file named *switch*. The file extension will be `.exe` on Windows and `.set` on Unix.

Compiling and Linking the *MobileStations*

To configure the compilation and linking of the *MobileStation* executables, do the following:

1. Click *Compiler/Linker/Make* for *MobileStation* in the Partitioning Diagram Model window.
2. In the *Include* text box, add `-I<your working directory>`, e.g. `-I/home/mobile` . After the addition, the text box will look as shown in [Figure 219](#).

3. Add the following flags in the *Library Flag* text box:

- `-DXMAIN_NAME=SDL_Main`
- `-DXTENV_INC="<gui.h>"`
- `-DMARIE, -DJOHN, -DPARISPIZZA` or `-DLYONPIZZA`, depending on which executable you are configuring

The `XMAIN_NAME` flag is used to rename the main function in the generated SDL system. A main function is provided in the GUI source code. The SDL main function, renamed `SDL_Main`, will be started in a thread from the new main function.

`XXTENV_INC` is a flag for using environment code together with the TCP/IP module. The *gui.h* file contains definitions of some macros in the generated environment file.

4. Click the *Source Files* tab. A list of the external files to compile is shown.
5. Click the Add button. A dialog box is shown.
6. Select *gui.c* in the working directory. Click *Open*. The file is added to the file list.
7. **Windows only:** The GUI must be compiled using a resource compiler.
- Click the *Additional Compiler* tab.
 - Enter *rc* in the *Compiler Name* text box.
 - Enter *-l 0x41d %I -fo %o %s* in the Options text box.
 - Add the file *gui.rc* to the list of files to compile by clicking the *Add* button and select *gui.rc* from your working directory.
 - Enter *.res* in the *Object Extension* text box.
 - Click on the *Make* tab.
 - Change from Microsoft *nmake* (using temporary response file) to Microsoft *nmake* in the *Make tool* drop down list.
8. Click the *Linker* tab.

9. Enter the following in the *Options* text box:
 - **Windows:** Change *-subsystem:console* to *-subsystem:windows*
 - **Unix:** Add *-lXm -lXi -lXi1* between *-lpthread* and *-L/usr/lib*.
10. Click the *Save* button.
11. Click the *Full Make* button. The SDL system is analyzed, code is generated and a makefile is generated and executed.

You will now have an executable file. The file extension will be *.exe* on Windows and *.sct* on Unix.

The Target System

The generated executable files are located in a directory structure created by the Targeting Expert. The Targeting Expert uses the target directory given in the Organizer as root.

In the mobile system, *target* is given as target directory. The generated target directory structure is shown in [Figure 220](#).

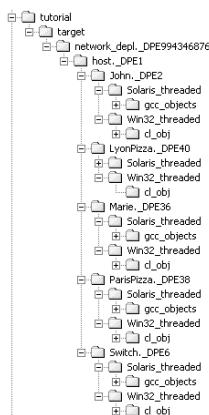


Figure 220: The Generated Target Directory Structure

The executable files are located in the platform-specific directories (*Win32_threaded* and *Solaris/Linux*)

_threaded, respectively). The object files are located in subdirectories of the platform directories.

Running the System

What You Will Learn

- To run the executable files generated from Targeting Expert
- Use the mobile system

An Overview of the System

[Figure 221](#) shows the run-time architecture of a MobileStation executable.

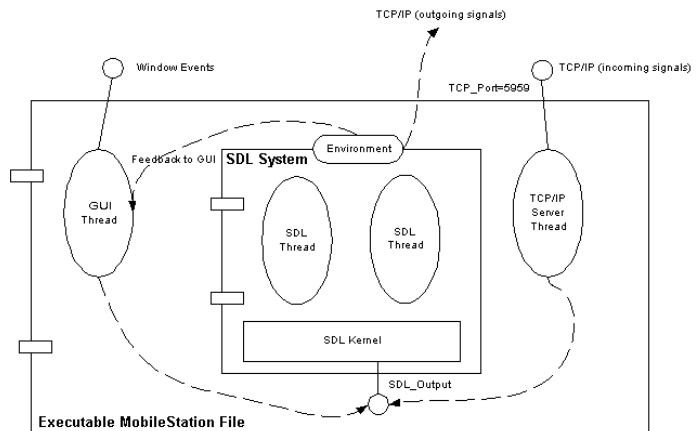


Figure 221: Run-time architecture of a MobileStation Executable

The SDL system interacts with the outside world through its environment. The GUI message loop is run in a thread of execution of its own. The TCP/IP server thread executes the same way. The environment threads interact with the SDL system by inserting signals. This is done by calling the SDL kernel function `SDL_Output`.

When a signal is sent from the SDL system to the environment, it is interpreted either as GUI feedback or is sent to an external receiver via TCP/IP.

Using the System

Start the mobile system in the following order:

1. Start the Marie, John, ParisPizza and LyonPizza executables. A GUI will pop up for each of them.
2. Start the Switch executable. Switch will initialize the MobileStation executables by sending signals. You will see that the *On* buttons on the mobile phone windows become enabled.

To make a call, do the following:

1. Click the *On* button on the MobileStations. You will be prompted to enter a PIN code. The correct PIN codes are shown in [Table 3](#).

Table 3 MobileStation PIN Codes

Executable	PIN Code
Marie	1
John	2
ParisPizza	3
LyonPizza	4

2. Click *OK*. “PIN OK” will be displayed. You are now ready to make a call.
3. Click *OK*. The menu system of the GUI is activated. Select “Phone Book” by pressing the “<” and “>” buttons and click *OK*.
4. Select a name from the list and click *OK*. If the receiving MobileStation is on, its display will show “Incoming Call”.
5. Click *OK* on the receiving MobileStation. A call is established.
6. Click *OK* on the calling party and the receiving party when you wish to finish the conversation.

The system has more features not described in this manual. Experiment to discover its secrets!

Running the System

The tutorial is finished. To get more detailed information about the Threaded integration and the TCP/IP module, please read [chapter 64, *Integration with Operating Systems, in the User's Manual*](#).

A

Abstract Syntax Notation One (ASN.1): [17](#)
AccessControl System (example): [272](#)
Aggregation between classes (OM notation): [13](#)
Aggregation lines: [333](#)
ASN.1: [17](#)
Association lines: [333](#)
Associations between classes (OM notation): [13](#)

B

Batch facilities: [25](#)
Behavior tree, example: [199](#)
Block (SDL concept): [5](#)

C

Cadvanced SDL to C Compiler (SDL Suite): [29](#)
Cbasic SDL to C Compiler (SDL Suite): [29](#)
CIF format: [33](#)
Class notation (OM notation): [12](#)
Class symbols: [333](#)
 an example: [344](#)
 case sensitivity: [340](#)
 edit a diagram: [339](#)
 limitations: [344](#)
 viewing a class: [342](#)
Cmicro Library: [270](#)
Command
 next-transition: [142](#)
 output-via: [143](#)
 set-gr-trace: [141](#)
 Set-Trace: [141](#)
 show-next-symbol: [142](#)
Control unit file: [35](#)

D

Data types in SDL: [7](#)
Demon Game system (example): [41](#), [129](#)
DP symbols
 Node: [354](#)
 Thread: [354](#)

G

Generalization (OM notation): [12](#)

H

High-level MSC (HMSC): [10](#)
HMSC (MSC concept): [10](#)
HMSC diagrams: [33](#)

I

Inheritance of classes (OM notation): [12](#)

Instance (MSC concept): [9](#)

Integration

Bare: [271](#)

Light: [271](#)

Tight: [271](#)

Integration mechanism in SDL & TTCN Suite: [23](#)

L

Licenses: [25](#)

Link file: [35](#)

M

Message Sequence Charts (MSC): [9](#)

Messages (MSC concept): [9](#)

MSC: [9](#)

MSC diagrams: [33](#)

MSC instance (MSC concept): [9](#)

MSC language: [9](#)

MSC message (MSC concept): [9](#)

MSC reference (MSC notation): [9](#)

MSC/GR: [11](#)

MSC/PR: [11](#), [33](#)

Multiplicity (OM notation): [14](#)

N

Notations

Abstract Syntax Notation One (ASN.1): [17](#)

Message Sequence Charts (MSC): [9](#)

Object Model (OMT/UML): [12](#)

Specification and Description Language (SDL): [3](#)

State Chart: [15](#)

Tree and Tabular Combined Notation (TTCN): [17](#)

O

Object Model diagrams: [34](#)

Object Model notation: [12](#)

Object notation (OM notation): [14](#)

Object-orientation in SDL: [8](#)

OM diagrams: [34](#)

OM notation: [12](#)

OMT notation: [12](#)

P

PId (SDL concept): [7](#)

Preferences: [46](#)

Displaying and Changing: [46](#)
Help: [47](#)
Saving: [49](#)
Setting the default printer: [48](#)
Setting the drawing area size: [49](#)
Procedure (SDL concept): [5](#)
Process (SDL concept): [5](#)
Process Diagram, creating: [96](#)
R

Remote procedure (SDL concept): [6](#)
S

SC diagrams: [34](#)
SC notation: [15](#)
SDL: [3](#)
SDL diagrams: [32](#)
SDL language: [3](#)
SDL Simulator
Find dynamic errors: [153](#)
Restart: [148](#)
Send signal: [143](#)
Starting and Generating: [137](#), [353](#)
Trace value: [155](#)
Using breakpoints: [163](#)
View internal status: [148](#)
SDL Suite overview: [23](#)
SDL Suite, starting: [23](#), [43](#)
SDL Target Tester: [289](#)
SDL, Creating a structure: [51](#)
SDL/PR: [33](#)
Service (SDL concept): [5](#)
Signal (SDL concept): [6](#)
Specialization (OM notation): [12](#)
Specification and Description Language (SDL): [3](#)
Start notation (SC notation): [16](#)
State Chart diagrams: [34](#)
State Chart notation: [15](#)
State notation (SC notation): [15](#)
System (SDL concept): [5](#)
System diagram, checking: [74](#)
System diagram, print: [72](#)
System file: [34](#)
T

Termination notation (SC notation): [16](#)
Test case (TTCN concept): [17](#)
Test suite (TTCN concept): [17](#)

Text documents: [34](#)
Tool overview (SDL Suite and TTCN Suite): [26](#)
Tool overview (SDL Suite): [27](#)
Transition (SC notation): [15](#)
Tree and Tabular Combined Notation (TTCN): [17](#)
TTCN: [17](#)
TTCN language: [17](#)
Types in SDL: [8](#)
U

Unified Modeling notation: [12](#)
Z

Z.100: [3](#)
Z.120: [9](#)