Chapter

6

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 154 and Figure 155.

To use the new version:

1. Make a new empty directory `sdl92` of your own (under `~/demongame` **on UNIX**, and under `C:\Telelogic\SDL_TTCN_Suite4.5\work` **in Windows**).
2. Copy all files in the directory `$telelogic/sdt/examples/demongame/sdl92/process_type` **(on UNIX)**, or `C:\Telelogic\SDL_TTCN_Suite4.5\sdt\examples\demongame\sdl92\process_type` **(in Windows)**, to this new directory.

Note: Installation directory

On UNIX, the Telelogic Tau 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 Telelogic Tau environment at your site for instructions on how to set this variable correctly.

In Windows, the Telelogic Tau installation directory is assumed to be `C:\Telelogic\SDL_TTCN_Suite4.5` throughout this tutorial. If you cannot find this directory on your PC, you should ask your system manager or the person responsible for the Telelogic Tau 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.

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.

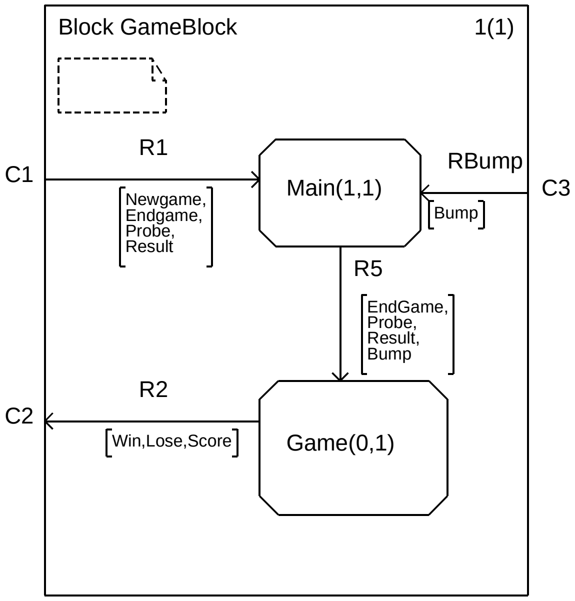


Figure 154: The block GameBlock, redesigned

The exact layout of your diagrams may differ slightly from the above.

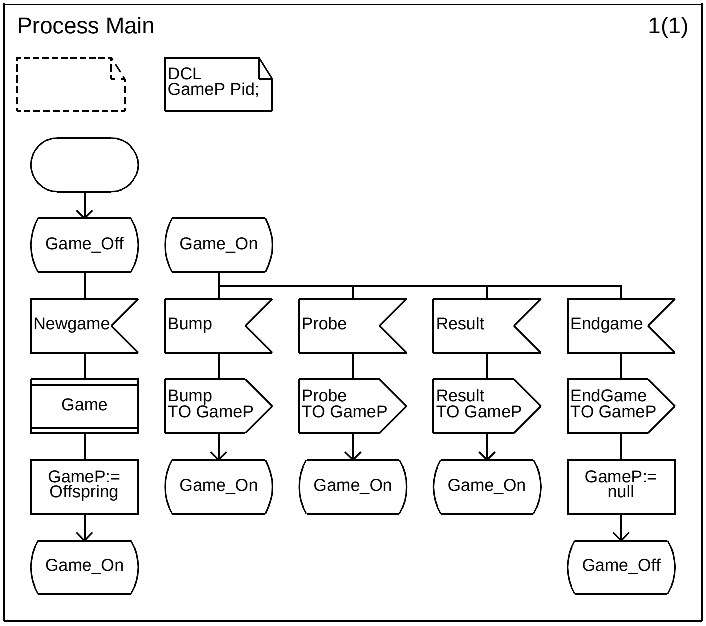


Figure 155: The process Main, redesigned

The exact layout of your diagrams may differ slightly from the above.

Creating a Process Type from a Process

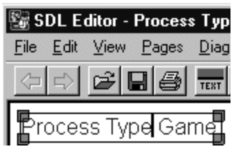
What You Will Learn

- 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”.

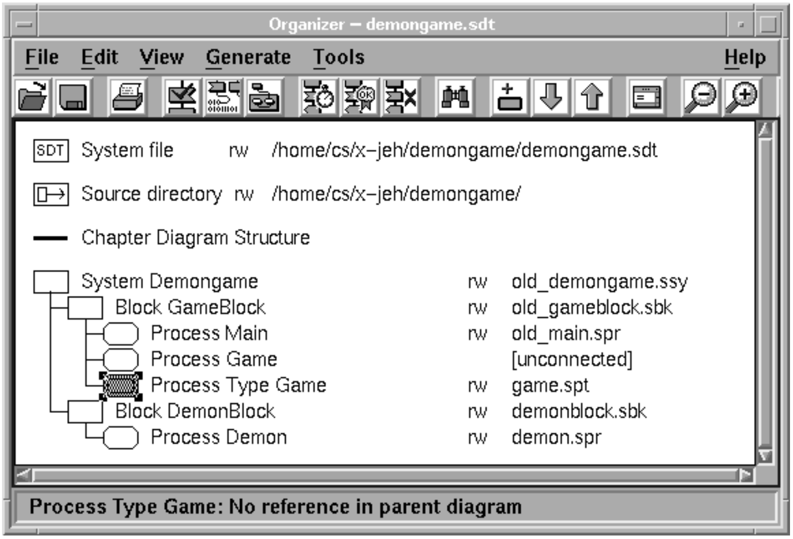


Process Type Game

Figure 156: 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.



```
SDT  System file        rw   /home/cs/x–jeh/demongame/demongame.sdt
     Source directory   rw   /home/cs/x–jeh/demongame/
     Chapter Diagram Structure
     System Demongame                         rw   old_demongame.ssy
       Block GameBlock                        rw   old_gameblock.sbk
         Process Main                         rw   old_main.spr
         Process Game                              [unconnected]
         Process Type Game                    rw   game.spt
       Block DemonBlock                       rw   demonblock.sbk
         Process Demon                        rw   demon.spr
```

```
Process Type Game: No reference in parent diagram
```

Figure 157: 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:

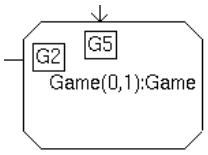


Figure 158: Naming the connection points

6. Also add a process type reference symbol with the name Game. The block diagram should now look like this:

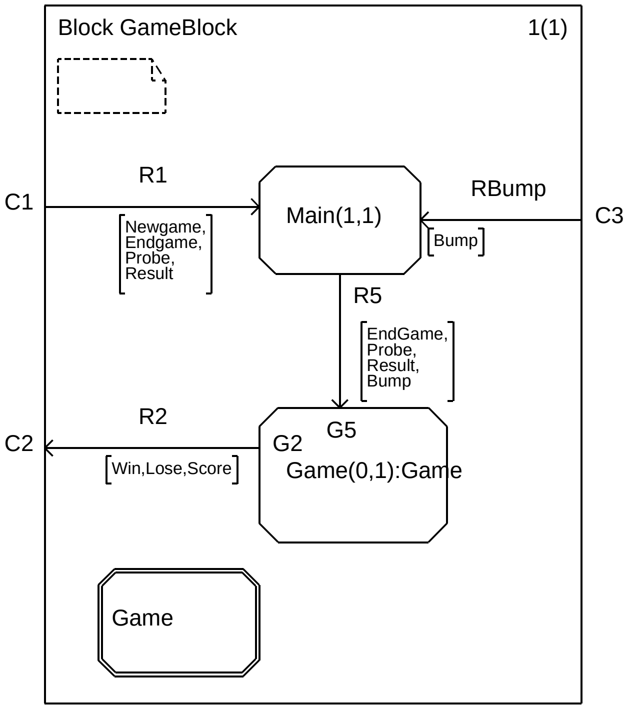


Figure 159: The resulting block

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.

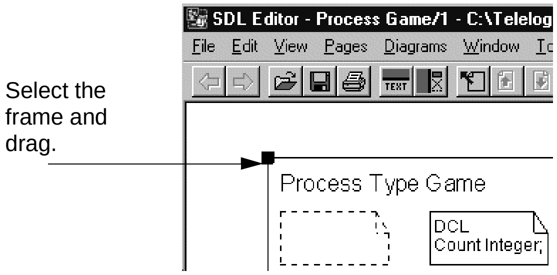


Figure 160: Adjusting the frame symbol

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:

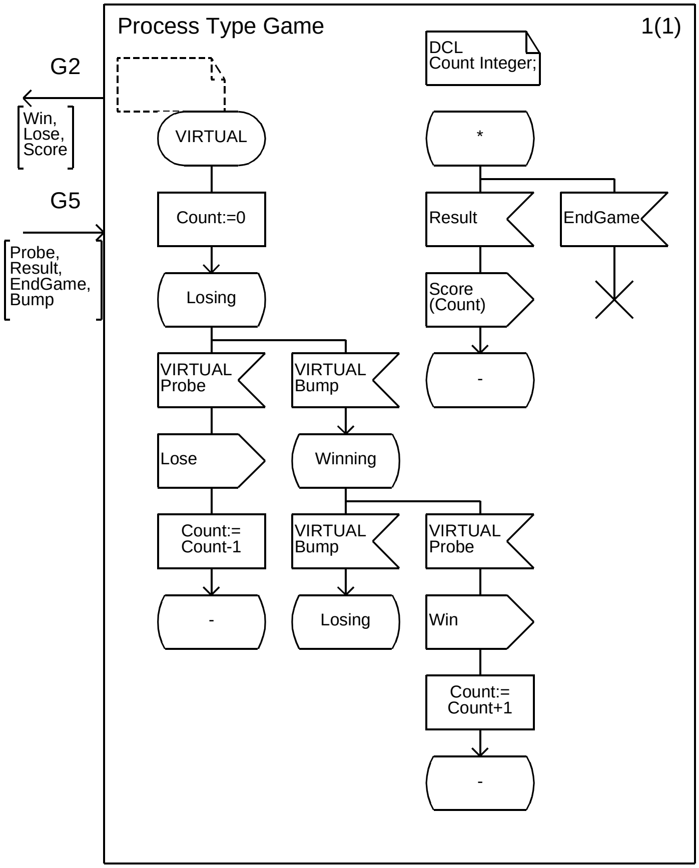


Figure 161: The resulting process type Game

Connecting to the Finished Diagram

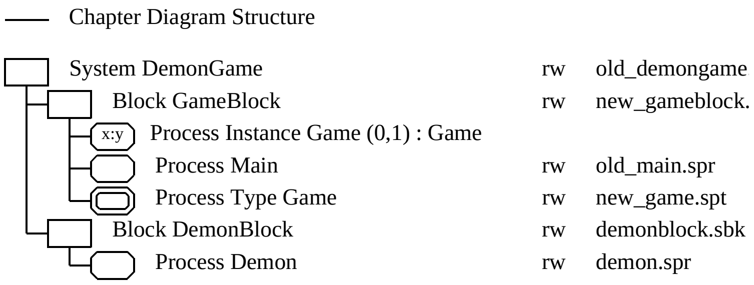
The finished Game process type diagram is also available as the file `game.spt`. If you instead of drawing the diagram wish to use this file, do as follows:

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.

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		
System DemonGame	rw	old_demongame
Block GameBlock	rw	new_gameblock.
Process Instance Game (0,1) : Game		
Process Main	rw	old_main.spr
Process Type Game	rw	new_game.spt
Block DemonBlock	rw	demonblock.sbk
Process Demon	rw	demon.spr

Figure 162: 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 run-time, 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.

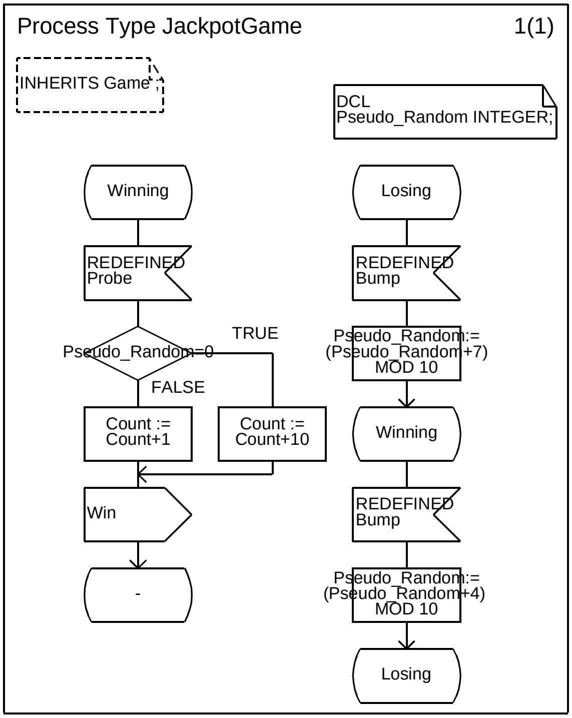


Figure 163: The process type JackpotGame

- 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.

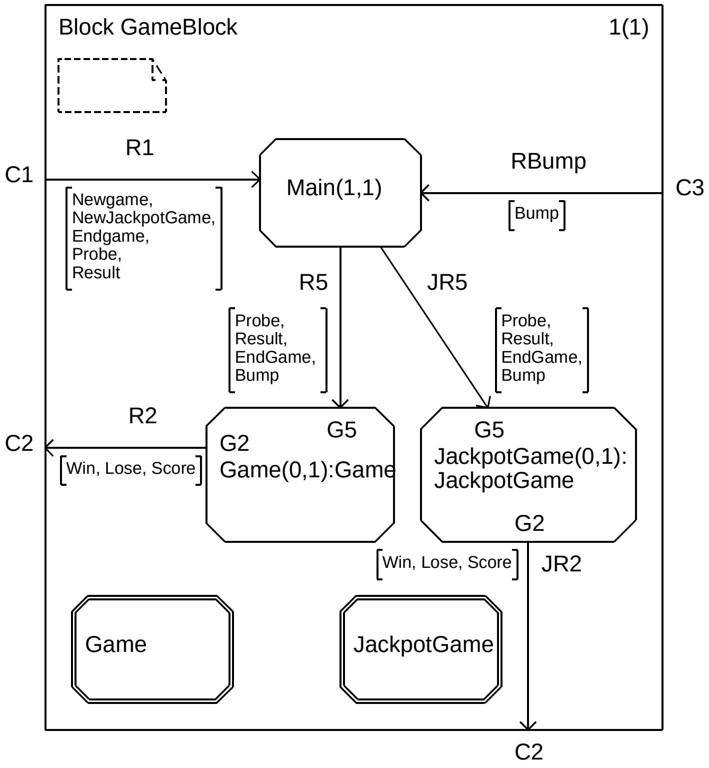

Block GameBlock
1(1)
R1
C1
Newgame, NewJackpotGame, Endgame, Probe, Result
Main(1,1)
RBump
C3
Bump
R5
JR5
Probe, Result, EndGame, Bump
Probe, Result, EndGame, Bump
R2
C2
Win, Lose, Score
G2
G5
Game(0,1):Game
G5
JackpotGame(0,1): JackpotGame
G2
Win, Lose, Score
JR2
Game
JackpotGame
C2

Figure 164: 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:

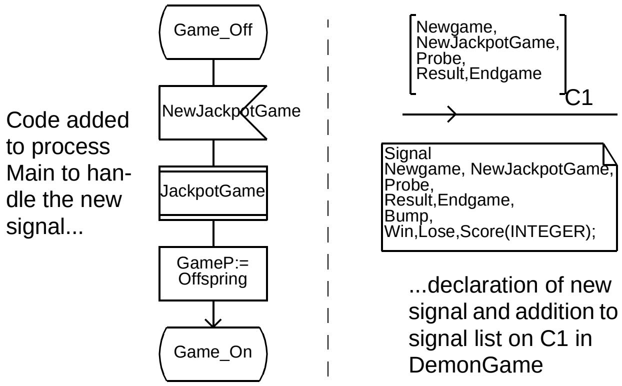
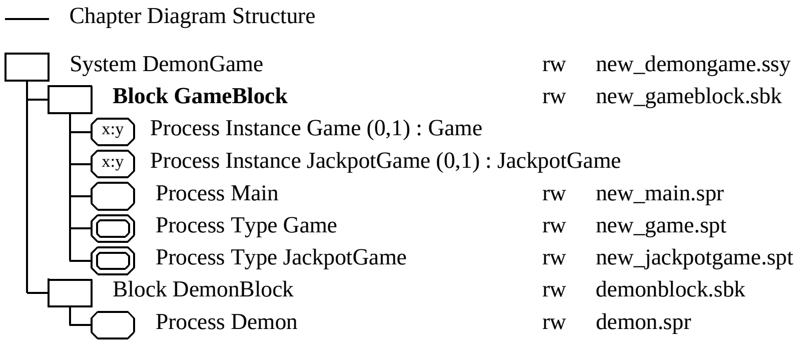

Code added to process Main to handle the new signal...
Game_Off
NewJackpotGame
JackpotGame
GameP:= Offspring
Game_On
Newgame, NewJackpotGame, Probe, Result,Endgame
C1
Signal Newgame, NewJackpotGame, Probe, Result,Endgame, Bump, Win,Lose,Score(INTEGER);
...declaration of new signal and addition to signal list on C1 in DemonGame

Figure 165: 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		
System DemonGame	rw	new_demongame.ssy
Block GameBlock	rw	new_gameblock.sbk
Process Instance Game (0,1) : Game		
Process Instance JackpotGame (0,1) : JackpotGame		
Process Main	rw	new_main.spr
Process Type Game	rw	new_game.spt
Process Type JackpotGame	rw	new_jackpotgame.spt
Block DemonBlock	rw	demonblock.sbk
Process Demon	rw	demon.spr

Figure 166: 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.

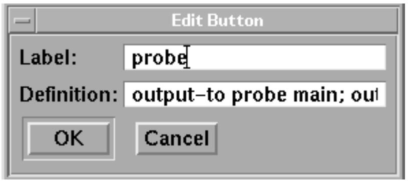
2. We suggest that you check the following features:

   – 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.)

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`**)



Label: probe

Definition: output-to probe main; out

Figure 167: 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.

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*).

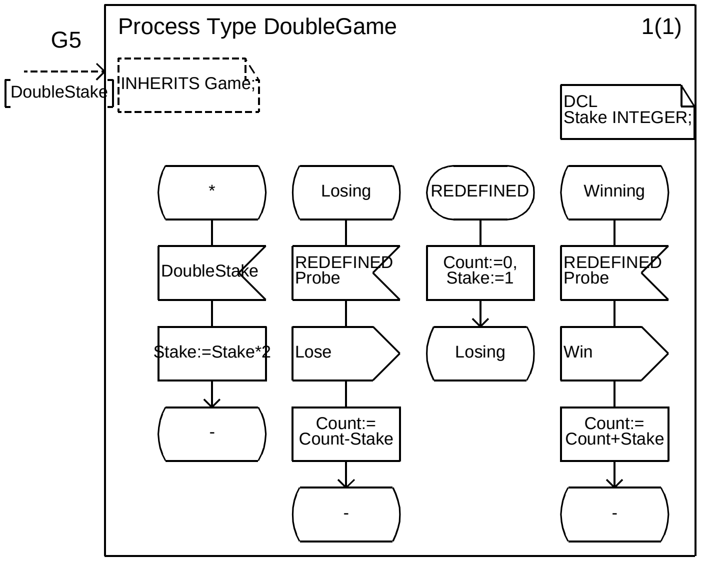


Figure 168: The process type DoubleGame

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 to the process DoubleGame; see below.

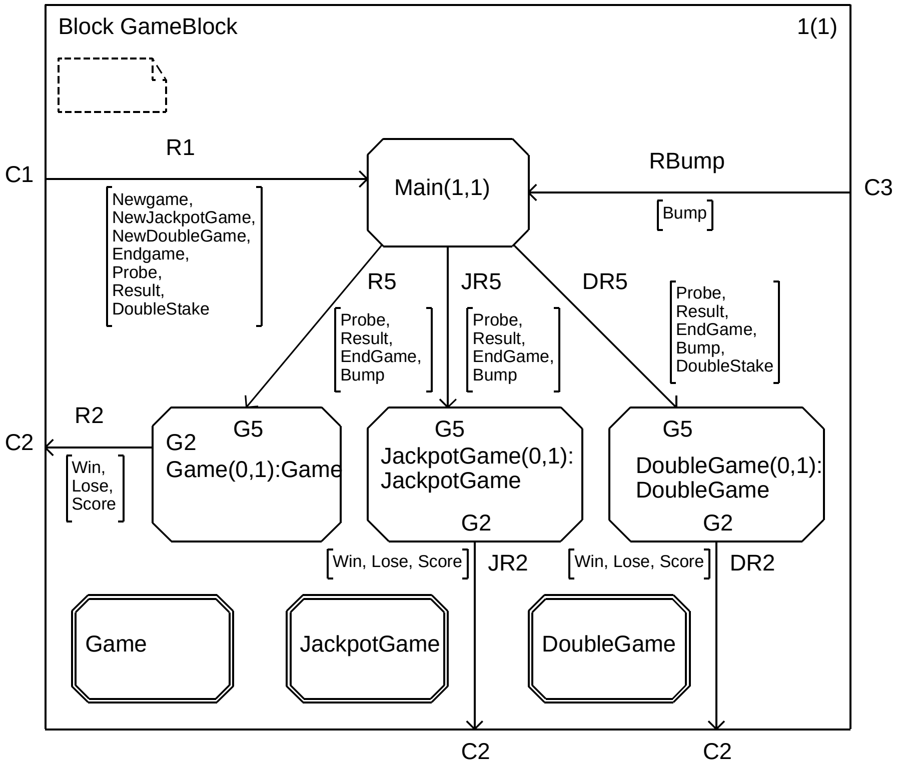

Block GameBlock
1(1)
R1
C1
Newgame, NewJackpotGame, NewDoubleGame, Endgame, Probe, Result, DoubleStake
Main(1,1)
RBump
C3
Bump
R5
JR5
DR5
Probe, Result, EndGame, Bump
Probe, Result, EndGame, Bump
Probe, Result, EndGame, Bump, DoubleStake
R2
C2
Win, Lose, Score
G2
G5
Game(0,1):Game
G5
JackpotGame(0,1): JackpotGame
G2
G5
DoubleGame(0,1): DoubleGame
G2
Win, Lose, Score
JR2
Win, Lose, Score
DR2
Game
JackpotGame
DoubleGame
C2
C2

Figure 169: 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.**

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.

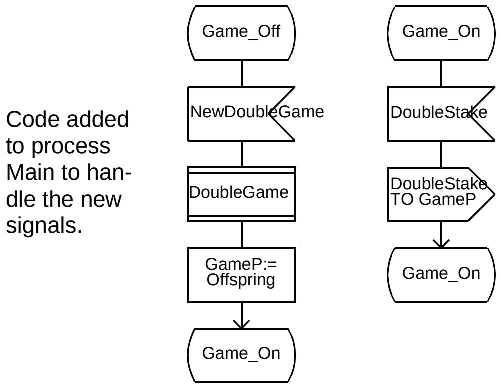


Figure 170: New code in process Main

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.

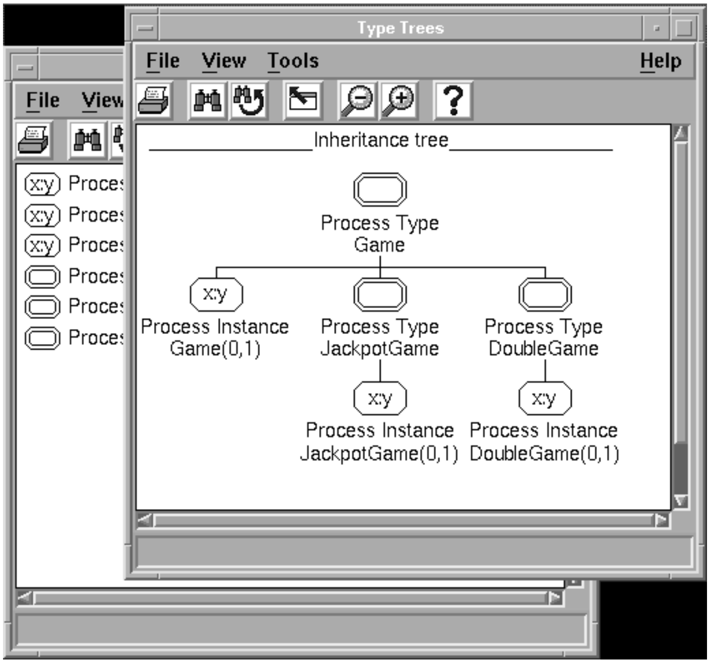


Figure 171: The two windows of the Type Viewer

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.

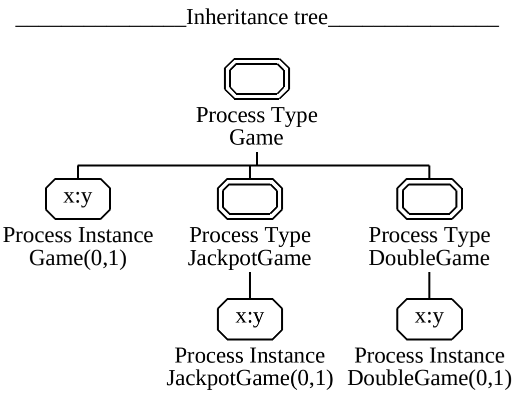


Figure 172: The inheritance tree for the process type Game

Figure 172 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.

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.

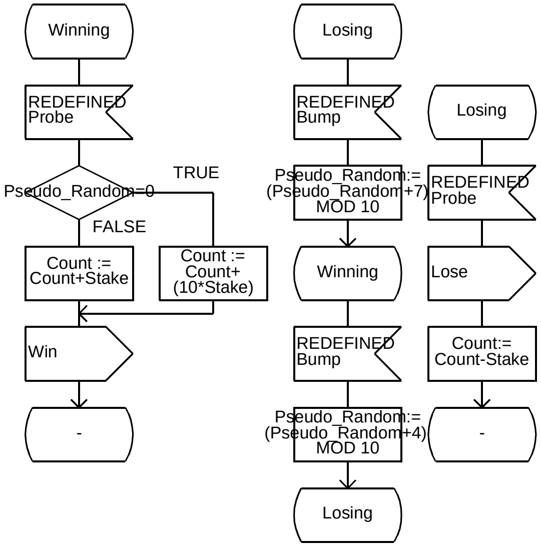


Figure 173: The changes to the process type SuperGame

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 the 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:

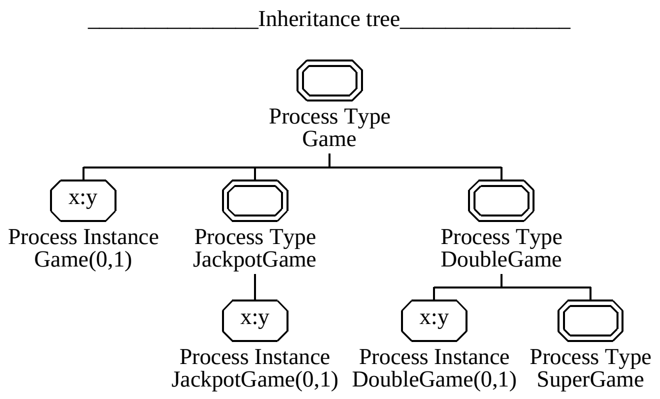


Figure 174: The process type SuperGame, added

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.

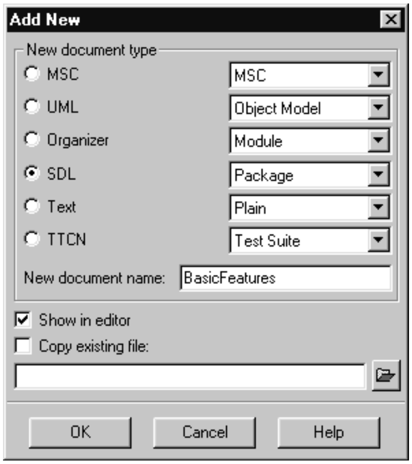
1. Start by copying the files `gameblock.sbt` and `main.spt` from the directory `$telelogic/sdt/examples/demongame/sdl92/packages` **(on UNIX)**, or `C:\Telelogic\SDL_TTCN_Suite4.5\sdt\examples\demongame\sdl92\packages` **(in Windows)** to the same directory you created earlier for this tutorial.
   – 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`**.



Add New

New document type

MSC: MSC
UML: Object Model
Organizer: Module
[✓ SDL]: Package
Text: Plain
TTCN: Test Suite

New document name: BasicFeatures

[✓] Show in editor
[ ] Copy existing file:

Figure 175: 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.

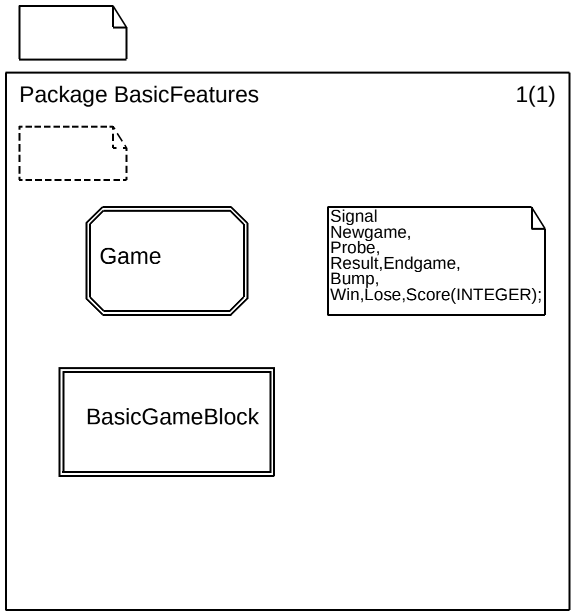


Figure 176: The package BasicFeatures

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 177).
   - 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).

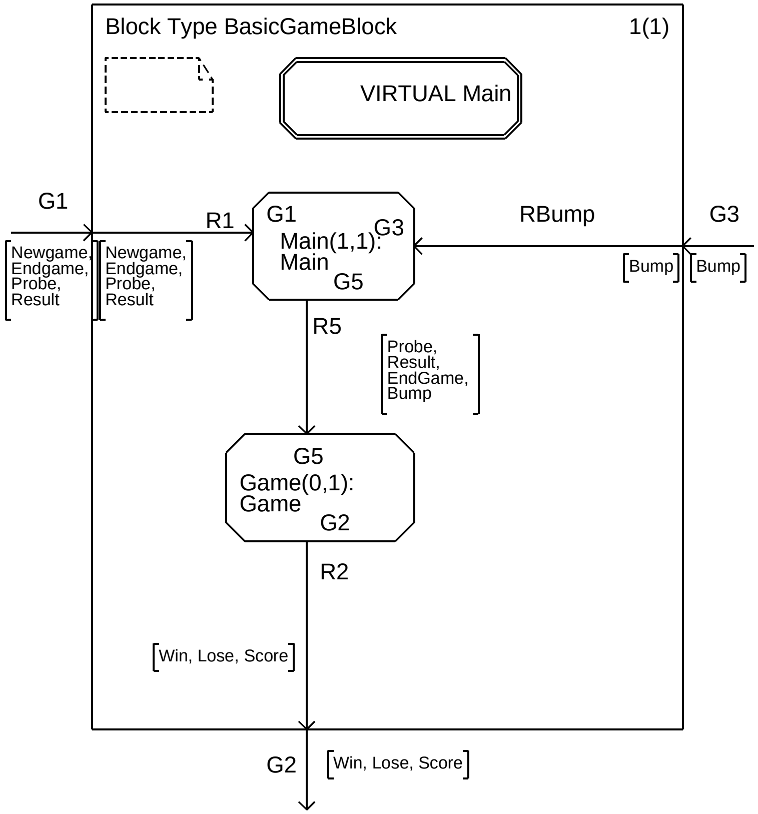


Figure 177: The block type BasicGameBlock

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.

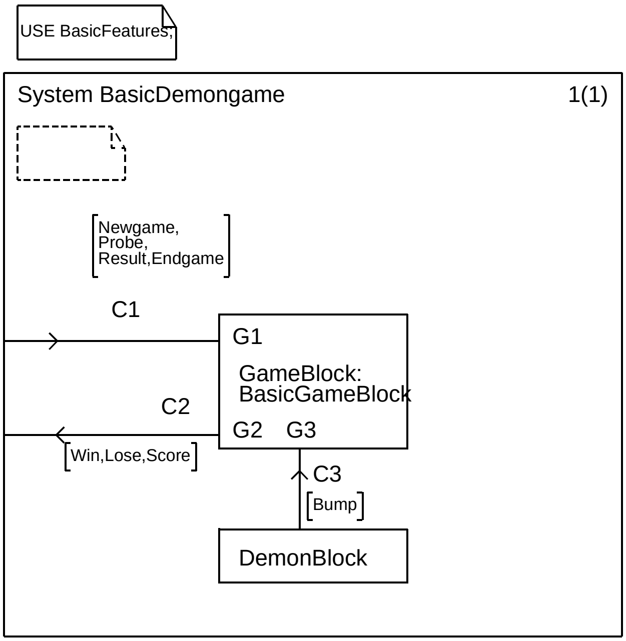


Figure 178: USE of packages

   – 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):

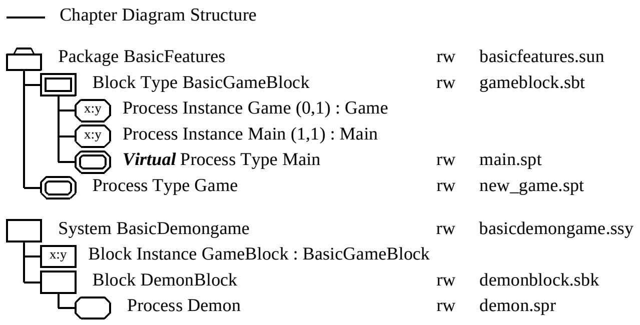

Chapter Diagram Structure
Package BasicFeatures rw basicfeatures.sun
Block Type BasicGameBlock rw gameblock.sbt
x:y Process Instance Game (0,1) : Game
x:y Process Instance Main (1,1) : Main
Virtual Process Type Main rw main.spt
Process Type Game rw new_game.spt
System BasicDemongame rw basicdemongame.ssy
x:y Block Instance GameBlock : BasicGameBlock
Block DemonBlock rw demonblock.sbk
Process Demon rw demon.spr

Figure 179: 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 175 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`

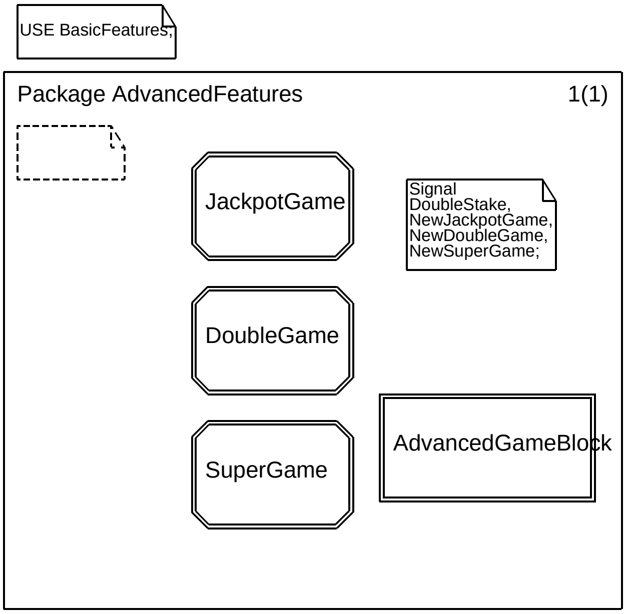


Figure 180: The package AdvancedFeatures

Block Type AdvancedGameBlock

The diagram contains a reference to a REDEFINED process type Main, and a dashed instantiation symbol Main.

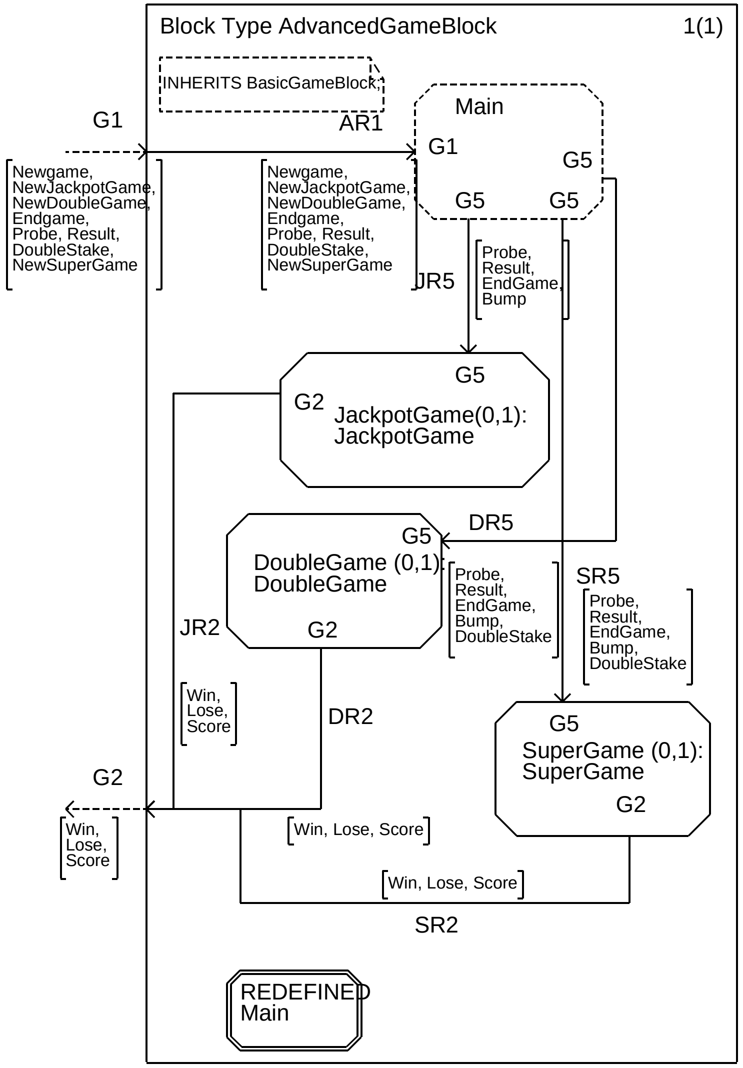

Block Type AdvancedGameBlock
1(1)
INHERITS BasicGameBlock;
G1
AR1
Main
G1
G5
G5
G5
Newgame, NewJackpotGame, NewDoubleGame, Endgame, Probe, Result, DoubleStake, NewSuperGame
Newgame, NewJackpotGame, NewDoubleGame, Endgame, Probe, Result, DoubleStake, NewSuperGame
JR5
Probe, Result, EndGame, Bump
G5
G2
JackpotGame(0,1): JackpotGame
DR5
G5
DoubleGame (0,1): DoubleGame
Probe, Result, EndGame, Bump, DoubleStake
SR5
Probe, Result, EndGame, Bump, DoubleStake
JR2
G2
Win, Lose, Score
DR2
G5
SuperGame (0,1): SuperGame
G2
G2
Win, Lose, Score
Win, Lose, Score
Win, Lose, Score
SR2
REDEFINED Main

Figure 181: Block type AdvancedGameBlock

- The block type AdvancedGameBlock is already provided on the file `advancedgameblock.sbt`. Copy that file from the directory `$telelogic/sdt/examples/demongame/sdl92/packages` **(on UNIX)**, or `C:\Telelogic\SDL_TTCN_Suite4.5\sdt\examples\demongame\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.

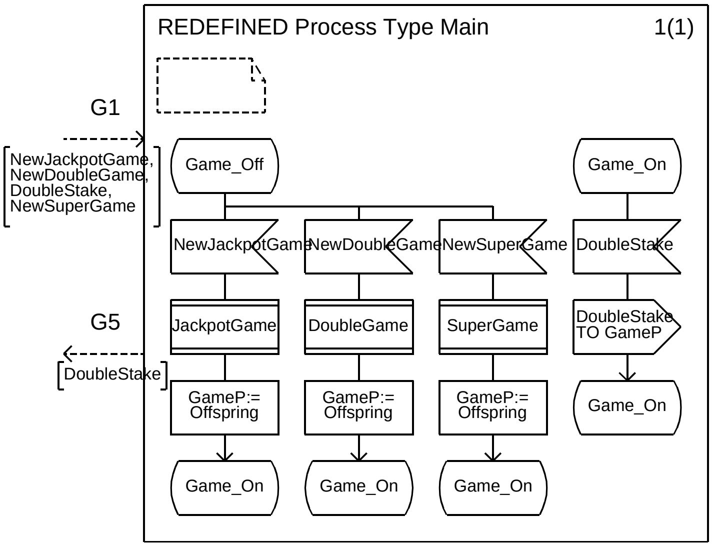


Figure 182: The redefined process type Main

- The REDEFINED process type Main is also provided on the file `advancedmain.spt` in the directory `$telelogic/sdt/examples/demongame/sdl92/packages` **(on UNIX)**, or `C:\Telelogic\SDL_TTCN_Suite4.5\sdt\examples\demongame\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.

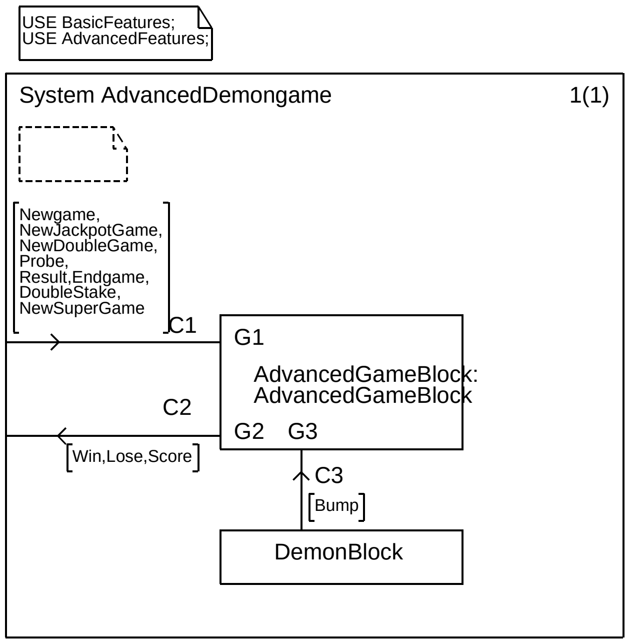

USE BasicFeatures;
USE AdvancedFeatures;
System AdvancedDemongame
1(1)
Newgame, NewJackpotGame, NewDoubleGame, Probe, Result,Endgame, DoubleStake, NewSuperGame
C1
G1
AdvancedGameBlock: AdvancedGameBlock
C2
G2
G3
Win,Lose,Score
C3
Bump
DemonBlock

Figure 183: 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 rewrite 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:

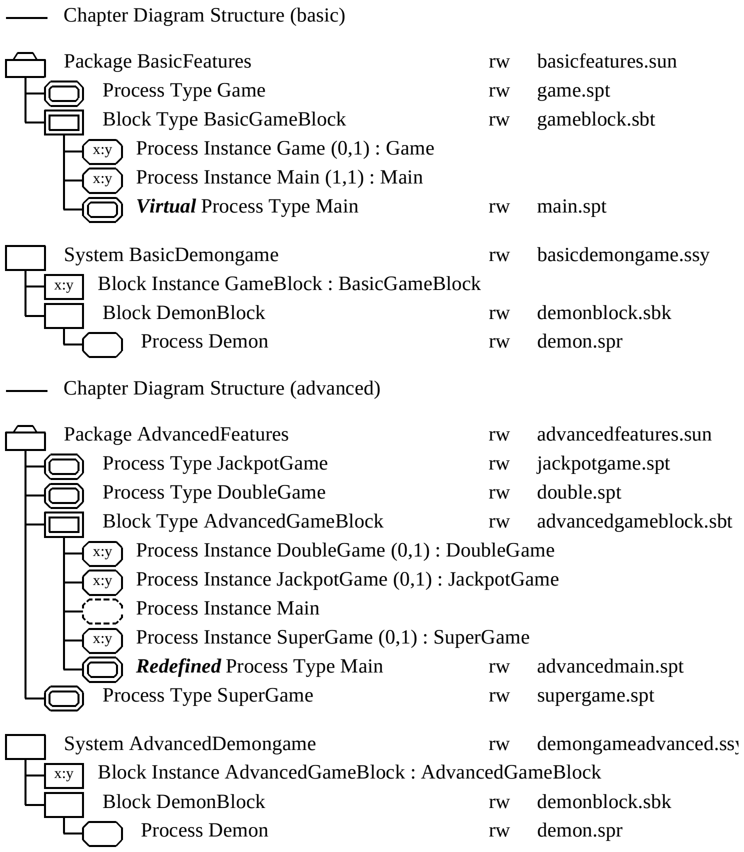
1. Memorization of “highest score ever” since system start (there should be only one highest score, common for all types of games).
2. A “hall of fame” that memorizes the name of the player that reaches the “highest score ever”. (The name is assumed to be provided by the environment).
3. A “gameover” 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:\Telelogic\SDL_TTCN_Suite4.5\sdt\examples\demongame\sdl92\exercises` **(in Windows)**

Appendix: Diagrams for the DemonGame Using Packages



Chapter Diagram Structure (basic)		
Package BasicFeatures	rw	basicfeatures.sun
Process Type Game	rw	game.spt
Block Type BasicGameBlock	rw	gameblock.sbt
Process Instance Game (0,1) : Game		
Process Instance Main (1,1) : Main		
Virtual Process Type Main	rw	main.spt
System BasicDemongame	rw	basicdemongame.ssy
Block Instance GameBlock : BasicGameBlock		
Block DemonBlock	rw	demonblock.sbk
Process Demon	rw	demon.spr
Chapter Diagram Structure (advanced)		
Package AdvancedFeatures	rw	advancedfeatures.sun
Process Type JackpotGame	rw	jackpotgame.spt
Process Type DoubleGame	rw	double.spt
Block Type AdvancedGameBlock	rw	advancedgameblock.sbt
Process Instance DoubleGame (0,1) : DoubleGame		
Process Instance JackpotGame (0,1) : JackpotGame		
Process Instance Main		
Process Instance SuperGame (0,1) : SuperGame		
Redefined Process Type Main	rw	advancedmain.spt
Process Type SuperGame	rw	supergame.spt
System AdvancedDemongame	rw	demongameadvanced.ss
Block Instance AdvancedGameBlock : AdvancedGameBlock		
Block DemonBlock	rw	demonblock.sbk
Process Demon	rw	demon.spr

Figure 184: Hierarchical structure

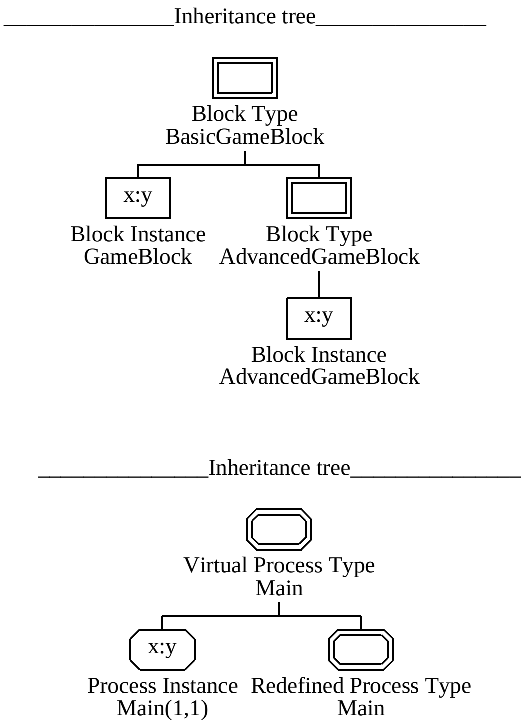


Figure 185: Inheritance tree for the block type and process type Main

(The inheritance tree for the process type Game is displayed in Figure 172 on page 252.)