Chapter

43 *The UML2SDL Utility*

The UML2SDL utility converts a model described in UML to an SDL system. This includes conversion of UML Static Structure diagrams to corresponding SDL concepts, as well as translation of UML (Harel) State Chart diagrams to SDL process diagrams.

This chapter includes setup instructions and a description of the functionality of the tool. In the end of the chapter, the mapping rules are described.

This guide assumes that you are familiar with the concepts of UML – static structure diagrams – as well as SDL.

Setting Up the UML2SDL Utility

To efficiently run the UML2SDL utility, you will need to set up a specialized Organizer menu bar containing the menu *UML To SDL*. This modified Organizer menu is defined in the file `org-menus.ini` that is located in `/orca/uml2sdl/examples/` in the installation directory. How to add the UML2SDL menu to the Organizer is described in

.

Converting UML Diagrams

When UML diagrams are to be converted into SDL diagrams, the UML diagrams need to be placed within a module in the Organizer. The diagrams may be either static structure diagrams or state charts. If multiple diagrams exist within the module, all diagrams will be converted at the same time.

To convert UML diagrams in a module:

1. Select a diagram within the module.
2. Select the desired conversion alternative from the *UML To SDL* menu (see below).

The UML2SDL converter will create a new module with the same name as the converted module, but with the prefix "SDL_" added. The new module will contain all the resulting SDL diagrams.

The *UML To SDL* Menu

The *UML To SDL* menu in the Organizer contains four alternatives:

- *Generate SDL System* – generates a system using the default transformation options.
- *Generate SDL Package* – generates a package using the default transformation options.
- *Generate SDL System* – allows you to configure the generation of an SDL system using transformation options.
- *Generate SDL Package* – allows you to configure the generation of an SDL package using transformation options.

Transformation Options

The UML2SDL utility is run as a command-line tool, but is started from the Organizer. If you select an alternative in the *UML To SDL* menu that allows you to change the transformation options, a dialog is opened in which you may specify the transformation options to UML2SDL:

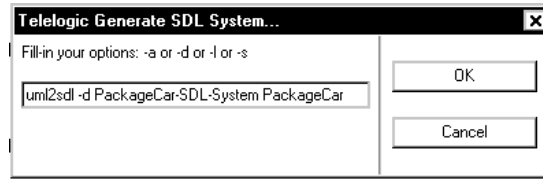

Figure 340: Setting transformation options for a package generation

If you select an alternative in the *UML To SDL* menu that uses the default transformation options, the dialog is not opened and none of the options described below are used.

The UML2SDL utility accepts a set of flags which allow you to configure the transformation:

```
uml2sdl [ -a | -d | -l | -o | -p | -s ] <module>
```

- `-a`
  Avoid types: All *«process»* and *«block»* classes that can have a type property, will automatically have this type set to “false”.
- `-d <directory>`
  The name of the sub directory that will contain the generated SDL files. A suggestion is given in the dialog, which you can alter. If the directory name does not match an existing sub directory, a new one will be created inside the current working directory.
- `-l`
  Local types: Push all type definitions as low as possible in SDL block hierarchies.
- `-o`
  Output SDL/PR to stdout, that is, the generated SDL/PR will be output in the Organizer log. No SDL diagram is created in the Organizer.

- `-p`
  If used, the UML package will be translated to an SDL package; otherwise it will be translated to an SDL system. This option is preset on the command line if you selected *Generate SDL Package* from the *UML To SDL* menu.
- `-s`
  Signal default: All operations with no return values are considered to be signals and not remote procedures. If not used, only operations following the *«signal»* stereotype or operations with a property "{async}" will be mapped to signals.
- *<module>*
  The name of the package or system that will be created. A suggestion is given in the dialog, which you can alter.

Transformation Rules

The transformation rules for UML static structure diagrams are described below. For information about the transformation rules applied when converting state charts, please refer to

.

General

To ensure traceability between the UML model and the SDL model, all mapped entities keep their name throughout the models.

An Organizer module is considered to represent a UML package. A module may contain several UML static structure diagrams. The module, including all UML static structure diagrams, is transformed into an SDL package or system.

Each SDL system or package will also contain a block type representing the architecture of the UML model.

Classes

The UML2SDL utility makes some basic assumptions about how a class should be interpreted in the SDL context. The basic approach is however to instruct the tool by using stereotypes. A stereotype is a meta classification of a class. The tool relies on the following stereotypes:

- A class with the stereotype «newtype» becomes an SDL newtype.
- A class with the stereotype «process» becomes an SDL process.
- A class with the stereotype «block» becomes an SDL block.

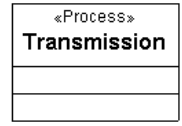

Figure 341: A class with the stereotype « process »

A class may have attributes and operations. Depending on how the class is interpreted, attributes become variables in a process, or fields in a struct. An attribute have a name and a type; the type may be omitted. The notation for the attribute is:

```
< name > [ ':' < type >]
```

An operation has a name. It may also contain parameters and return values. An operation with a return value is translated to a remote procedure; otherwise it is translated to a signal. The notation of an operation is:

```
< name > ['(' { < parameter > [':' < type > ] }*
')' ] [':' < return value >]
```

A UML static structure diagram may contain references to classes defined in other packages. Such classes are given the name according to the following notation:

```
< Name of external package > '::' < class name >
```

Each externally defined class will generate a use clause, referring to the external package, in the generated SDL system or package.

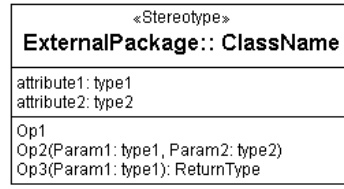

Figure 342: The notation for a process

Relations

Inheritance

Two classes connected with an inheritance relationship will generate an inheritance clause in the subtype. If two «process» classes are related by the inheritance relationship, there will be an inheritance clause in the process type representing the subclass.

Inheritance relationships between two «newtype» classes are not allowed due to limitations in SDL.

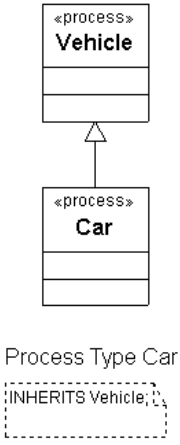

Figure 343: An inheritance relationship in the UML model will result in an inheritance relationship in the SDL model

Aggregation

Two classes may be related by an aggregation. Depending on the context the following translations will be made:

An aggregation between a «block» class and a «process» class will place the process inside the generated block type:

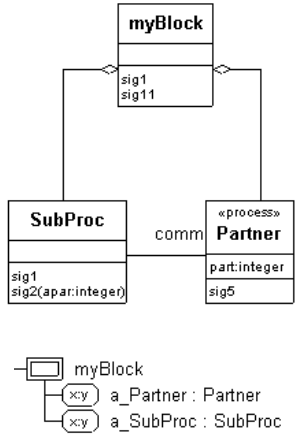

Figure 344: Aggregation expressed in UML and the generated SDL diagrams in the Organizer view

An aggregation between a «process» class and a «newtype» class will place a variable of the newtype in the process. The newtype definition will be placed in the block containing the process.

Association

Two classes connected with an association will be transformed as two classes and connected with a signal-route and/or a channel depending on the context.

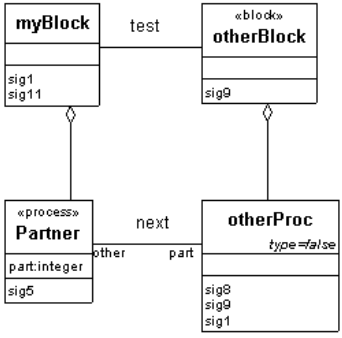

Figure 345: Communicating classes

In above, two classes “Partner” and “otherProc” are connected by an association. In the generated SDL the two processes “otherProc” and “Partner” will exist in two separate blocks and they will be able to communicate through a channel and two signal routes.

State Charts

State Charts placed inside the module will be converted together with the other diagrams inside that module. The converter assumes that if a state chart has the same name as a process then the converter will try to merge the resulting SDL process behavior into the diagram of that process.

A Small Example

This example is intended to show how the UML2SDL utility can be used. The example contains a small analysis model of a game – the Demon game – which is intended to be implemented in SDL through a design model. The Demon game is used as an example in other parts of the Telelogic Tau documentation. For example, see

.

Model Relationships

The purpose of an analysis model is to identify the problem: **what** is to be done? The following model is the design model which identifies the solution. The design model answers the question **how** it is to be done. Good practice is to have clear dependencies between the two models, that is, traceability. Traceability is one of the key factors in a successful project.

Very often, the analysis model can be reused when the design model is created. The information provided by the problem statement is needed in the design model. The purpose of the UML2SDL utility is to automate this reuse as much as possible.

The Analysis Model

The Class Diagram

The analysis model is presented in . It contains a set of classes which describes an overview of the Demon game. A pure analysis model might not be as detailed as the one given in the example, but the level of detail in the example is chosen to highlight the functionality of the UML2SDL utility.

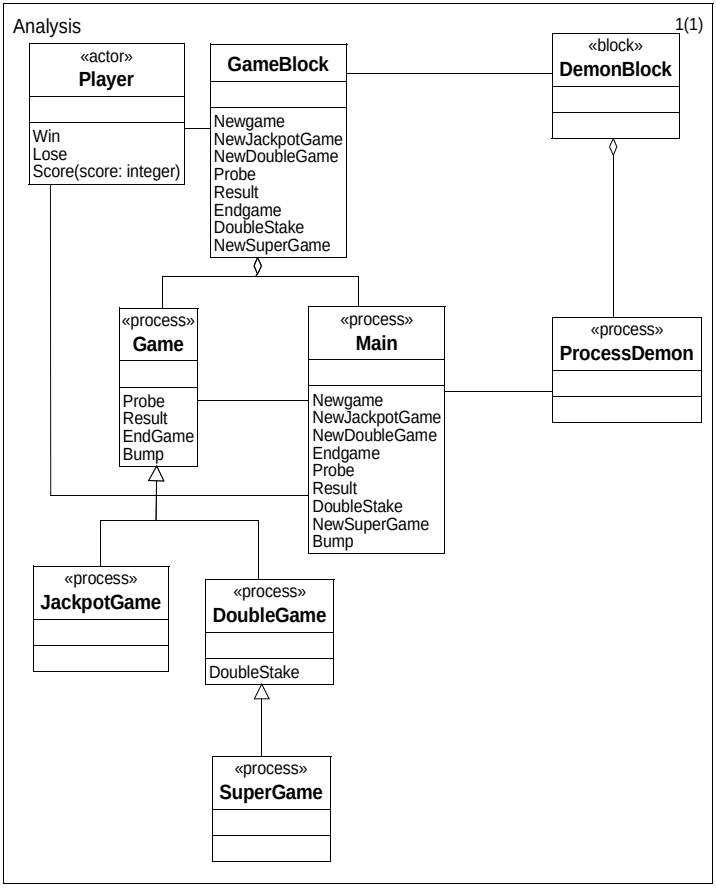

Figure 346: The Analysis Object Model

All operations of the classes in the Analysis Object Model will be mapped to signals since no operation contains a return value. Aggregate classes containing «process» classes, or marked with the stereotype «block», like GameBlock and DemonBlock, will become SDL blocks. The aggregations also tell the UML2SDL utility where to place the «process» classes. For example, the process Demon will be placed inside the block DemonBlock and the other processes will be placed inside the GameBlock.

Associations between classes are mapped to channels and signal routes. The association between the two classes GameBlock and DemonBlock will become a channel between the corresponding blocks. The association between the classes Main and the ProcessDemon will result in a path of communication between the resulting processes.

The inheritance relationship between the Game class and its sub-classes will become inheritance relationships between the corresponding processes as well. Since associations are inherited in UML but signal routes or channels are not inherited in SDL, the UML2SDL utility creates signal routes/channels for inherited associations. This results in signal routes between the process Main and Game, as well as between Main and the sub-processes of Game.

Classes with the stereotype «actor» is mapped as communication with the environment. In our example the class Player will become a channel to the environment. The operations of the class Player will be signals to the environment.

The State Charts

There exist two state charts named Main and Game in the module that will be converted, see . The intention of using state charts in the analysis is to get an overview of the behavior of important classes, see . The state charts will be transformed together with the class diagram. Since two «process» classes Main and Game exist the result of the UML to SDL conversion will be two complete process descriptions.

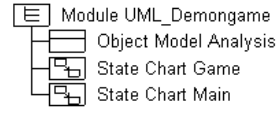

Figure 347: The module that will be converted.

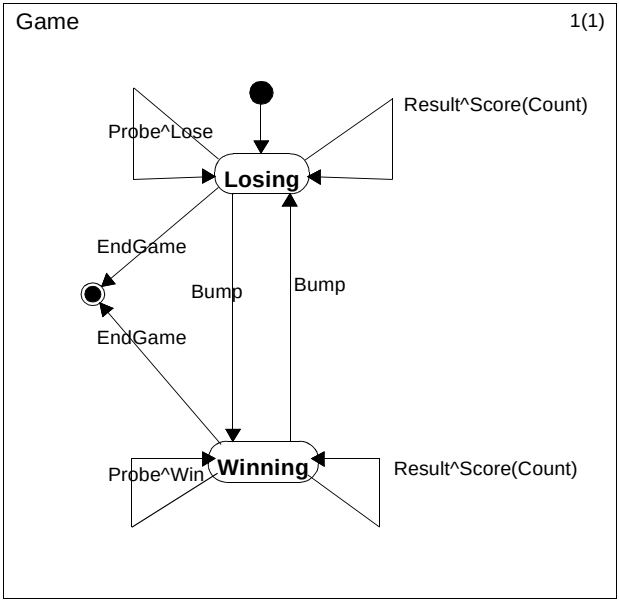

Figure 348: The behavior of Game.

The Design Model

The nature of software engineering always requires design information to be added to the analysis at some stage, because we can never solve a problem without providing a solution. This is also true when the UML2SDL utility is used.

Typically, information that needs to be added is behavior that is not described at the UML level. For instance, there is no good way of describing behavior that is redefined in a sub-class at the UML level. For example, such behavior is added to the sub-processes of process Game. Also, there was no need of modeling the simple behavior of ProcessDemon at the stage of analysis, but we have to do that in the design.

Besides adding the basic behavior, we also need to provide a full design. This includes actions like introducing variables, timers, etc. For example, in the process Main we need to describe how the process is responsible for the creation of Game processes, see .

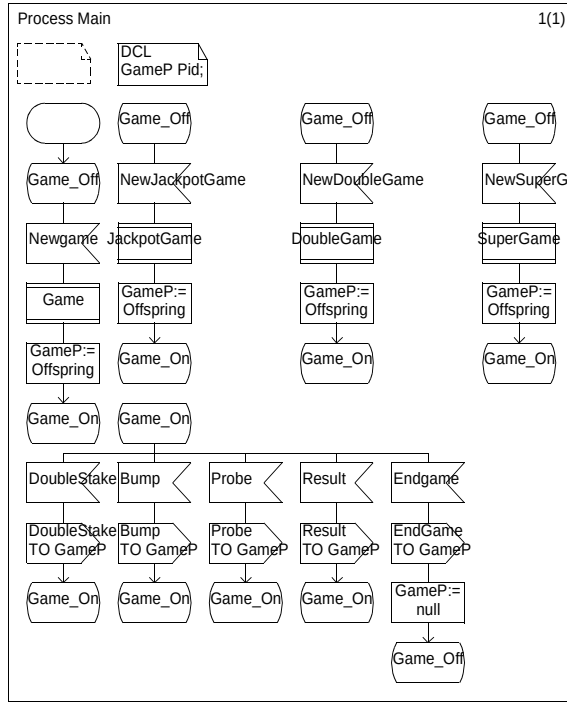

Figure 349: The complete process Main

When these steps are taken, the Demon game should be complete and possible to simulate like other SDL systems.

Summary

The intention with the UML2SDL utility is to automate the transfer from the analysis model to the design model as much as possible. The transfer will most likely include manual steps, but in theory it is possible to generate an SDL model which is detailed enough to simulate.

Chapter 43 The UML2SDL Utility