Chapter

# 76 *SOMT Implementation*

**This chapter describes the different ways to implement an SDL design and the tasks that have to be done in connection with this. The focus is on automatic implementation relying on automatic code generation and using C as the target language.**

# Implementation

The implementation activity results in a tested and running application in the target environment.

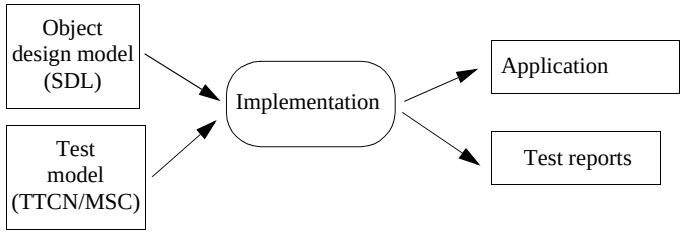


*Figure 707: Overview of the implementation activity*

The activities during implementation are highly dependent upon the application and target and there are several possible ways to implement an SDL design. It can be manually implemented in software or hardware or a more automatic implementation can be used relying on automatic code generation. The discussion in this section is mainly intended to give an overview of different aspects on the latter approach using C as the target language. There are several tasks to be done in the implementation activity:

- Partitioning the SDL system into different software (and/or hardware) run-time modules.
- Implementing the adaptation code that is needed for this specific SDL system to operate in its environment.
- Selecting and implementing a general integration strategy for the target hardware and run-time environment.
- The generation of C code and the customizing of it for different application areas.
- Testing to ensure that the application works in the target environment.

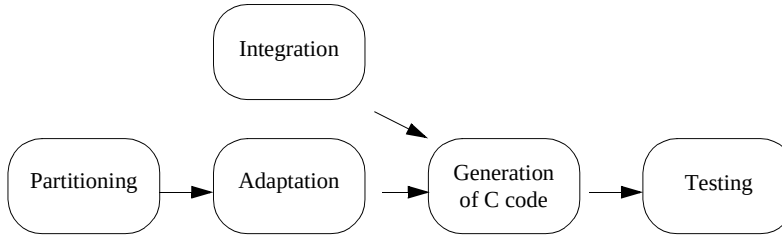


*Figure 708: Example of activities during the implementation*

# Partitioning an SDL System

The purpose of the partitioning task is to partition the SDL system into separate parts that will form stand-alone executables by themselves, either to run as separate programs on one computer or distributed on several computers in a network.

There are two different possible strategies for how to do the partitioning in practise:

- Use directives to the C code generator to define what parts of the system that will form separate run-time executables.
- Create different SDL systems for the different partitions and generate code for them separately.

# Adaptation

An application generated from an SDL description can be viewed as having three parts:

- The SDL system
- The physical environment of the system
- The environment functions, where the SDL system is connected with the environment

When adapting the application, the environment functions may have to be specified depending on the integration mechanism used. The environment functions are the place where the two worlds, the SDL system and the physical environment, meet. Signals sent from the SDL system

to the environment can be specified to perform any event in the physical environment, and events in the environment are specified to cause signals to be sent into the SDL system.

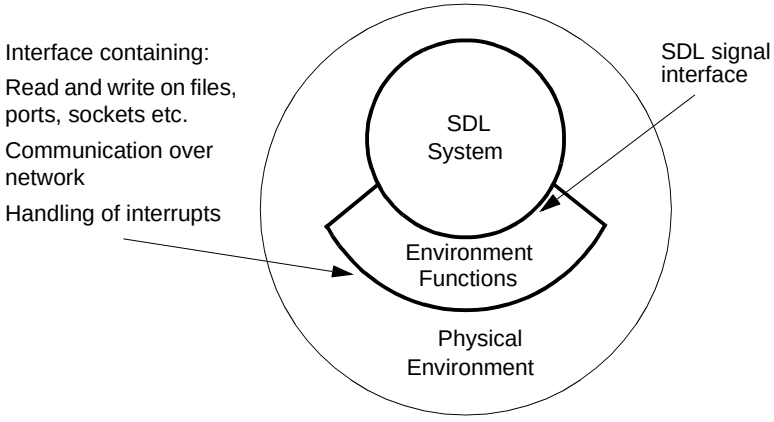


*Figure 709: The structure of an application*

In a distributed system an application might consist of several communicating SDL systems. Each SDL system will become one executable program. The environment functions that has to be written for each SDL system are:

- `xInitEnv` – handles the initialization of the environment and will be called during the start-up of the application (before the SDL system is initialized).
- `xCloseEnv` – is called when the SDL system terminates.
- `xOutEnv` – will be called each time a signal is sent out of the SDL system.
- `xInEnv` – This function is repeatedly called during the execution of the system. During this call, the environment could be scanned for events which should result in a signal sending into the SDL system.
- `xGlobalNodeNumber` – returns a number that is unique for each communicating SDL system that constitutes an application.

# Integration

The purpose of the integration task is to select (and implement if no predefined integration exists) a generic integration mechanism that makes it possible to execute a generated application in a target hardware/software environment. There are three different strategies available for this integration:

- Making an integration using the SDL run-time system, where the SDL application directly runs on a micro processor without any additional operating system support.
- Making a light integration to an operating system, where an SDL system (together with a run-time system) is treated as one task in the operating system.
- Making a tight integration to a real-time operating system, where each SDL process instance (or set of instances of a specific type) results in a task in the operating system.

Both the integration using the SDL run-time system and the light integration use a supplied run-time system to execute the SDL system that takes care of the SDL semantics including scheduling of processes etc.

A tight OS integration consists of a set of files that define how the SDL semantics is mapped to the operating system in question. The main categories are:

- Macro definitions that define the macros in the generated code
- Functions that handle constructs that cannot be used directly in the operating system (i.e. saving of signals)
- Functions that are dependent on the specific operating system (i.e. allocate/free memory)
- Definitions and handling of SDL predefined data types
- Identifiers for signals, timers and remote procedure calls
- Post processing utilities to enumerate signal and process types

An existing integration for an operating system is highly reusable and several predefined integrations are available.

# C Code Generation

Although the C code generation is a highly automated activity, there are several ways of customizing the generated code. Some examples are:

- Generating separate C files for different SDL structural entities
- Assigning priorities to processes to enable scheduling (which is undefined in SDL)
- Making a user-defined implementation for the handling of certain signals
- Making a user-defined main-loop (run-time system) in the generated application

Separation of the C files is recommended for large systems, since a minor, local change in for example a block diagram only requires a regeneration and recompilation of the code for that unit. The object files (the compiled versions of the C files) for the other unchanged units can then be used in the link operation to form a new executable program. Thus, the turn-around time from a change in the design to an executable application is minimized.

When doing a customization of the code, it is important that this is done in a way that keeps maintenance of the design easy.

# Testing

The purpose of the testing task in the implementation activity is to verify that the application works in its target environment. The input is the test cases specified in the system design and the output is a test report and a tested application. The details of the testing task is very much depending on the target environment and the possibilities to run tests against the application in this environment. Essentially the tasks involved in the testing are:

- Creating a test environment that can communicate with the application and where the tests can be executed
- Transforming the test cases to a format that can be used in the test environment
- Run the tests
- Analyze the results of the tests

# Summary

The final SOMT activity, the implementation, completes the development activities by creating an application that runs on the target environment. From the SDL object design, C code is created, the code is adapted to handle the environment of the SDL system and then integrated to the hardware by means of generic interfaces to real-time operating systems. Finally, the application is tested in its target environment.

# Chapter 76 SOMT Implementation