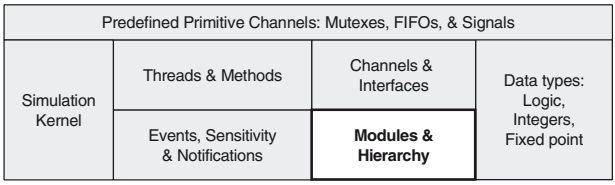
Chapter 10
Structure

Design Hierarchy



This chapter describes SystemC's facilities for implementing structure, sometimes known as design hierarchy. Design hierarchy concerns both the hierarchical relationships of modules discussed here and the connectivity that lets modules communicate in an orderly fashion. Connectivity will be discussed in the next chapter.

10.1 Module Hierarchy

Thus far, we have examined modules containing only a single level of hierarchy with all processes residing in a single module. This level of complexity might be acceptable for small designs. However, larger system designs require partitioning and hierarchy to enable understanding and project management. We consider project management to include documentation and practical issues such as integration of third-party intellectual property (IP).

Design hierarchy in SystemC uses instantiations of modules as member data of parent modules. In other words, to create a level of hierarchy, create an **`sc_module`** object within a parent **`sc_module`**. The new **`sc_module`** is the submodule within the parent module.

Consider the hierarchy in Fig. 10.1 for the following discussions and examples. In this case, we have a parent module named `Car`, with submodules named `Engine` and `Body`. To obtain the hierarchical relationship, we create an `Engine` and `Body` object within the definition of the `Car` class.

D.C. Black et al., *SystemC: From the Ground Up*, Second Edition,
DOI 10.1007/978-0-387-69958-5_10, © Springer Science+Business Media, LLC 2010

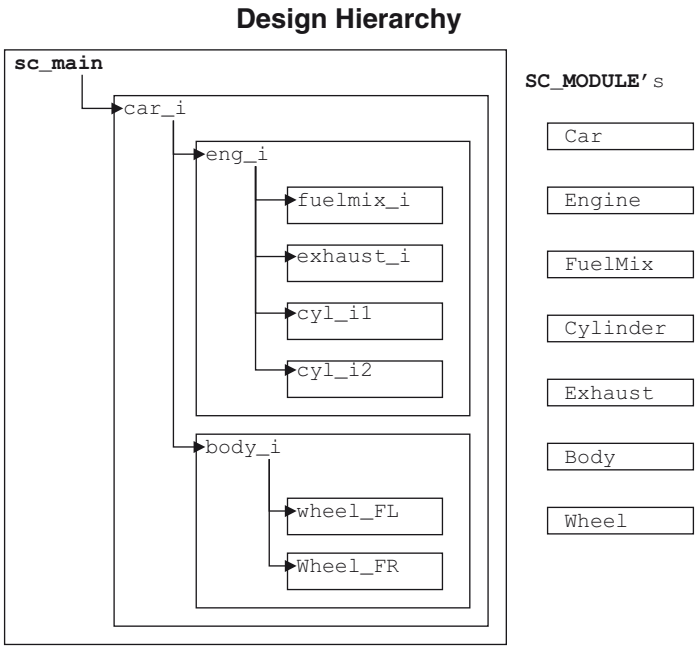


Fig. 10.1 Design hierarchy

C++ offers two basic ways to create submodule objects within the definition of a parent module or object. First, a submodule object may be created directly by declaration just like any simple member data of the module class. Second, a submodule object may be indirectly referenced by means of a pointer in combination with dynamic allocation.

When submodules are created outside of **`sc_main`**, the submodule objects must be, at minimum, initialized within the module constructor. When using indirect reference by means of a pointer, the pointer must be initialized within the constructor using `new`. Constructors can be implemented within the header file (`.h` file) or within the implementation file (`.cpp` file). Since hierarchy reflects an internal implementation decision, the authors prefer to see the constructor defined in the implementation file. The only time construction cannot be done within the implementation file is when you are defining a templated module, which triggers compiler restrictions[1]. Templated modules are a relatively rare occurrence.

Creation of hierarchy at the top level (**`sc_main`**) is slightly different from instantiation within modules. This difference results from differences in C++ syntax requirements for initialization outside of a class definition.

[1]Future versions of C++ compiler/linker tool sets may fix this restriction.

Since it is likely you may see any combination of these approaches, we will illustrate all six approaches:

- Direct top-level (**sc_main**)
- Indirect top-level (**sc_main**)
- Direct submodule header-only
- Direct submodule
- Indirect submodule header-only
- Indirect submodule

There are likely a few more variants, but understanding these should suffice. For easy reference, Table 10.1 at the end of this chapter lists the pros and cons of these approaches.

10.2 Direct Top-Level Implementation

First, we illustrate (Fig. 10.2) top-level implementation with direct instantiation, which has already been presented in `Hello_SystemC` and is used in all the succeeding discussions in the book. It is simple and to the point. Sub-design instances are simply instantiated and initialized in one statement. The name given via the constructor is the hierarchical instance name used by the SystemC kernel and is very useful during debug. The SystemC instance string is the name used by SystemC when printing error or informational messages. Normally the C++ instance name and the SystemC string name are defined to be identical. This comment applies to all SystemC module instantiation styles.

```
//FILE: main.cpp
  #include <systemc>
  #include "Car.h"
  int sc_main(int argc, char* argv[]) {
    Car car_i("car_i");
    sc_start();
    return 0;
  }
```

Fig. 10.2 Example of `main` with direct instantiation

10.3 Indirect Top-Level Implementation

A minor variation on this approach is top-level implementation with indirect instantiation (Fig. 10.3). This approach adds two lines of syntax with both a pointer declaration and an instance creation via `new`. This variation takes more code; however, it adds the possibility of dynamically configuring the design with the addition of **if-else** and looping constructs.

A design with a regular structure might even construct an array of designs and programmatically instantiate and connect them. We have used the `suffix_iptr` to indicate that `car_iptr` is a pointer to an instantiation.

```
//FILE: main.cpp
  #include <systemc>
  #include "Car.h"
  int sc_main(int argc, char* argv[]) {
    Car* car_iptr;               // pointer to Car
    car_iptr = new Car("car_i"); // create Car
    sc_start();
    delete car_iptr;
    return 0;
  }
```

Fig. 10.3 Example of `main` with indirect instantiation

10.4 Direct Submodule Header-Only Implementation

When dealing with submodules (i.e., beneath or within a module), things become mildly more interesting because C++ semantics require the use of an initializer list for the direct approach. Remember that **`SC_CTOR`** is a macro that hides the C++ constructor syntax. The constructor implemented in **`SC_CTOR`** requires initialization with a SystemC instance name, therefore the requirement to initialize the submodules. The next figure (Fig. 10.4) shows an example of direct instantiation in the header.

```
//FILE:Car.h
  #include "Body.h"
  #include "Engine.h"
  SC_MODULE(Car) {
    Body   body_i;
    Engine eng_i ;
    SC_CTOR(Car)
    : body_i("body_i") //initialization
    , eng_i("eng_i")   //initialization
    {
      // other initialization
    }
  };
```

Fig. 10.4 Example of module with direct instantiation in header

10.5 Direct Submodule Implementation

One disadvantage of the preceding approach is that it exposes the complexities of the constructor body to all potential users, even those who just want to use your module and don't care to know about your hierarchy partitioning choices. Moving the constructor into the implementation (i.e., the *module*.cpp file) requires the use of **`SC_HAS_PROCESS`**. The next figure (Fig. 10.5) shows an example of using direct instantiation.

```
//FILE:Car.h
  #include "Body.h"
  #include "Engine.h"
  SC_MODULE(Car) {
    Body   body_i;
    Engine eng_i;
    Car(sc_module_name nm);
  };
```

```
//FILE:Car.cpp
  #include <systemc>
  #include "Car.h"
  // Constructor
  SC_HAS_PROCESS(Car);
  Car::Car(sc_module_name nm)
  : sc_module(nm)
  , body_i("body_i")
  , eng_i("eng_i")
  {
    // other initialization
  }
```

Fig. 10.5 Example of module with direct instantiation and separate compilation

10.6 Indirect Submodule Header-Only Implementation

Use of indirection renders the instantiation a little bit easier to read for the submodule header-only case; however, no other advantages are clear. The next figure (Fig. 10.6) shows an example of indirect instantiation in the header.

```
//FILE:Body.h
  #include "Wheel.h"
  SC_MODULE(Body) {
    Wheel* wheel_FL_iptr;
    Wheel* wheel_FR_iptr;
    SC_CTOR(Body) {
      wheel_FL_iptr = new Wheel("wheel_FL_i");
      wheel_FR_iptr = new Wheel("wheel_FR_i");
      // other initialization
    }
  };
```

Fig. 10.6 Example of module with indirect instantiation in header

10.7 Indirect Submodule Implementation

Moving the module indirect approach into the implementation file has the advantage of possibly supplying pre-compiled object files making this approach good for IP distribution. This advantage is in addition to the possibility of dynamically determining the configuration discussed previously. The figure below (Fig. 10.7) shows an example of indirect instantiation with separate compilation.

```
//FILE:Engine.h
  class FuelMix;
  class Exhaust;
  class Cylinder;
  SC_MODULE(Engine) {
    FuelMix*  fuelmix_iptr;
    Exhaust*  exhaust_iptr;
    Cylinder* cyl1_iptr;
    Cylinder* cyl2_iptr;
    Engine(sc_module_name nm); // Constructor
  };
```

```
//FILE: Engine.cpp
  #include <systemc>
  #include "FuelMix.h"
  #include "Exhaust.h"
  #include "Cylinder.h"
  // Constructor
  SC_HAS_PROCESS(Engine);
  Engine::Engine(sc_module_name nm)
  : sc_module(nm)
  {
    fuelmix_iptr = new FuelMix("fuelmix_i");
    exhaust_iptr = new Exhaust("exhaust_i");
    cyl1_iptr    = new Cylinder("cyl1_i");
    cyl2_iptr    = new Cylinder("cyl2_i");
    // other initialization
  }
```

Fig. 10.7 Example of module with indirect separate compilation

Notice the absence of # include `FuelMix.h`, `Exhaust.h`, and `Cylinder.h` in `Engine`.h of the preceding example. This omission could be a real advantage when providing `Engine` for use by another group. You need to provide only `Engine.h` and a compiled object or library (e.g., `Engine.o` or `Engine.a`) files. You can then develop your implementation independently. This approach is good for both internal and external IP distribution.

10.8 Contrasting Implementation Approaches

The following table (Table 10.1) contrasts the features of the six approaches.

Some groups have the opinion that the top-level module should instantiate a single design with a fixed name (e.g., Design_top) and then apply a consistent approach for all the submodules. Some EDA tools perform all this magic for you.

Table 10.1 Comparison of hierarchical instantiation approaches

Level	Allocation	Pros	Cons
Main	Direct	Least code	Inconsistent with other levels
Main	Indirect	Dynamically configurable	Involves pointers
Module	Direct header only	All in one file Easier to understand	Requires submodule headers
Module	Indirect header only	All in one file Dynamically configurable	Involves pointers Requires submodule headers
Module	Direct with separate compilation	Hides implementation	Requires submodule headers
Module	Indirect with separate compilation	Hides submodule headers and implementation Dynamically configurable	Involves pointers

10.9 Exercises

For the following exercises, use the samples provided at www.scftgu.com.

Exercise 10.1: Examine, compile, and run the **sedan** example. Which styles are simplest?

Exercise 10.2: Examine, compile, and run the **convertible** example. Notice the forward declarations of **Body** and **Engine**. How might this be an advantage when providing IP?