Chapter 4

FEW WORDS ABOUT IMPLEMENTATION CLASS HIERARCHY

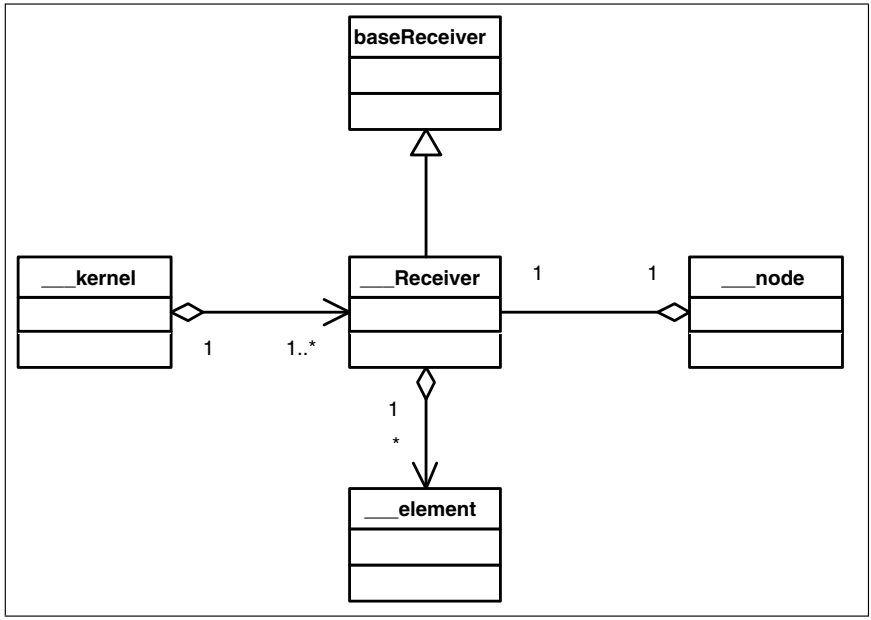


Figure 4.1. General Implementation Class Hierarchy

This chapter informs the reader about the organization of our kernel implementation. During the development process of alternative kernels for SystemC, several implementation hierarchies and data structures were investigated. In this book we did not make an effort to unify them. This book presents a snapshot of the current status of the project so that other interested developers can use the concepts and ideas to de-

velop their own multi-MoC kernels. Once the reader goes through the Synchronous Data Flow, Communicating Sequential Processes and Finite State Machine chapters, a distinct difference in class hierarchy with the SystemC kernel development can be noticed. We expect further gradual changes in implementation hierarchy in the future as we improve our SystemC kernel implementations. Nonetheless, we propose a hierarchy that allows for an extensible design for multi-MoC modeling. We simply lay a foundation that can support this, but do not currently employ it to its maximum potential.

In general, the class hierarchy resembles the class diagram shown in Figure 4.1. Some of the terminology used in Figure 4.1 are borrowed from [25]. It is not necessary to strictly conform to this class hierarchy, because some implementations do not require such a class structure and some require more encapsulation. The terminology used are as follows:

Kernel: A class that allows for creation and simulation of multiple instances of a model represented by a specific MoC.

Node: A representation of a specific function block that exhibits behavior specified by the MoC. For example, a *CSPnode* is a representation for a CSP process by encapsulating an instance of *CSPReceiver*.

Receiver: An encapsulation class to separate the data structure of an MoC from its communication with the designer and MoC-specific nodes. The common functionalities can be derived from a *baseReceiver* class.

baseReceiver: A class that encapsulates common functionalities and data structures employed by a receiver. Examples are queues that are used in DE and CSP MoCs as runnable lists and graph structures as used in representing an SDF and CSP model.

Element: Embedded deepest of all classes in terms of class hierarchy, an *element* class defines a structure that aids in creating the main data structure used to construct a model for an MoC.

This class hierarchy is only to provide minimal organization in developing the additional kernels and classes for encapsulation and functionality. Additional classes if required, should be added for better object oriented programming practices.

The CSP kernel class diagram in Figure 4.2 illustrates the CSP kernel implementation loosely employing the general implementation class hierarchy.

A CSP model is best represented as a graph. This graph is represented in *CSPReceiver* by a list-based data structure using Standard Template

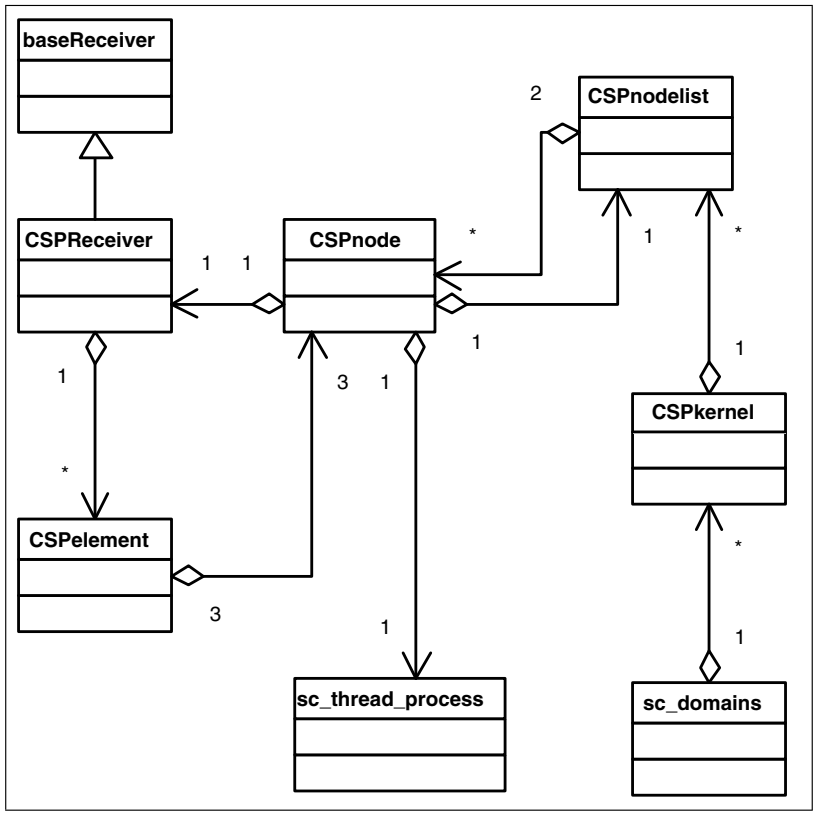


Figure 4.2. CSP Implementation Class Hierarchy

Library (STL) *vector* class. This list contains pointers to *CSPelement*. The *CSPelement* class provides the *CSPReceiver* with information about which nodes are connected via channels and the data to be transferred on each channel. *CSPchannel* inherits from *CSPelement* as shown later in this chapter, because in essence they exhibit the exact same behavior. A *CSPnode* has one instance of a *CSPReceiver* and contains a pointer to the SystemC thread class *sc_thread_process*. The modeler creates an instance of *CSPnode* within an *SC_MODULE(...)* to distinguish that module as a CSP module. An additional class called *CSPnodelist* holds pointers to *CSPnode*s and this class contains member functions that simulates the CSP MoC.

The FSM implementation hierarchy is simple where class *FSMReceiver* once again is a derived class from *baseReceiver*. However, the data structure present in this receiver uses a *map*<...> STL structure. This structure contains a *string* key field that holds the name of the state and a pointer to class *sc_method_handle*. *FSMnode* class is not

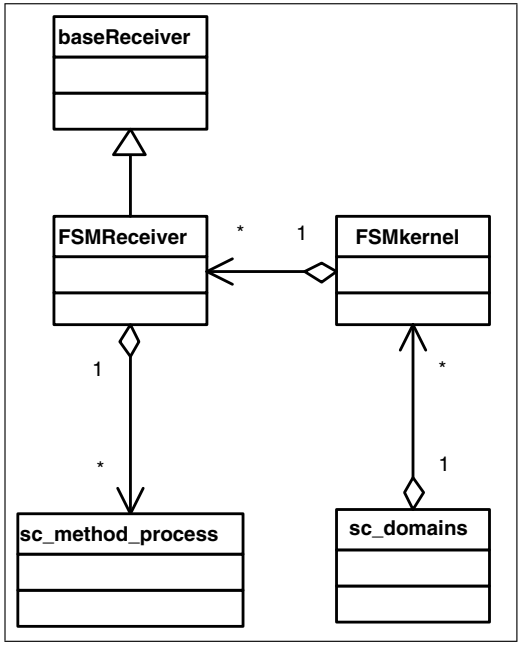


Figure 4.3. FSM Implementation Class Hierarchy

presented in Figure 4.3 because FSM transitions are explicitly defined within the state or entry function of that module. Hence, there is no need for communication between states other than signaling the FSM with the next state to carry out FSM simulation, which is done within the entry function of the process.

The *sc_domains* class in both Figure 4.2 and Figure 4.3 is a toplevel encapsulation class that implements the Application Protocol Interface (API). The purpose of *sc_domains* is to allow different kernels to interact with each other. Another use of the API is to allow for multiple models of the same MoC, for example three SDF models to function in the same model. Additional information regarding the API is discussed in Chapter 8.

1. MoC Specific Ports and Channels

MoC-specific ports and channels are needed due to the differences in communication protocols of the new kernels and the Discrete-Event kernel in SystemC. For example, SDF channels do not require the channels to generate events when pushing data onto a channel. Furthermore, the CSP rendez-vous semantics require its own event and event handling mechanism because the channels themselves play an important role in

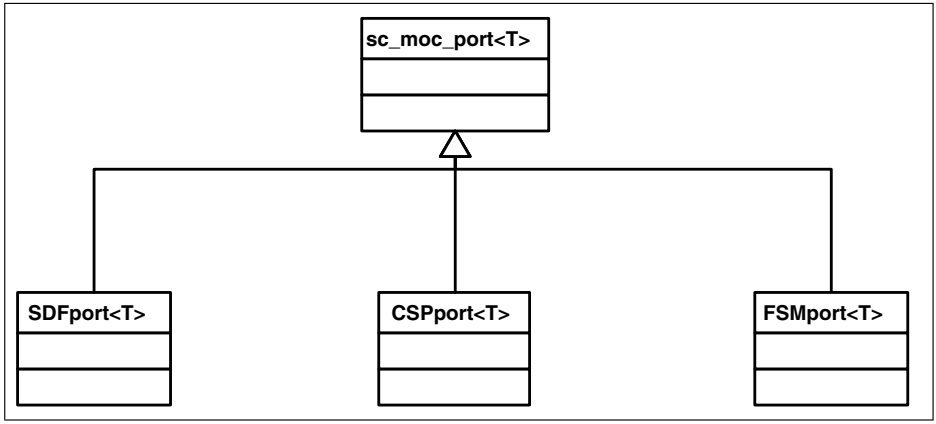


Figure 4.4. *sc_moc_port* Implementation Class Hierarchy

Table 4.1. Some Member functions of class *sc_moc_port*

Member Function	Purpose
operator()	Overloaded operator for binding a *sc_moc_channel* to a port.
getport()	Returns a pointer to the channel bound to this port.
push(...)	Inserts a value onto the channel
pop(...)	Returns the first value on the channel

the suspension and resumption of processes rather than simply providing a medium through which data is transferred.

MoC-Specific Ports

Figure 4.4 describes a class hierarchy for ports that accommodates multiple MoC communication. Basic functionality of the ports is implemented in *sc_moc_port* class and specializations are implemented in the derived classes. All the derived port classes are also polymorphic by making them *template* classes.

Listing 4.1 shows the class declaration and definition for *sc_moc_port*. The private data member of this class is a pointer to an *sc_moc_channel* object called *port*. This variable addresses the channel that is bound to this port. The roles of the member functions are shown in Table 4.1. Most of the generic roles of the port are implemented in the base class. If there is a need to add specific functionality for a particular port or channel then it can be added to the derived class.

Listing 4.1. class *sc_moc_port*

```
template <class T>  class sc_moc_port
{
  private:
    sc_moc_channel<T> * port;
    void bind(sc_moc_channel<T> * p);

  public:
    sc_moc_port<T>();
    ~sc_moc_port<T>();
    sc_moc_port(sc_moc_channel<T> * p);

    void operator () (sc_moc_channel<T> * port) { bind(port); };
    void operator () (sc_moc_channel<T> & port) { bind(&port);};
    T * getport();

    // Vector push/pop functions
    void push(T p);
    void push(T * p);
    T pop();

    void print()  { if (port != NULL) { port->print(); } };
};

template <class T >
  sc_moc_port<T>::sc_moc_port() {
    port = NULL;
    };

template <class T >
  sc_moc_port<T>::~sc_moc_port()  { };

template <class T >
  sc_moc_port<T>::sc_moc_port(sc_moc_channel<T> * p)  { port = p
      ; };

template < class T >
  void sc_moc_port<T>::bind(sc_moc_channel<T> * p)  { port = p
      ; };

template < class T>
  T * sc_moc_port<T>::getport() { return port; };

template < class T>
  void sc_moc_port<T>::push(T p)  { port->push(p);};
template < class T>
  void sc_moc_port<T>::push(T * p)  { port->push(p);};

template < class T>
  T sc_moc_port<T>::pop() { return(port->pop());  };
```

MoC-Specific Channels

Similarly, channels for the these MoC-specific ports follow a hierarchy displayed in Figure 4.5. The base class is *sc_moc_channel* from which the *SDFchannel*, *CSPchannel* and *FSMchannel* are all derived. Basic functions of a channel are implemented in the base class *sc_moc_channel* and MoC-specific channel implementations are contained in their respective derived class. The *SDFchannel* and *FSMchannel* are used to transport

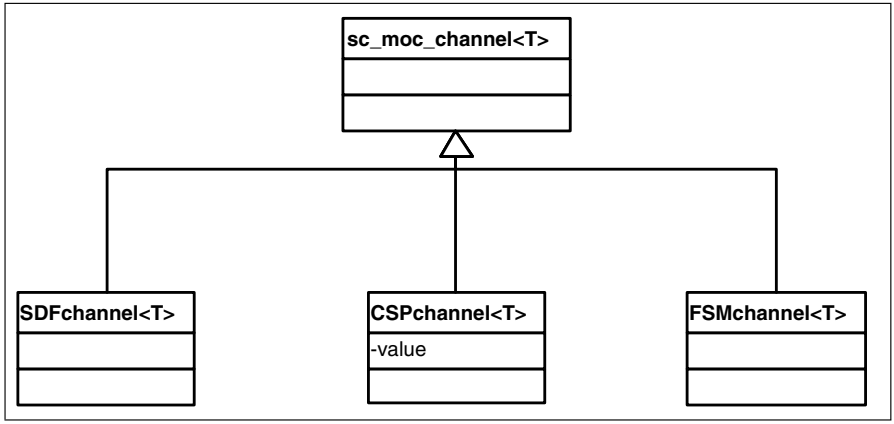


Figure 4.5. *sc_moc_channel* Implementation Class Hierarchy

data from one node to another. However, a *CSPchannel* plays an integral role in the rendez-vous communication, which requires additional implementation details to the channel. Figure 4.5 shows a data member *value* in *CSPchannel* that holds the value to be transferred once rendez-vous synchronization occurs. This *value* is of templated type allowing for all values of different types to be transferred. This is an example where specialization of an MoC-specific channel is done. Nonetheless, note that these MoC-specific channels and ports have no relation with *sc_channel* or *sc_port* and MoC-specific channels and ports are implemented using C++ data types.

Listing 4.2 displays the declaration and definition of the class *sc_moc_channel*. A list data structure is used to preserve the tokens pushed onto a channel. We use the *vector* class from STL. The *push(...)* member function inserts a value into the list and *pop()* returns the first value in the list. Note that both *sc_moc_port* and *sc_moc_channel* are template classes allowing for any type of data to be transferred through these ports and channels.

Note about SDF Ports and Channels

In Chapter 5, SDF ports and channels are not presented. However, the implementation does have SDF ports and SDF channels similar to FSM ports and FSM channels in Chapter 7. Class *SDFport* derives from *sc_moc_port* without any need for specialization since its purpose is to only transfer data and the *SDFchannel* is a derived class from *sc_moc_channel*. They are both templated classes as shown in Figure

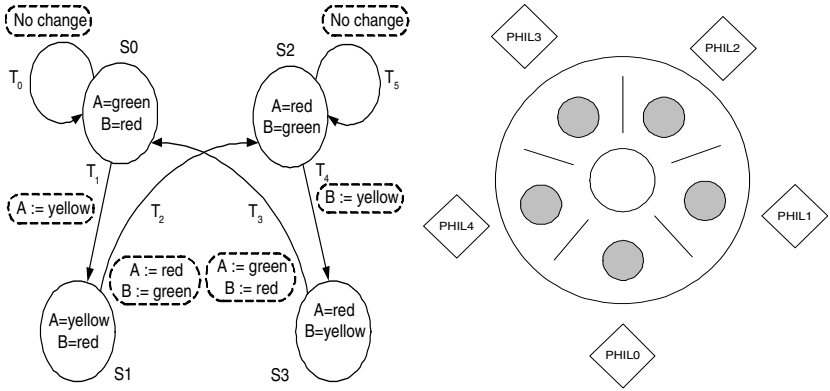
Listing 4.2. class *sc_moc_channel*

```
template <class T>
  class sc_moc_channel {
  private:
    vector<T> * mainQ;

  public:
    sc_moc_channel<T >(); // constructor
    ~sc_moc_channel<T>(); // destructor

    // Vector push/pop functions
    void push(T p);
    void push(T * p);
    T pop();

};

template <class T >
  sc_moc_channel<T>::sc_moc_channel() { mainQ = new vector<T>();
        };

template <class T >
  sc_moc_channel<T>::~sc_moc_channel()  { delete mainQ; };

template < class T>
  void sc_moc_channel<T>::push(T p) { mainQ->push_back(p);  };

template < class T>
  void sc_moc_channel<T>::push(T * p) { mainQ->push_back(*p); };

template < class T>
  T sc_moc_channel<T>:: pop() {
    T newT = mainQ->front();
    mainQ->erase(mainQ->begin());
    return (newT);
  }
```

4.4 and Figure 4.5. We provide SDF examples that employ the SDF ports and channels at our website [36].

baseReceiver class

baseReceiver currently only holds the receiver type, indicating whether an *FSMReceiver* or *CSPReceiver* has been derived. However, the usage of this base class can extend to encompass common data structures and helper functions. One such use of the *baseReceiver* can be to allow for implementation of the data structure required to represent MoCs that require a graph construction. We consider a graph like structure for SDF, FSM and CSP and currently we employ individual representations for each kernel shown in Figure 4.6. However, our current implementation does not unify the idea of representing commonly used data structures in the *baseReceiver* class and leave it aside as future work.



(a) FSM Traffic Light Controller (b) CSP Dining Philosopher

(c) SDF FIR

Figure 4.6. Graph-like Representation for SDF, FSM, CSP

2. Integration of Kernels

Kernel integration is a challenging task especially for kernels based on MoCs that exhibit different simulation semantics other than the existing DE scheduler semantics of SystemC. The MoC implementation chapters discuss the addition of a particular MoC in SystemC and documentation is provided describing the integration of these different MoCs. MoCs such as SDF and FSM are easy to integrate with the reference implementation and themselves, whereas CSP requires an understanding of QuickThreads [35] and the coroutine packages in SystemC. Integration of the SDF and FSM kernels are relatively straightforward, requiring minor additions to the existing source with the usage of Autoconf/Automake [21, 22]. In Appendix B we describe a method of adding newly created classes to the SystemC distribution using Autoconf/Automake. This approach is used for all MoC integration. However, the CSP kernel integration is non-trivial, which we describe in detail in Chapter 6.