Chapter 6

COMMUNICATING SEQUENTIAL PROCESSES KERNEL IN SYSTEMC

Any multi-MoC framework designed to model and simulate embedded systems, or any other complex system composed of concurrently executing components which are communicating intermittently, needs to implement some MoCs that are geared for specific communicating process models. Current SystemC reference implementation lets the user create concurrently executing modules using *SC_THREAD()* or *SC_CTHREAD()* constructs. Modules that have such threads in them communicate via channels which are usually of the *sc_fifo*, *sc_mutex* and other predefined channel types and their derivatives. These channels have blocking and non-blocking read/write interfaces that the threads can call to block themselves or attempt communication with other threads. These thread constructs also can synchronize with clock signals, or events using *wait()* or *wait_until()* function calls directly, or through read/write calls on one of the channel types. Clearly, such threading mechanisms and structures are provided with the Discrete-Event (DE) kernel in mind. What if one wants to model software components which are not necessarily synchronized with a global clock when they suspend or do not need to synchronize with events that are created at the DE kernel level? Often, designers want to model a software system without the notion of clock based synchronization, and later on refine the model to introduce clocks. For such untimed models of concurrent components, designers would much prefer a different MoC than the clock-based DE MoC.

In [28], Communicating Sequential Processes (CSP) is introduced as a model of computation for concurrency that originally dates back to 1978 [27]. In this MoC, sequential processes are combined with process combinators to form a concurrent system of communicating components.

The protocol for communication in such an MoC is fully synchronous as opposed to data flow networks. For example, in data flow networks, buffers in the channels connecting two computing entities are assumed, and, based on buffer size, the computations proceed asynchronous to each other, leaving the communicable data at the buffers for the other components to pick up as and when ready. Of course, in real implementations buffers are of limited size and hence often times requires process blockings. In CSP, the communication happens through a *rendez-vous* mechanism [27]. This necessitates synchronization at the data communication points between the processes, as buffering is not allowed on the channels, and the communicating processes both need to be ready to communicate for communication to take place. If one of the two communicating processes is not ready, the other blocks until both are ready. This imposes a structure and semantics that is amenable to *trace theory*, and in later work to *failure-divergence* semantics. Such theoretical underpinnings make this MoC quite useful for formal analysis, and in recent times formal verification and static analysis tools for CSP models have appeared [40].

CSP provides a convenient MoC for creating a system model which consists of components that need to communicate with each other and their communication is based on synchronous rendez-vous, rather than buffered asynchronous communication. Refining such models with a clocked synchronous model later on is easier than refining a fully asynchronous model. Moreover, the models built can be formally analyzed for deadlock and livelock kind of problems more easily. We therefore picked CSP MoC as one of the first concurrency related MoC for our extension of SystemC.

Rendez-vous Communication

Implementation of the Communicating Sequential Processes Model of Computation requires understanding of rendez-vous communication protocol. Every node or block in a CSP model is a thread-like process that continuously executes unless suspended due to communication. The rendez-vous communication protocol dictates that communication between processes only occurs when both the processes are ready to communicate. If either of the processes is not ready to communicate then it suspends until its corresponding process is ready to communicate, at which it is resumed for the transfer of data.

Figure 6.1 illustrates how the rendez-vous protocol works. T1 and T2 are threads that communicate through the channel labelled C1. T1 and T2 are both runnable and have no specific order in which they are executed. Let us consider process point 1, where T1 attempts to put a value

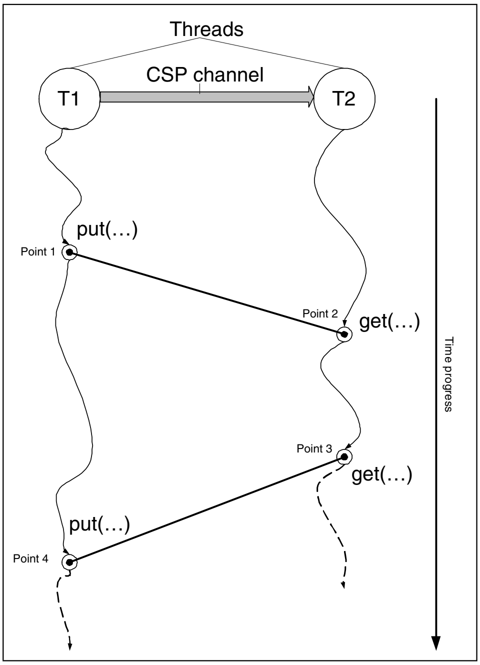

Figure 6.1. CSP Rendez-vous Communication

on the channel C1. However, process T2 is not ready to communicate, causing T1 to suspend when the *put(...)* function within T1 is invoked. When process T2 reaches point 2 where it invokes the *get(...)* function to receive data from C1, T1 is resumed and data is transferred. In this case T2 receives the data once T1 resumes its execution. Similarly, once T2 reaches its second invocation of *get(...)* it suspends itself since T1 is not ready to communicate. When T1 reaches its invocation of *put(...)*, the rendez-vous is established and communication proceeds. CSP channels used to transfer data are unidirectional. That means if the channel is going from T1 to T2, then T1 can only invoke *put(...)* on the channel and T2 can only invoke *get(...)* on the same channel.

1. Implementation Details

We present some design considerations in this section followed by the data structure employed for CSP and implementation details.

1.1 Design Considerations and Issues

Careful thought must be given to the inclusion of a CSP kernel in SystemC. This is necessary because CSP is an MoC disjoint from conventional hardware models. Though CSP is more generally considered a software MoC, it is an effective MoC when targeting models for concurrency. Clearly, the semantics of CSP are different from the semantics of a Discrete-Event MoC. This implies that, unlike the SDF implementation in SystemC where we targeted for the simulation semantics to remain exactly the same as the DE semantics, in CSP we want them to be completely distinct. Therefore, the Evaluate-Update paradigm is not employed in the implementation.

1.2 Data Structure

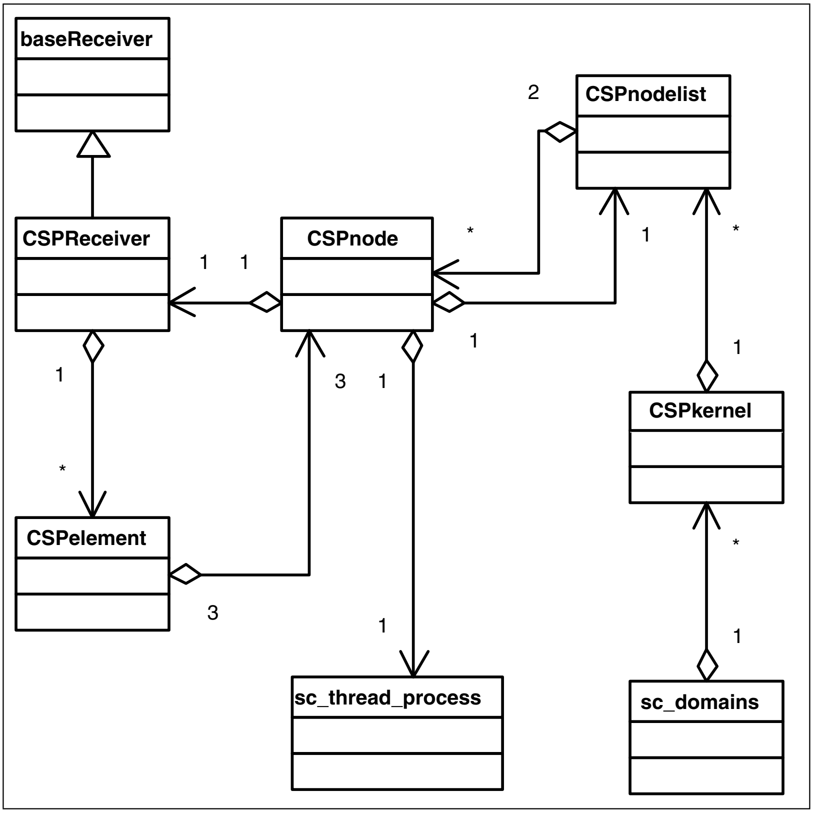

Figure 6.2. CSP Implementation Class Hierarchy

Chapter 4 familiarizes the reader with general implementation class hierarchies that present a minimal organization structure followed by the CSP kernel. This section describes implementation of the class hierarchy shown in Figure 6.2.

A *baseReceiver* class preserves basic information about the receiver that inherits from the *baseReceiver*. This class presently only holds the type of the inheriting receiver, but this can be extended to encompass

Listing 6.1. class baseReceiver

```
class baseReceiver {
  private:
    receiverType type;

  protected:
    receiverType getType();
    void setType(receiverType t);
    void setCSP();

  public:
    baseReceiver();
    ~baseReceiver();

};
```

common functionality as described in Chapter 4. Listing 6.1 shows the *baseReceiver* class with an enumerated *receiverType* data type. Variable *type* is set via the derived class, identifying the derived class as a CSP receiver by the use of *setCSP()* function.

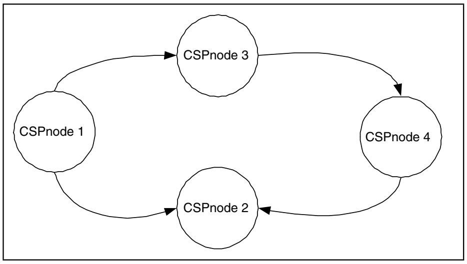

Figure 6.3. Simple CSP model

CSP models require a data structure that represents a graph, which we call a CSP graph (CSPG). The *CSPelement* class is responsible for encapsulating information used to construct this CSPG. Figure 6.3 shows an example of a CSPG. The graph representation is implemented by a list of pointers to objects of type *CSPelement*, which is discussed later in this section. However, for the purpose of creating this CSPG, each object of *CSPelement* contains a pointer to the *CSPnode*s that this *CSPelement* is connected to and from.

From Listing 6.2 *toNode* and *fromNode* point to the objects of type *CSPnode* (defined later in this section) distinguishing the direction of the communication as well. There are two Boolean flags called *putcalled* and *getcalled* that store the state of the channel. The *putcalled* Boolean value is set to *true* if a corresponding *CSPnode* connected to this channel invokes the *put(...)* function call. Similarly, *getcalled* is set when the

Listing 6.2. class CSPelement

```
class sc_module;
class CSPnode;

class CSPelement {

  private:
    CSPnode * me;
    CSPnode * toNode;
    CSPnode * fromNode;
    static int id;
    bool putcalled;
    bool getcalled;
    csp_event * ev; // store the event that this element is
        going to be triggered on

  public:

    CSPelement();
    ~CSPelement();

    CSPelement(CSPnode * from, CSPnode * to, int id );
    CSPelement(CSPnode * from, CSPnode * to );
    void setid(int i);
    void setto(CSPnode * to);
    void setfrom(CSPnode * from);
    int getid();

    bool getput();
    bool getget();
    void setput(bool p);
    void setget(bool g);

    void setev(csp_event * e);
    csp_event * getev();
    void clearev();

    CSPnode* getme();
    void setme(CSPnode * m);

    CSPnode* getto();
    CSPnode* getfrom();
    CSPnode* get_resume_ptr(CSPnode * myself);
    CSPnode* get_suspend_ptr(CSPnode* myself);
    string * getmyname(CSPnode * myself);

    //overloaded operators
    bool operator==(const CSPelement & a) ;
    bool amIfrom(CSPnode * from);

    friend ostream& operator << (ostream& os, CSPelement & p);
        //output
};
```

get(...) function is invoked by its corresponding CSP process. Another Boolean variable typedefed to *csp_event* represents whether there exists an event on the channel. If an event exists then one of the processes connected to this channel was suspended. SystemC events are not used for *csp_event*, but regular *bool* data types. This avoids the use of SystemC's DE semantics and events.

Other than general set and get functions for the private members of this class, the important member function is the overloaded *equals* operator. The implementation of this overloaded operator compares the *fromNode* and *toNode* to verify that the *CSPelement* objects on both sides of the *equals* operator have the same addresses for the *fromNode* and *toNode*. If they do, then a particular channel or *CSPelement* that connects two *CSPnode*s is found. The responsibility of *CSPelement* is exactly the same as that of a channel. This is a result of adhering to the general implementation hierarchy, where the CSP channels are effectively represented by *CSPelement* objects. Hence, we inherit *CSPelement* in *CSPchannel*, which is discussed later. This is the mechanism that we employ in searching for the channels through which communication occurs. However, this imposes a limitation that there can only be a maximum of two channels between the two same *CSPnode*s. This gives rise to a problem that if there exists two channels in the same direction between the two same nodes, then according to the equals operator, they will be indistinguishable. Thus, we limit the users to only one channel in the same direction between two *CSPnode*s. We justify this implementation in the following manner:

- By allowing for a templatized data transfer communication that can transfer a data type defined by the user. This allows the user to pass in different values through the communication channel through the user defined data type.

- A single *CSPchannel* can result to multiple suspension points with multiple calls to *get(...)* or *put(...)*.

Figure 6.3 shows a simple CSP model with four CSP processes that are connected via channels. The analogous representation of this simple CSP model using our data structure is shown in Figure 6.4, which shows how objects of *CSPelement* are used to construct a CSPG. The list holding the *CSPelement*s is the *CSPReceiver*. *CSPReceiver* objects are data members of a *CSPnode* that are composed with *CSPelement* objects.

Figure 6.4 shows four *CSPnode*s and their respective *CSPelement*s for the purpose of providing a connection between two CSP processes. The gray box displays objects of *CSPReceiver*. The role of the receiver is simply to encapsulate the *CSPelement*s as shown in Figure 6.4. A simple data structure is employed to represent the CSPG. We employ C++ STL *vector<...>* class to store the addresses of every *CSPelement* inserted in the CSPG and iterate through the list to find the appropriate

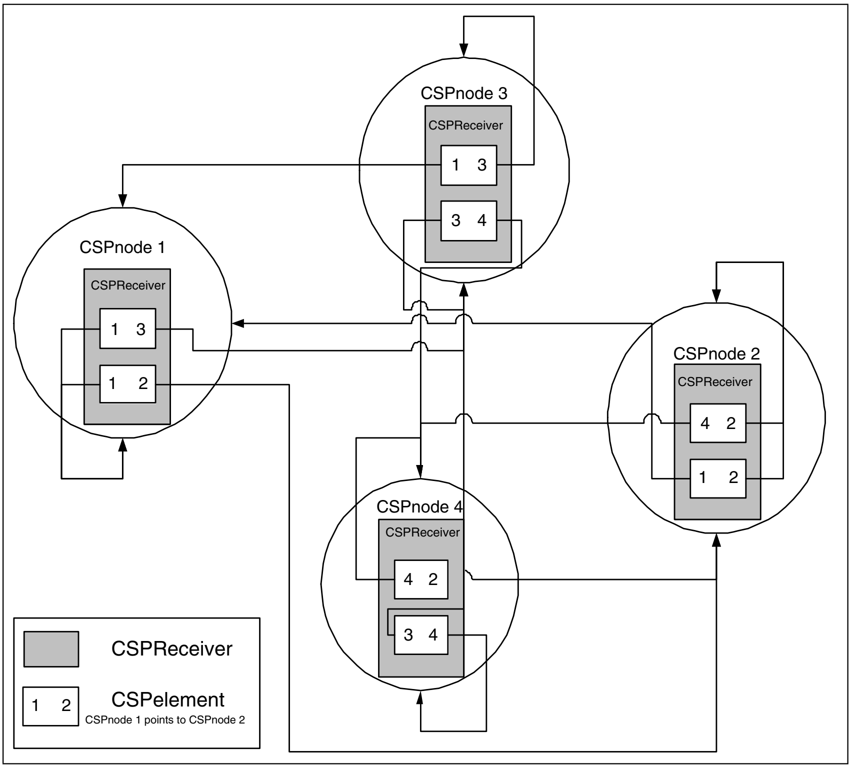

Figure 6.4. Implementation of a Simple CSP Model

channel for communication when required. Every *CSPnode* has its own *CSPReceiver* object that contains the *CSPelement*s that address that particular CSP process. Listing 6.3 displays the class definition describing the *elementlst* as the container of the *CSPelement* addresses along with a private helper function that is used to traverse through the list and identify the requested channel.

We discuss some of the important member functions from this class and their input and output arguments.

put(...):

Inputs:

- A pointer to the *CSPelement* to identify what channel it is to be passed on to.
- The *CSPnode* that is responsible for sending this token.

Outputs:

Listing 6.3. class *CSPReceiver*

```
class CSPReceiver: public baseReceiver {

private:
  vector<CSPelement*> elementlst;
  int id;

  // private helper functions
  CSPelement* findElement(CSPelement * e);

public:
  CSPReceiver();
  ~CSPReceiver();

  // overloaded Constructors
  CSPReceiver(CSPnode * fromNode, CSPnode * toNode);

  //functions
  void get(CSPelement * e, CSPnode * me);
  void put(CSPelement * e, CSPnode * me);

  void push_into(CSPelement * e);
  friend ostream& operator<<(ostream& os, CSPReceiver & p) ;

  // event finders
  csp_event * getevent(CSPelement * el);
  void setevent(CSPelement * el, csp_event *ev);

  void suspendProc(CSPelement * e, CSPnode * me);
  void resumeProc(CSPelement * e, CSPnode * me);
};
```

- The process suspends if a *get(...)* has not been called on the channel.

get(...):

Inputs:

- A pointer to the *CSPelement* that a token is to be received from.
- The address of the *CSPnode* making the *get(...)* invocation.

Outputs:

- If a *put(...)* has been called the suspended process that called the *put(...)* is scheduled for execution (resumption).

suspendProc(...): Suspends the currently executing thread.

Inputs:

- A pointer to the *CSPelement* that requires suspension due to rendez-vous protocols.

- A pointer to the *CSPnode* that is to be suspended.

Outputs:

- The *CSPnode* currently executing suspends itself.

resumeProc(...): Resumes a particular thread for execution.

Inputs:

- A pointer to the *CSPelement* that is to be resumed due to rendez-vous protocols.
- A pointer to the *CSPnode* that is to be resumed.

Outputs:

- The *CSPnode* is scheduled for resumption.

It may seem redundant to supply these functions with the owner of the call, where the owner is the process invoking the member function. However, this is necessary because every *CSPnode* contains all the *CSPelements* that addresses that process, either as a *fromNode* or a *toNode*. Furthermore, the direction is preserved when inserting the address of the *CSPelement* objects in their respective receiver lists. This is to allow the process to know whether it is the calling process or the called process.

To avoid a convoluted written explanation, let us consider Figure 6.3 where the direction of the communication is from CSPnode 1 and towards CSPnode 3. Our implementation adds a pointer in CSPnode 1's receiver and the same pointer in CSPnode 3's receiver pointing to an object of *CSPelement* whose *fromNode* points to CSPnode 1 and *toNode* is CSPnode 3. For the purpose of the *CSPnode* knowing the direction of communication, it is necessary to compare the process's pointer to both the *fromNode* and *toNode* to realize the direction of communication.

Listing 6.4 defines the *CSPnode* class that encapsulates the *CSPReceiver* as shown in Figure 6.4. Other important private members of this class are *sc_thread* and *my_thread_list*. *sc_thread* holds a pointer to SystemC's *sc_thread_process* object and *my_thread_list* is a pointer to an object that contains a list of *CSPnodes* in a model. These private data members are used during the simulation of the CSP model. The remainder of the member functions are mandatory *set(...)* and *get(...)* functions.

A CSP channel implemented as a class called *CSPchannel* inherits from base class *sc_moc_channel*, but *CSPchannels* must also support rendez-vous communication as well as the capability to transfer data. For this reason, the *CSPchannel* is specialized. Listing 6.5 shows the definition of this class.

Listing 6.4. class *CSPnode*

```
class CSPnodelist;
class CSPnode {

private:
  CSPReceiver * cspbox; // one CSPnode has one receiver
  int cspid;
  ProcInfo * process;
  sc_thread_handle sc_thread;
  CSPnodelist * my_thread_list;

  // helper functions
  int getid();

public:
  CSPnode();
  ~CSPnode();

  // Set up Process Information
  void setprocaddr(void * a);
  void setprocname(string * n);
  void setprocname(const string & n);
  void* getprocaddr();
  string* getprocname();

  // setup the link between two or more nodes
  void points_to(CSPnode* to);
  void points_to(CSPnode * to, CSPelement * el);
  void points_to(CSPnode & to, CSPelement & el);

  //member functions
  bool send();
  bool send(CSPelement * sendTo);
  void send(CSPelement & sendTo);
  void get(CSPelement * getFrom);
  void get(CSPelement & getFrom);
  bool suspend();
  void portbind(CSPelement *e);

  // set which CSPnodelist it belongs to
  void set_my_thread_list(CSPnodelist * my_list);
  CSPnodelist * get_my_thread_list();

  void print();

  void setnodeev(CSPelement * thisNode, csp_event * e);
  csp_event * getnodeev(CSPelement * getFrom);

  // after execution reschedule immediately
  void reschedule();

  void setmodule(sc_thread_handle mod);
  sc_thread_handle getmodule();
};
```

Notice from Figure 6.5 that multiple inheritance is used to define *CSPchannel*. Inheritance from *sc_moc_channel* and *CSPelement* provides functionality and data structure available in both these base classes. From an object oriented programming sense, the *CSPelement* actually defines a channel between two CSP processes. Thus, the relationships of

Listing 6.5. class *CSPchannel*

```
template <class T> class CSPchannel :
  public CSPelement, public sc_moc_channel<T>
{
  public:

    CSPchannel<T>() {};
    ~CSPchannel<T>() {};

  void push(T & val, CSPnode & node);
  T get(CSPnode & node);
};

template <class T>
  void CSPchannel<T>::push( T & val,CSPnode & node) {
    sc_moc_channel<T>::push(val);
    node.send((CSPelement*) this);

  };

template <class T>
  T CSPchannel<T>::get(CSPnode & node)  {
    node.get((CSPelement*)this);
    return (sc_moc_channel<T>::pop());
  };
```

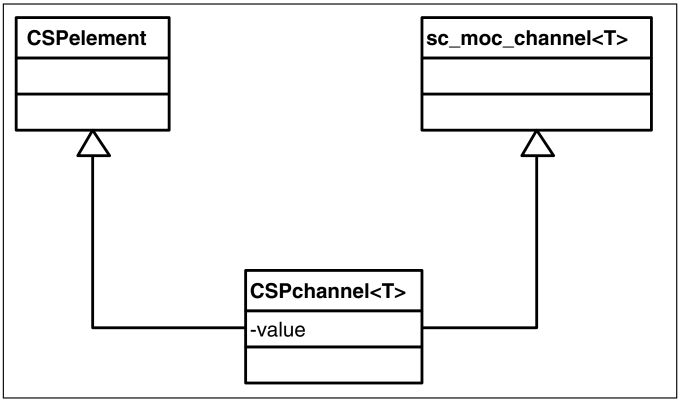

Figure 6.5. Class diagram for *CSPchannel*

CSPchannel is one of an “is a” with both *sc_moc_channel* and *CSPelement.* The member functions in *CSPchannel* are shown in Table 6.1.

Table 6.1. Member function for class *CSPchannel*

Member Function	Purpose
push(...)	Attempts to send a token on the channel
get(...)	Attempts to receive a token from the channel

It follows that there is a need to specialize the *CSPport* class such as to support this specialized *CSPchannel*. Using the *sc_moc_port* base class data structure, *CSPport* appropriately calls member functions of *CSPchannel* when a value is to be inserted or extracted. Listing 6.6 displays the class definition for *CSPport*. The implementation of the *CSPport* class serves the basic purpose of allowing two *CSPnode*s visibility of the *CSPchannel* that connects them. We have implemented overloaded *()* operators to allow CSP port binding. However, we do not perform any checks for port binding errors.

Listing 6.6. class *CSPport*

```
template <class T> class CSPport : public sc_moc_port<T> {
  public:
    CSPport<T>() {};
    ~CSPport<T>() {};    CSPelement & read();
    void push(T & p, CSPnode & node);
    T get(CSPnode & node);

};

template <class T >
void CSPport<T>::push(T  & p, CSPnode & node)
{
  CSPchannel<T> * castchn = static_cast< CSPchannel<T> * > (port
      );
  if (port != NULL) {
    castchn->push(p, node);
  }
};

template <class T >
T CSPport<T>::get(CSPnode & node)  {
  CSPchannel<T> * castchn = static_cast< CSPchannel<T> * > (port
      );
  return (castchn->get(node));
  };
```

The *CSPnodelist* class shown in Listing 6.7 can be considered to be the class that defines the CSP simulator object in SystemC. Hence, an object of *CSPnodelist* performs the simulation for CSP. The private members are simply two *vector<...>* lists where *nodelist* is the list of pointers to all the *CSPnode*s and *runlist* is a list of the runnable CSP processes. Though the *runlist* is of type *vector<...>* we have implemented a queue with it. This behavior is necessary to correctly simulate a CSP model. Other private members are pointers to the coroutine packages used to implement QuickThreads [35] in SystemC. *m_cor* identifies the executing simulation context's coroutine whereas *m_cor_pkg* is a pointer to a file static instance of the coroutine package through which blocking and resumption of thread processes can be performed. For further details about QuickThread implementation in SystemC please refer to Appendix A.

Coroutine is SystemC's implementation of the QuickThread core package as the client package.

Some of the important member functions are listed below:

void push_runnable(CSPnode & c) The *CSPnode* is pushed onto the *runlist* such that it can be executed.

CSPnode * pop_runnable() Retrieves the top runnable thread.

void next_thread() Selects the next CSP process to execute.

void sc_csp_switch_thread(CSPnode * c) Used in blocking the currently executing thread and resuming execution of the thread identified by the pointer *c*.

sc_cor* next_cor() Retrieves a pointer to the next thread coroutine to be executed.

Implementation details of these classes are not presented, but we direct the reader to refer to implementation details available at our website [36]. This brief introduction of the CSP data structure allows us to proceed to describing how the CSP scheduling and simulation is performed. For some readers it may be necessary to refer to Appendix A where we describe the coroutine package for SystemC based on [35].

2. CSP Scheduling and Simulation

Table 6.2. Few Important Member Functions of CSP Simulation class *CSPnodelist*

Method / Variable name	Maintained by	
	CSP Kernel	QuickThread Package
runlist	✓	-
nodelist	✓	-
m_cor_pkg	✓	✓
m_cor	✓	✓
push(...)	✓	-
push_runnable(...)	✓	-
sc_switch_switch_thread(...)	✓	-
pop_runnable(...)	✓	-
next_cor(...)	✓	-
run_csp(...)	✓	-

Simulation of a CSP model uses a simple queue based data structure that contains pointers to all the *CSPnodes*. This queue is constructed by using C macros that work similar to the existing *SC_THREAD()* macros. We introduce the macro *SC_CSP_THREAD()* that takes three

Listing 6.7. class CSPnodelist

```
class CSPnodelist {

  private:
    vector<CSPnode* > * runlist;
    vector<CSPnode* > * nodelist;

  public:
    CSPnodelist();
    ~CSPnodelist();

    void push_runnable(CSPnode & c);
    void push(CSPnode & c);

    void next_thread();

    CSPnode * pop_runnable();
    void removefront();

    //sizes of lists
    int nodelist_size();
    int runnable_size();

    void csp_trigger();
    void runcsp(CSPnodelist & c);
     sc_cor_pkg* cor_pkg()
        { return m_cor_pkg; }
    sc_cor* next_cor();

    vector<CSPnode*> * getnodelist();
    vector<CSPnode*> * getrunlist();
    void init();
    void clean();
    void initialize(bool nocrunch);

    void sc_csp_switch_thread(CSPnode * c );
    void print_runlist();

    void push_top_runnable(CSPnode & node);

  private:
    sc_cor_pkg*                   m_cor_pkg; // the simcontext's
         coroutine package
    sc_cor*                       m_cor;     // the simcontext's
         coroutine

};
```

arguments: the entry function, the *CSPnode* object specific for that *SC_CSP_THREAD()* and the *CSPnodelist* to which it will be added. This macro calls a helper function that registers this CSP thread process by inserting it in the *CSPnodelist* that is passed as an argument.

Invoking the function *runcsp(...)*, initializes the coroutine package and the current simulation context is stored in the variable *main_cor*. The simulation of the CSP model starts by calling the *sc_csp_start(...)* function. Table 6.2 shows a listing of some important functions and variables and whether the CSP kernel or the QuickThread package manages them. The variable *m_cor_pkg* is a pointer to the file static instance of the coroutine package. This interface for the coroutine package is better explained in Appendix A. All thread processes require being prepared for simulation. The role of this preparation is to allocate every thread its own stack space as required by the QuickThread package. After this preparation, the first process is popped from the top of the *runlist* using *pop_runnable(...)* and executed. The thread continues to execute until it is either blocked by executing another thread process or it terminates. This continues until there are no more processes on the *runlist*.

Listing 6.8. class csp_trigger() function

```
void CSPnodelist::csp_trigger() {
  while (true)  {
      sc_thread_handle thread_h = pop_runnable()->getmodule();
      removefront();
      while( thread_h != 0 && ! thread_h->ready_to_run() ) {
        thread_h = pop_runnable()->getmodule();
        removefront();
      }
      if( thread_h != 0 ) {
        m_cor_pkg->yield( thread_h->m_cor );
      }

      if( runnable_size() == 0 ) {
        // no more runnable processes
        break;
      }
  };
}
```

We present the function *csp_trigger()* in Listing 6.8 that is responsible for performing the simulation. The *pop_runnable()* function extracts the topmost pointer to a *CSPnode* that has an *sc_thread_handle* as a private member, which is retrieved by invoking the *getmodule()* member function. The *m_cor_pkg→yield(thread_h→m_cor)* function invokes a function implemented in the *sc_cor_qt* class. This *yield(...)* function is responsible for calling a helper function to switch out the currently executing process, saving it on its own stack and introducing the new process for execution. The process coroutine is sent by the *thread→m_cor* argument. A check is done if the runnable queue is empty and then the simulation is terminated. However, most CSP processes are suspended during their execution, which requires brief understanding of how blocking is performed using QuickThreads. For most readers it will suffice to explain that when a process suspends via the *suspendProc(...)* function, the state of the current process is saved and a helper function called *next_cor()* is invoked. The *next_cor()* returns a pointer of type *sc_cor* which is the coroutine for the next thread to execute.

Listing 6.9. function next_cor() function

```
sc_cor* CSPnodelist::next_cor()
{
  sc_thread_handle thread_h = pop_runnable()->getmodule();
  removefront();
  while( thread_h != 0 && ! thread_h->ready_to_run() ) {
    thread_h = pop_runnable()->getmodule();
    removefront();
  }
  if( thread_h != 0 ) {
    return ( thread_h->m_cor );
  } else
  return m_cor;
}
```

Implementation of the *next_cor()* function is similar to the *csp_trigger()* function. This is because once a CSP process is suspended, the next process must continue to execute. So, *next_cor()* implements a similar functionality as *csp_trigger()* with the exception of calling *yield(...)* on the process to execute, and the coroutine is returned instead. Furthermore, if there are no more processes on the *runlist*, then the main coroutine of the simulation is returned by returning *m_cor* as shown in Listing A.11. Therefore, the suspension of processes is in essence performed by yielding to another process, where QuickThreads serve their purpose by making it relatively simple for blocking of thread processes. Likewise, resumption of the threads is simple as well. Using the coroutine package, resumption is done by rescheduling the process for execution. Therefore, when *resumeProc(...)* is invoked, the address of the process to be resumed is inserted into the *runlist* queue. Once the top of the queue reaches this process, the thread is resumed for execution. During modeling, non-deterministic behavior is introduced by randomization in the user constructed models. According to this implementation, CSP models have the potential for executing infinitely such as the Dining Philosopher problem. We visit the implementation of this example using our CSP kernel for SystemC.

3. Example of CSP Model in SystemC

Early in Chapter 2, we introduced the Dining Philosopher problem. A schematic of the way it can be implemented is shown in Figure 6.6. In this section, we revisit this example and provide the reader with modeling guidelines along with code fragments to describe how it is modeled using our kernel. However, during our earlier discussion, we did not present the possibility of deadlock. A deadlock occurs in the Dining Philosopher problem when for instance every philosopher feels hungry

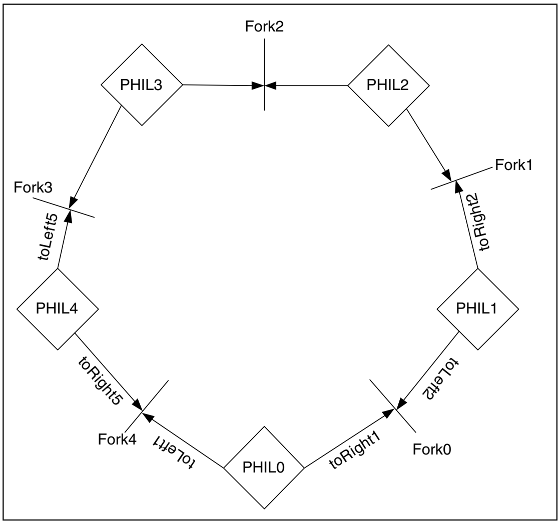

Figure 6.6. CSP Implementation of Dining Philosopher

and picks up the fork to their left. That prevents any of the philosophers eating since two forks are required to eat causing the model to deadlock. We use a simple deadlock avoidance technique where we have a footman that takes the philosophers to their respective seats and, if there are four philosophers at the table, asks the fifth philosopher to wait and seats him only after one is done eating. This is a rudimentary solution, but for our purpose it is sufficient.

We begin by describing the module declaration of a philosopher in Listing 6.10. The original implementation that we borrow is available at [60]. That implementation is a pure C++ based implementation that we modify to make a CSP SystemC example. Each philosopher has a unique *id* and an object of *ProcInfo*. This *ProcInfo* class is implemented as a debug class to hold the address of the process and the name of the process purely for the reasons of output and debugging. The full source description will have the implementation of this class, though we do not describe it since it is not directly relevant to the implementation of the CSP kernel in SystemC. There is an instantiation of a *CSPnode* called *csp* through which we enable our member function invocations for CSP and two *CSPport*s, *toRight* and *toLeft*. The *toRight* connects to the *CSPchannel* that connects the philosopher to the fork on its right and *toLeft* to the one on its left. There are several intermediate functions defined in this module along with the main entry function. The entry function is called *soln()* that is bound to a CSP process through the *SC_CSP_THREAD()* macro.

Listing 6.10. Philosopher Module Declaration

```
SC_MODULE(PHIL) {

  int id;
  int st;
  string strid;
  int   _timeToLive;

  CSPnode csp;
  CSPport<int> toRight;
  CSPport<int> toLeft;

  int * drop;
  int * pick;
  ProcInfo proc;

  void askSeat(int id);
  void getfork();
  void dropfork();
  void soln();
  int getstate();
  void print();

  // footman required for deadlock free solution
  bool reqSeat();

  SC_CTOR(PHIL) {
    st = -1;
    SC_CSP_THREAD(soln, DP, csp)  {
    };
  };
};
```

We begin describing the implementation of the *PHIL* class by displaying the entry function *soln()* as shown in Listing 6.11. Many print statements are inserted to view the status of each of the philosophers and the forks. This is handled by the *print_states()* function. However, the core functionality of the entry function begins by invoking *getfork()*. Listing 6.12 shows the implementation of this function. The *state[x]* array is global and simply holds the state value for every philosopher, which is updated immediately to allow the *print_states()* to output the updated values. The philosopher requests a fork on either the left or right of himself by calling the *get(...)* member function on the port. If the fork is available to be picked up and has been recognized by the *CSPchannel* then the philosopher process will continue execution and request the other fork. However, if the fork is not ready to be picked up, this process will suspend.

Once the philosopher has both the forks in hand, *soln()* goes to its eating state where we simply output *EATING* and wait for a random amount of time defined by functions from [60]. After the eating state, *Listing 6.11.* function soln()

```
void PHIL::soln() {
    int duration = _timeToLive;
    int eatCount = 0;
    int totalHungryTime = 0;
    int becameHungryTime;
    int startTime = msecond();

    while(1) { // ( msecond() - startTime < duration * 1000 ) {

      if ((reqSeat() == true) && ((state[id] != 0) || (state[id] != 6))) {
        becameHungryTime = msecond();
        print_states();
        cout << " PICKING UP FORKS " << endl;
        getfork();
        cout << " DONE PICKING UP FORKS " << endl;
        print_states();
        totalHungryTime += ( msecond() - becameHungryTime );
        eatCount++;
        cout << " EATING " endl;
        state[id] = 3;
        usleep( 1000L * random_int( MeanEatTime ) );
        cout << " DONE EATING " << endl;
        print_states();
        cout << " DROPPING FORKS " << endl;
        dropfork();
        usleep( 1000L * random_int( MeanThinkTime ) );
        cout << " DONE DROPPING FORKS " << endl;
        print_states();
        cout << " THINKING " << endl;
        state[id] = 6;
        usleep( 1000L * random_int( MeanThinkTime ) );
        state[id] = 0;
        print_states();
        --space;
        csp.reschedule();
      } else {
          cout << " STANDING " << endl;
          csp.reschedule();
      }
    }
    state[id] = 7;
    totalNumberOfMealsServed += eatCount;
    totalTimeSpentWaiting += ( totalHungryTime / 1000.0 );
    cout << "Total meals served = " << totalNumberOfMealsServed << "\n";
    cout << "Average hungry time = " <<
    ( totalTimeSpentWaiting / totalNumberOfMealsServed ) << "\n";
};
```

the philosopher enters the state where he attempts to drop the forks by calling *dropfork()* described in Listing 6.13.

Dropping of the forks is modeled by sending a value on the channel which is performed via the *push(...)* on the port. If the *push(...)* is invoked without the corresponding CSP node at the end of the channel ready to accept the token, the process will suspend. Returning back to the entry function, after the forks have been dropped there is a random

Listing 6.12. function getfork()

```
void PHIL::getfork()  {
  if ( numPhil % 2 )  {
    // even-numbered philosophers pick left then right
    state[id] = 1;
    print_states();
    toLeft.get(csp);

    state[id] = 2;
    print_states();
    toRight.get(csp);
  }
  else  {
    // odd-numbered philosopher; pick right then left
    state[id] = 2;
    print_states();
    toRight.get(csp);

    state[id] = 1;
    toLeft.get(csp);
    print_states();
    }
};
```

Listing 6.13. function dropfork()

```
void PHIL::dropfork() {
  // drop left first , then right not that it matters
  state[id] = 4;
  print_states();
  toLeft.push(*drop, csp);
  state[id] = 5;
  print_states();
  toRight.push(*drop, csp);
};
```

usleep(...) that suspends execution for microsecond intervals. This completes the eating process for the philosopher such that he returns to his thinking state followed by a random valued *usleep(...)*. According to the queue based implementation, once the process completes its first iteration of the entry function, it must be rescheduled so that the process address is added onto the *runlist*. We provide the *reschedule()* function that the user must invoke to reinsert the CSP process address into the *runlist*.

For the behavior of the fork, we define the module as shown in Listing 6.14. The *FORK* module also has an *id* to differentiate the different forks on the table, an integer valued variable *queryFork* that represents the state of the fork where 1 means that the fork is down and -1 means the fork is not down. There is an instance of a *CSPnode* object called *csp* and two *CSPport*s called *fromRight* and *fromLeft*. The *fromLeft* port connects to a *CSPchannel* coming from the *toRight* port of a philosopher

Listing 6.14. Module FORK

```
SC_MODULE(FORK) {
  int id;
  int queryFork;
  CSPnode csp;
  CSPport<int> fromRight;
  CSPport<int> fromLeft;
  int * drop;
  int * pick;

  ProcInfo proc;

  void reqFork();
  void addressFork();

  SC_CTOR(FORK) {
    queryFork = 1;
    SC_CSP_THREAD(addressFork, DP, csp);
  };
};
```

and the *fromRight* connects to the neighboring philosopher's *toLeft.* The entry function *addressfork()* is described next.

The *addressFork()* function dictates the fork's behavior. This behavior is dependent on the state of the philosophers. Listing 6.15 shows that there are four cases that have implementation for the fork. Cases 1 and 2 only occur when the fork is down on the table and cases 4 and 5 only occur when the fork is not available on the table. We implemented a function that gets the *id*s of the philosophers that surround the fork. We use simple tricks with the *id* of the forks and philosophers to locate the neighbors as shown in Listing 6.16. Our heuristic for finding the neighbors involves looking to the left of the fork and then the right of the fork. We identify each fork with a corresponding *id* as well. Based on this *id* we locate the *id*s of the neighboring philosophers with sufficient cases to ensure that *id*s of forks with *id* 4 and 0 perform an appropriate wrap around to complete the circular setup as shown in Figure 6.6. The *addressFork()* function checks the state of the neighbors and accordingly either gives itself (the fork) to the philosopher or requests itself back, otherwise it simply does nothing. We list the functionality of the fork as follows:

Case 1: The philosopher to the right has requested a fork so the fork gives itself through the *put(...)* function to the philosopher on the right.

Case 2: The philosopher to the left has requested this fork, so the fork gives itself to the requesting philosopher since the fork is still down for Cases 1 and 2.

Listing 6.15. addressfork() member function

```
void FORK::addressFork()  {
   while(true) {
   // Get my neighbors
   int * nbors = get_my_neighbors(id);
   bool resched = false;
      for (int i =0; i < 2; i++)  {
         cout << ”PHIL_” << nbors[i]+1 << ” FORK_” << id+1 ;
         switch(state[nbors[i]]) {
            case 1: {
               // Guy asks on his Left so Send Right
               if ((i != 0) && (queryFork ==1))  {
                  queryFork = -1;
                  forks[id] = queryFork;
                  state[nbors[i]] = 8;
                  print_states();
                  fromRight.push(*pick, csp);
               }
               break;
            }
            case 2: {
               // Guy asks on his Right so Send Left
               if ((i != 1)  && (queryFork ==1)){
                  queryFork = -1;
                  forks[id] = queryFork;
                  state[nbors[i]] = 9;
                  print_states();
                  fromLeft.push(*pick, csp);
               }
               break;
            }
            case 4: {
               if ((i != 0)  && (queryFork !=1)){
                  queryFork = 1;
                  forks[id] = queryFork;
                  print_states();
                  fromRight.get(csp);
               }
               break;
            }
            case 5: {
               if ((i != 1)&& (queryFork !=1)) {
                  queryFork = 1;
                  forks[id] = queryFork;
                  print_states();
                  fromLeft.get(csp);
               }
               break;
            }
            default:  {
               break;
            };
         };
         cout << ”\t”;
         };
         csp.reschedule();
      delete nbors;
   } // END WHILE

};
```

Listing 6.16. get_my_neighbors function

```
int * get_my_neighbors(int id)  {
  int * nbors = new int[2];

  if ((id != 0) && (id != 4)) {
    nbors[0] = id ;
    nbors[1] = id +1;
  } else  {
    if (id == 0)  {
      nbors[0] = 0;
      nbors[1] = id + 1;

    }else {
      if (id == 4)  {
        nbors[0] = id;
        nbors[1] = 0;
      };
    }
  }
  cout << "----====PHIL_" << nbors[0]+1 << "  FORK_" << id
       + 1 << "    PHIL_" << nbors[1]+1 << "  ====----" << endl;

  return nbors;
};
```

Case 4: The fork was given to the philosopher on the right so request the fork back from the philosopher.

Case 5: The fork was given to the philosopher on the left so this is requested back.

This model of the Dining Philosopher executes infinitely unless the conditions are un-commented in the *soln()* function [Listing 6.11, Line 8] which causes the *while()* loop to execute for a limited number of executions and terminates, causing the philosophers to in essence, die (perhaps die from over eating).

4. Modeling Guidelines for CSP Models in SystemC

There are some basic modeling guidelines that the implementation of the CSP kernel in SystemC imposes. A modeler should follow a particular scheme in constructing such models. To better understand these construction rules we present some basic modeling guidelines as follows:

1 Only use *CSPchannel*s for unidirectional communication as per CSP specifications.

2 Every *SC_MODULE()* can have multiple CSP processes initialized as long as there is no multiplicity in the communication channels between the same two CSP processes.

3 The current version of the CSP kernel requires instantiation of a *CSPnodelist* that is accessible by all modules so the use of the keyword *extern* may be required if separate files are used for creating models.

4 The simulation can be initialized by calling the member function *runcsp(...)* of the *CSPnodelist* object.

5 Simulation begins by invoking *sc_csp_start(...)*.

6 It may be necessary to update global variables such as the *state[x]* array in the Dining Philosopher problem to allow interpretation of immediate behaviors and responses.

7 Non-deterministic behavior may require the use of randomization functions.

5. Example of Producer/Consumer

A trivial example using CSP is the Producer/Consumer model. This model is simple and has two processes, a Producer, a Consumer and one channel between them. The communication direction between the processes goes from the Producer to the Consumer. This example is similar to the *simple_fifo* example in the SystemC distribution. The differences are that the processes are CSP processes and instead of an *sc_fifo* channel between the processes, there is a *CSPchannel*.

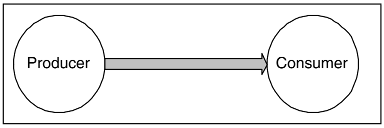

Figure 6.7. Producer/Consumer Example in CSP

Listing 6.17 shows the module declaration for the *PRODUCER* class. Notice an instance of *CSPnode* and a *CSPport*. The *production* pointer holds the string that the Producer sends to the Consumer one character at a time [Listing 6.17, Line 5]. In [Listing 6.17, Line 12], the *at(...)* member function from the *string* class returns a character at the location defined by the argument and stores it in a variable *ch*. This character is pushed onto the channel by invoking the *push(...)* member function on the port that connects the two CSP processes. An instance of *CSPnodelist* labelled as *DP* is accessible by both the *PRODUCER* and *CONSUMER* objects.

The *if* construct repeatedly sends the same string by the Producer when the *sz* string location counter is equal to the number of characters in the string. This makes the model run infinitely. The constructor of *PRODUCER* module sets the *production* pointer to a string and invokes the *SC_CSP_THREAD()* macro for registering this process as a CSP process.

Listing 6.17. *PRODUCER* module declaration

```
SC_MODULE(PRODUCER) {

  CSPnode csp;
  CSPport<char> toConsumer;
  string * production;
  ProcInfo proc;

  void sendChar() {
    int sz = 0;
    while(1)
    {
      char ch = production->at(sz);
      ++sz;
      toConsumer.push(ch, csp);
      //csp.send((token)&ch, toConsumer.read());
      // allow for infinite execution
      if (sz == (signed) production->size())
        sz = 0;
      csp.reschedule();
    }
  };

  SC_CTOR(PRODUCER) {
    production = new string();
    *production = "This is a test string for Produced/Consumer example :]";
    SC_CSP_THREAD(sendChar, DP, csp);
  };
};
```

The Consumer process shown in Listing 6.18 again has an instance of *CSPnode* and *CSPport*. The Consumer accepts a character from the channel and prints it out. The constructor is straightforward where *SC_CSP_THREAD()* macro registers the *CONSUMER* class as a CSP process.

The driver program for this model is presented in Listing 6.19. The channel that connects the Producer and Consumer is *ptoc*. This channel is bound with the processes' respective ports. The direction of the channel is set by using the *points_to(...)* member function from the *CSPnode* class. *runcsp(...)* prepares the CSP simulation for execution, and a global function *sc_csp_start(...)* triggers this CSP model.

Listing 6.18. *CONSUMER* module declaration

```
SC_MODULE(CONSUMER) {
  CSPnode csp; // CSP node
  CSPport<char> fromProducer;
  ProcInfo proc;

  void getChar()  {
    while (1)
    {
      char ch ;
      ch = fromProducer.get(csp);//(char*)csp.get(fromProducer.read()) ;
      cout << ”<<<<<<<<<< Received ” << ch << endl;
      csp.reschedule();
    }
  };

  SC_CTOR(CONSUMER) {
    SC_CSP_THREAD(getChar, DP, csp);
  };
};
```

Listing 6.19. Driver program for Producer/Consumer Example

```
int sc_main(int argc, char *argv[]) {
  CSPchannel<char> ptoc; // Channel from Producer to Consumer
  PRODUCER p(”Producer”); // Producer Instance
  p.toConsumer(ptoc); // Bind Producer
  p.csp.setprocname(”Producer”); //Debug information

  CONSUMER c(”Consumer”); // Consumer Instance
  c.fromProducer(ptoc); // Bind Consumer
  c.csp.setprocname(”Consumer”); //Debug information

  p.csp.points_to(c.csp, ptoc); // Set direction of channel

  DP.runcsp(DP); // Prepare CSP for execution
  sc_csp_start(””,&DP); // Start simulation
  return 0;
};
```

6. Integrating CSP & DE kernels

An understanding of QuickThreads and their implementation in SystemC as coroutine packages is required for integrating these two MoCs. We advise the reader to read Appendix A for a better understanding of QuickThreads and coroutine packages in SystemC.

Appendix A explains the workings of the reference kernel for the SystemC scheduler with focus on the coroutine packages. We briefly reiterate how SystemC manages its coroutines and thread processes. Our interest is primarily in thread processes because CSP processes are also thread processes that we want to schedule differently and separate from the DE kernel. The SystemC scheduler initializes thread processes by creating stack space along with initializing the stack with the appropriate function and its arguments. After thread initialization, the threads are executed by invoking the *yield(...)* function from the *sc_cor_pkg* that switches out the current executing process and prepares the new process (passed via the argument of the function) to execute. Suspension functions such as *wait(...)* perform this switch to allow other runnable processes to execute. The QuickThread package uses *preswitch* for context switching that allows for this implementation. A function called *next_cor(...)* is used to determine the next thread to execute. Once the runnable queues are empty, the control is returned to the main coroutine identified by the *main_cor* coroutine. This main coroutine can also be suspended, which is what happens when a new thread process is scheduled for execution. It is also resumed after no more thread processes are *runnable.*

Listing 6.20. Overloaded Constructor and helper function in *sc_cor_pkg_qt*

```
sc_cor_pkg_qt::sc_cor_pkg_qt( CSPnodelist* simc )
: sc_cor_pkg( simc )
{
  if( ++ instance_count == 1 ) {
    // initialize the current coroutine
    assert( curr_cor == 0 );
    curr_cor = &main_cor;

}

sc_cor*
sc_cor_pkg_qt::get_demain()
{
    return curr_cor;
}
```

Different semantics for Discrete-Event based simulation and CSP simulation justifies the need for separation of these two kernels. However, SystemC reference implementation treats the *sc_simcontext* class as the toplevel scheduler class with the main coroutine and coroutine package accessible only through an instance of *sc_simcontext.* For isolation, we included functionality in the CSP encapsulation to have pointers to the coroutine package and the main coroutine. We also implemented a CSP-specific *next_cor()* function along with several other thread core functions discussed earlier. The CSP kernel as a stand-alone kernel works without any concerns. However, we encounter an interesting problem when invoking the DE kernel to execute a DE model. As we know, SystemC is designed as a single scheduler simulation framework, which means the coroutine package is created from the *sc_simcontext* class in the *initialize(...)* function. When trying to invoke *initialize(...)* while in a CSP simulation, the loss of process stack space is experienced. This is due to a singleton pattern used in creating SystemC's DE scheduler. Hence, only one instance of the coroutine package must exist and given that we attempt to invoke the DE kernel from within the CSP kernel, the DE kernel must address the coroutine package created in the CSP kernel instance. This requires a couple of changes in the the coroutine package files and the *sc_simcontext* class. We first discuss the changes we made in the coroutine packages.

Listing 6.21. Overloaded Constructor in *sc_cor_pkg* class

```
class sc_cor_pkg
{
public:
    ...
    // overloaded constructor
    sc_cor_pkg( CSPnodelist* simc )
        : m_simcsp( simc ) { assert( simc != 0 ); }
    ...

    void setsimc(sc_simcontext * simc) { m_simc = simc; };
    void set_csp(CSPnodelist * csp) { m_simcsp = csp; };

    // get the simulation context
    sc_simcontext* simcontext()
        { return m_simc; }
    CSPnodelist * cspcontext()
        { return m_simcsp;}
private:

    sc_simcontext* m_simc;
    CSPnodelist * m_simcsp;
private:
    ...
};
```

Creating an instance of type *sc_cor_pkg_qt* makes a check for having one instance with the *instance_count* and its interface class constructor is also invoked. An object of *sc_cor_pkg_qt* results in the constructor of *sc_cor_pkg* being invoked. Hence, the overloaded constructor described in Listing 6.20 invokes the constructor of class *sc_cor_pkg* with an argument containing the *CSPnodelist* pointer. A helper function *get_demain()* is added to retrieve the *curr_cor* that signifies the current executing context. We use this to make a call-back to the process that performs the invocation of the DE kernel. The interface also undergoes modification to accommodate calls to the interface to extract the correct information. Listing 6.21 displays the additions to the *sc_cor_pkg* class.

A pointer to the *CSPnodelist* is added as a private variable and its respective member functions to set and get address of this pointer. These are the changes that have to be done in the coroutine packages to allow for a CSP model to execute using the coroutine package. At this point we

Listing 6.22. *next_cor()* member function in class *sc_simcontext*

```
sc_cor* sc_simcontext::next_cor()
{
    if( m_error ) {
      return m_cor;
    }
    sc_thread_handle thread_h = pop_runnable_thread();
    while( thread_h != 0 && ! thread_h->ready_to_run() ) {
      thread_h = pop_runnable_thread();
    }
    if( thread_h != 0 ) {
      return thread_h->m_cor;
    } else {
        return (oldcontext);
    }
}
```

only show the inclusion of one *CSPnodelist* (one CSP model) addressed by the coroutine packages. However, we plan to extend this later to support multiple CSP models using the same coroutine package.

We are considering invocations of the DE kernel through the CSP kernel, which requires altering the initialization code for the *sc_simcontext* class. We need to point the *m_cor_pkg* private member of class *sc_simcontext* to the *sc_cor_pkg* pointer in the *CSPnodelist* class. This is performed by invoking the *cor_pkg()* from the *CSPnodelist* followed by an invocation of *get_main()* to retrieve the main coroutine. We introduce a new private data member in *sc_simcontext* called *oldcontext* of type *sc_cor**, which we set by invoking the *get_demain()* member function on variable *m_cor_pkg*. We use *oldcontext* during the *next_cor()* function for class *sc_simcontext* as shown in Listing 6.22.

Variable *oldcontext* is returned when there are no more runnable threads in the system, similar to the original implementation of the *next_cor()* function where *main_cor* was being returned. The purpose of saving *oldcontext* is to allow the simulation to return to the coroutine that invoked the DE kernel. Suppose a CSP process invokes a DE kernel for some computation. *oldcontext* would then store the coroutine of the calling CSP process. The DE simulation returns to *oldcontext* once it has no more processes for execution, resuming the execution of the calling CSP process.

We illustrate the invocation of the DE kernel from the CSP in Figure 6.8. The assigned addresses are made up and do not resemble real addresses in our simulation, but we merely present them to further clarify the manner in which the *oldcontext* is used. During initialization of the CSP model, shown by the CSP block, *m_cor_pkg* and *main_cor* are set to their correct addresses. Every thread process has an *m_cor* variable

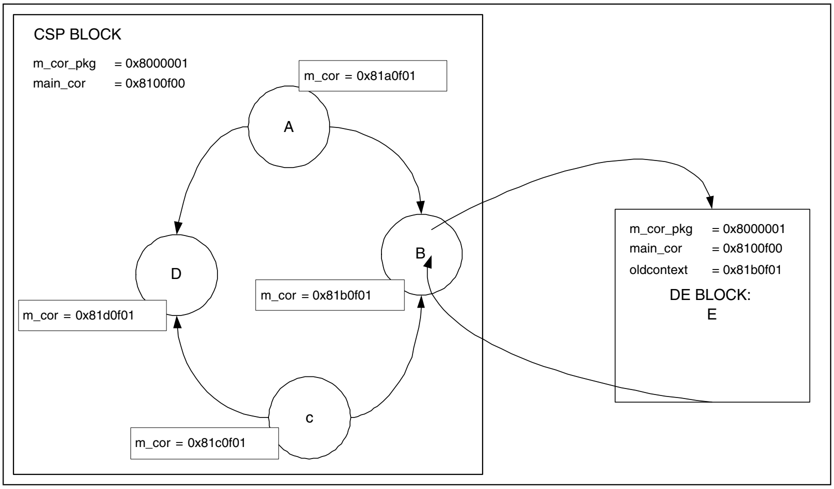

Figure 6.8. Example of DE kernel invocation in CSP

that holds the coroutine for that particular thread. At some point during the execution of process B, a DE model is supposed to execute. This DE model requires that the CSP kernel yield to the DE kernel to simulate the DE block. Hence, the initialization functions of the DE kernel are called where the addresses of the private data members *m_cor_pkg* and *main_cor* are extracted from the CSP kernel and the current simulation context is saved in *oldcontext*. Notice that the address of the *oldcontext* is the same as the *m_cor* value of process B. According to the *next_cor(...)* function definition in Listing 6.22, *oldcontext* is returned once there are no more threads to execute, implying that once the DE simulation model is complete and there are events to be updated, the scheduler returns control to *oldcontext* which is the calling CSP thread. This in effect allows for DE kernel invocations from CSP as we show via an implemented example in Chapter 9.