Chapter 9

HETEROGENEOUS EXAMPLES

1. Model using SDF kernel

We construct a heterogeneous example for an image converter shown in Figure 9.1. This system begins by downloading encrypted images that are decrypted, converted to a specific type, then encrypted again and uploaded back to the source. This is a multi-MoC model where the first DE block (Decryption block) is responsible for downloading encrypted images and decrypting them. These images are passed onto the SDF block that performs the Sobel edge detection algorithm on the image. This is our preferred conversion type. Output from Sobel is sent to the final DE block (Encryption block) that encrypts the converted image and uploads it to a particular location. All DE blocks consist of *SC_CTHREAD()* processes and the SDF block uses *SC_METHOD()* processes.

We create this model using three scenarios: a pure DE implementation, a DE implementation such that the processes are non-threaded using the transformation technique in [62] and a heterogeneous implementation using the DE and SDF kernels. The Discrete-Event model of the Converter uses control signals to indicate which process executes next and so on. The non-threaded model converted every *SC_THREAD()* or *SC_CTHREAD()* process to an *SC_METHOD()* using the transformation technique in [62]. The DE-SDF model is shown in Figure 9.1.

An interesting aspect about this model is the interaction between the DE and SDF blocks. By interaction we mean the data transfer. SystemC channels may be employed as the interaction medium between the DE and SDF blocks, but we advise using *SDFport*s and *SDFchannel*s to push the data onto the SDF toplevel. Both the blocks responsible for

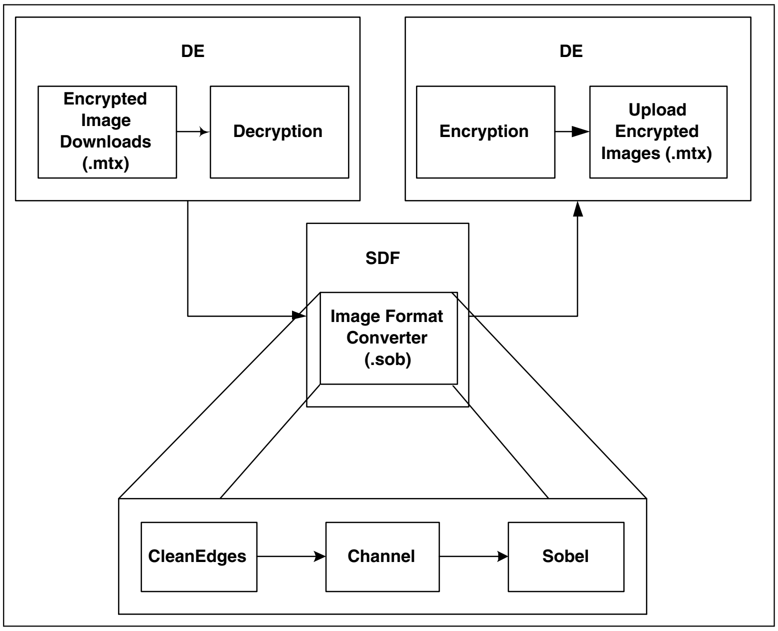
DE
Encrypted Image Downloads (.mtx)
Decryption
DE
Encryption
Upload Encrypted Images (.mtx)
SDF
Image Format Converter (.sob)
CleanEdges
Channel
Sobel

Figure 9.1. Image Converter Diagram

pushing data into the SDF and the SDF itself, must have access to the *SDFchannel* and since MoC-specific channels and ports do not generate SystemC events, this can be done. The block pushing data into the SDF model can also have an *SDFport* that binds to an *SDFchannel* that the SDF component retrieves its inputs from. A control signal can be used to trigger the SDF toplevel process once data is ready on the channel for the SDF to consume. This is a simple example showing how we implement a heterogeneous model (the image Converter) using our improved modeling and simulation framework.

Figure 9.2 shows approximately 13% improvement over the original model. We attribute the limitation in simulation efficiency increase to Amdahl's law [41]. The SDF block in this Converter model serves only a small portion of the entire system allowing for only that much improvement in total simulation performance. If the SDF component was responsible for more percentage of the original model, then the simulation efficiency of that model would be significantly larger than its counterpart (created using the DE kernel).

We profiled the Converter model. Taking an approximate percentage of time spent for the SDF model when using the original reference implementation kernel, we can see from Table 9.1 that the approximated total running time for the SDF is approximately 14%. This particular

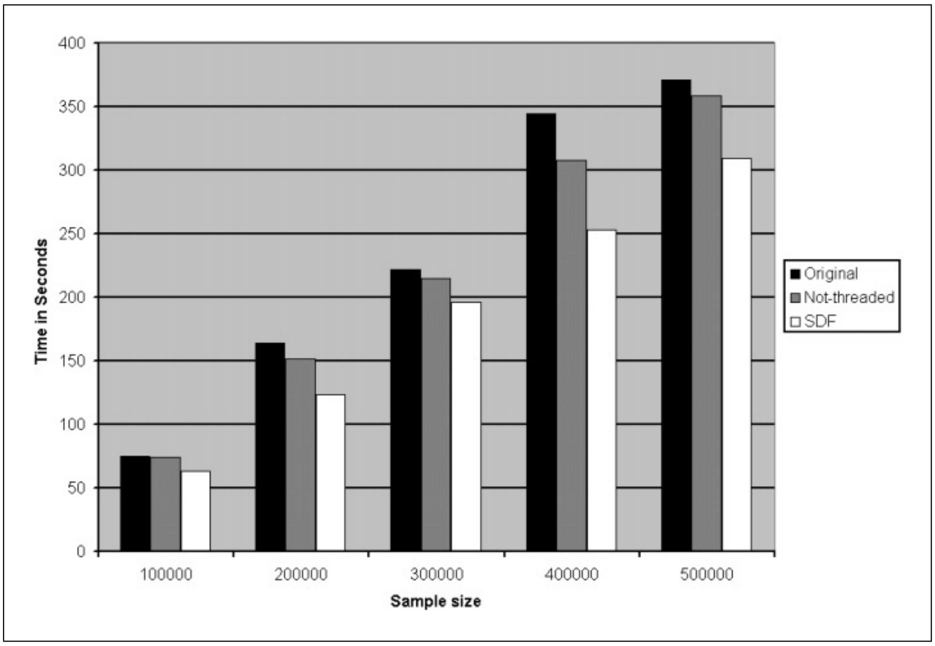

Figure 9.2. Converter Results

implementation of the Converter model has *sc_fifo* channels that generate numerous events during passing of data. Now, using Amdahl's law from Equation 9.1 and using 14% as the fraction of enhanced model we can follow Equation 9.1 [41] where s is the speedup of the enhanced portion. This equation implies that, if we were to enhance the simulation of the SDF components by 2x, then we can only achieve a simulation performance less than $\frac{0.14}{2} = 7\%$. Therefore, if our SDF components was either larger or if we have more SDF components in a model then the simulation performance will improve according to the percentage of SDF component in the model.

$$Overall\ Speedup = \frac{1}{((1 - .14) + .14/s)} \tag{9.1}$$

Table 9.1. Profiling Results for Converter Model

Function	% of total running time spent
cleanedges::cleanedge(void)	6.78
sobel::operate(void)	5.66
channel::latch(void)	1.18

2. Model using CSP and FSM kernels

We reuse the Dining Philosopher example introduced in Chapter 6 to construct a heterogeneous model using the CSP and FSM kernels. The model previously created was a pure CSP model, where every philosopher and fork was a CSP process. We mentioned that a superficial implementation of the Dining Philosopher problem can result in deadlock, to which we presented the idea of a footman. The role of the footman is to seat the philosophers to their designated seats and to monitor that only four philosophers are sitting at the dining table at any time. The implementation of the footman was done via a global function to have immediate verification of the state of the seats occupied at the dining table. However, we take this example further by implementing the footman as a Finite State Machine controller embedded in a CSP process.

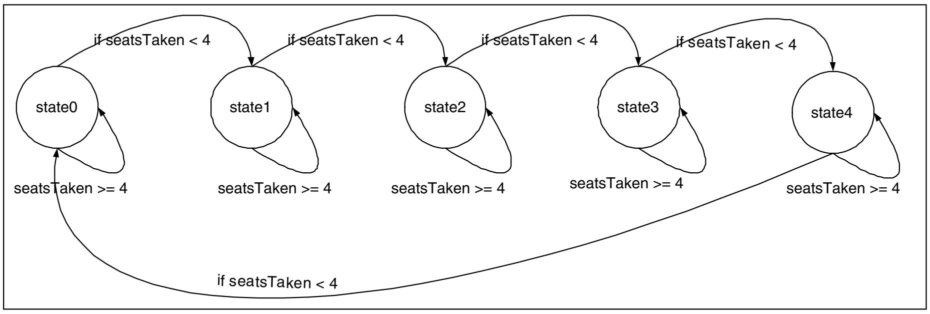

Figure 9.3. FSM implementation of the Footman

The FSM based footman allows the seating of four philosophers, where by each seat allocation signifies a state. A global seat counter called *seatsTaken* is updated every time a philosopher is assigned a seat and when a philosopher leaves his seat. The FSM in Figure 9.3 is combined with the Dining Philosopher implementation to yield a heterogeneous example shown in Figure 9.4.

Figure 9.3 presents a state machine diagram showing the functions of the footman. The initial state is state0. The functionality of every state is the same except for the next state transitions. Every state has a self-loop suggesting that the control in the FSM does not transition to another state whenever four philosophers are seated. However, if there are seats available, then a seat is allocated and the transition to the next state occurs. This is a simple FSM that changes the solution of the Dining Philosopher such that it ensures that every philosopher gets a turn to eat as well as serving as a deadlock avoidance mechanism. The module definition is shown in Listing 9.1 where object *s* is the

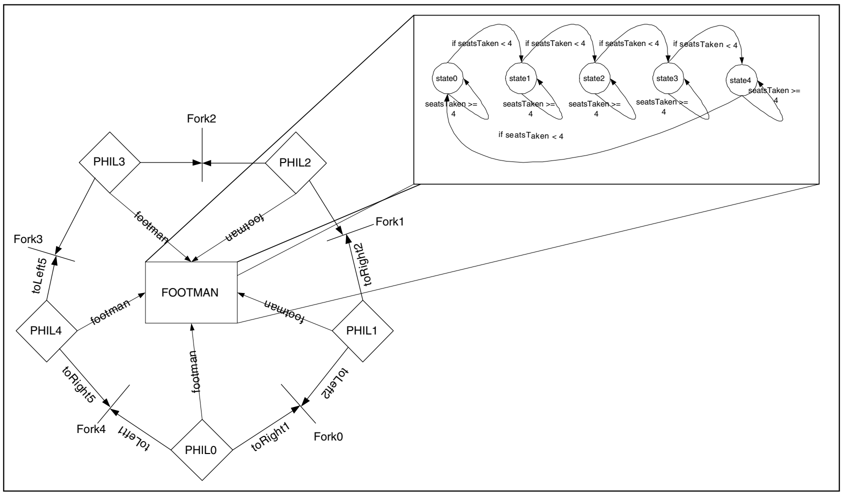

Figure 9.4. Dining Philosopher Model with FSM footman

Listing 9.1. Module definition Footman FSM

```
SC_MODULE(s) {

  int random;
  int * giveSeat;

  CSPnode * csp;
  CSPport<int> * fromPhil0;
  CSPport<int> * fromPhil1;
  CSPport<int> * fromPhil2;
  CSPport<int> * fromPhil3;
  CSPport<int> * fromPhil4;

  void state0();
  void state1();
  void state2();
  void state3();
  void state4();

  SC_CTOR(s) {
    giveSeat = new int();
    *giveSeat =1;
    fsm_model->setState("toplevel.state.state0");
    SC_FSM_METHOD(state0, fsm_model);
    SC_FSM_METHOD(state1, fsm_model);
    SC_FSM_METHOD(state2, fsm_model);
    SC_FSM_METHOD(state3, fsm_model);
    SC_FSM_METHOD(state4, fsm_model);
  };
};
```

state machine. This module itself has pointers to *CSPnode* and *CSPport* types. The variables with prefix *fromPhil* are the ports through which the philosophers communicate with the footman requiring the footman to be encapsulated in a CSP process. Therefore, the FSM defining the behavior of the footman is embedded in a CSP process through which the *CSPnode* object and *CSPchannel*s are tunneled. The rendez-vous communication occurs from the toplevel process encapsulating the FSM controller. Pointers to objects of type *CSPchannel* and *CSPnode* are necessary for the FSM to have access to the *CSPchannel*s that it must communicate with and make function calls on the *CSPnode* object.

The implementation of the state entry functions are shown in Listing 9.2. Every state has identical implementation except for the next state transition.

Listing 9.2. State entry functions for Footman FSM

```
void s::state0()  {
  if (seatsTaken < 4) {
    ++seatsTaken;
    seatAvailable[0] = true;
    fromPhil0->push(*giveSeat, *csp);
    fsm_model->setState("toplevel.state.state1");
  }
};

void s::state1()  {
  if (seatsTaken < 4) {
    ++seatsTaken;
    seatAvailable[1] = true;
    fromPhil1->push(*giveSeat, *csp);
    fsm_model->setState("toplevel.state.state2");
  }
};

void s::state2()  {
  if (seatsTaken < 4) {
    ++seatsTaken;
    seatAvailable[2] = true;
    fromPhil2->push(*giveSeat, *csp);
    fsm_model->setState("toplevel.state.state3");
  }
};

void s::state3()  {
  if (seatsTaken < 4) {
    ++seatsTaken;
    seatAvailable[3] = true;
    fromPhil3->push(*giveSeat, *csp);
    fsm_model->setState("toplevel.state.state4");
  }
};

void s::state4()  {
  if (seatsTaken < numPhil - 1)  {
    ++seatsTaken;
    seatAvailable[4] = true;
    fromPhil4->push(*giveSeat, *csp);
    fsm_model->setState("toplevel.state.state0");
    }
  };
```

The *seatAvailable* array maintains which seat has been occupied and a record of every philosopher to his particular seat is kept by the index of the array. For example, *seatAvailable[1]* refers to the seat that belongs to a philosopher with *id* one.

Listing 9.3. Toplevel CSP process for Footman

```
SC_MODULE(fsmtop) {

  s * s1;

  CSPnode csp;
  CSPport<int> fromPhil0;
  CSPport<int> fromPhil1;
  CSPport<int> fromPhil2;
  CSPport<int> fromPhil3;
  CSPport<int> fromPhil4;

  void entry();

  SC_CTOR(fsmtop) {
    s1 = new s("state");
    s1->csp = &csp;
    s1->fromPhil0 = &fromPhil0;
    s1->fromPhil1 = &fromPhil1;
    s1->fromPhil2 = &fromPhil2;
    s1->fromPhil3 = &fromPhil3;
    s1->fromPhil4 = &fromPhil4;

    SC_CSP_THREAD(entry, DP, csp);
  };
};
```

The toplevel CSP process contained in *fsmtop* module is defined in Listing 9.3. This module definition has instances of *CSPnode* and *CSPport*s that had pointer declarations in Listing 9.1. The constructor of module *fsmtop* initialize an instance of *s* and appropriately assigns the addresses of the ports in object *s1*.

The model with the addition of the footman CSP process is shown in Figure 9.4. This pictorial representation of the Dining Philosopher also shows *CSPchannel*s from every philosopher to the footman. This addition requires alteration to the main entry function for the philosopher. This change requests a seat from the footman before proceeding with the *getfork()* function to request forks. The only addition is calling *footman.get(...)* such that the philosopher suspends itself until the footman process is executed and a seat is allocated. The FSM controller manages the seat allocation and returns a value on the channel designating a seat, resuming the suspended philosopher process.

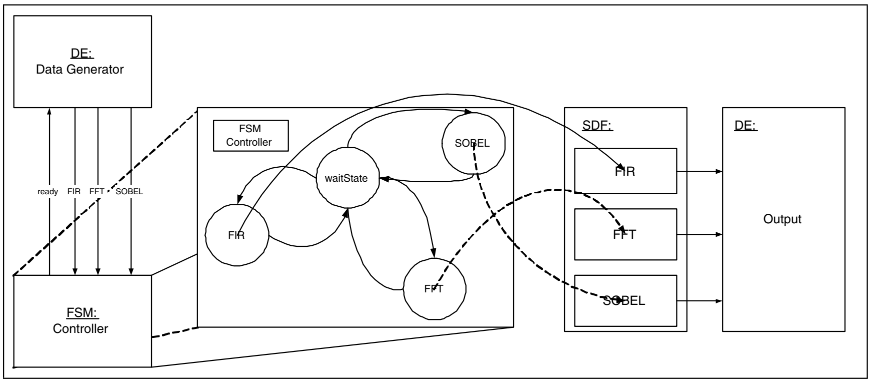

Figure 9.5. Heterogeneous Example using FSM, SDF and DE Models

3. Model using FSM, SDF and DE kernels

Another heterogeneous example is shown in Figure 9.5. This model is separated into DE, FSM, SDF and DE blocks as shown in the diagram. The DE models are the data generator and the output, the SDF models are the FIR, FFT and Sobel, and the FSM model is the controller responsible for triggering the SDF computations. The first DE block is the data generator, which at random selects one of the three SDF models for execution. It also uses the SystemC Verification library (SCV) [49] for generating randomized input data for the FIR and FFT. Input for the Sobel is read from a file. The data generator block sends a signal to the FSM controller to execute the chosen SDF computation.

The FSM model consists of four states. The initial state is the *waitState* and the other three states fire the FIR, FFT, and Sobel SDF models respectively. The *waitState* receives a signal from the data generator block and according to the signal, transition to the respective state is taken. For example, suppose the data generator block sends a signal to execute the FIR model, then the FSM takes a transition from the *waitState* to the *FIR* state. The SDF model of the FIR executes when in this state. Upon complete execution of the SDF FIR model, control returns back to the *FIR* state that takes a transition back to the *waitState* state. In turn, the FSM controller sends a signal back to the data generator that the controller is ready to receive another input.

This example illustrates the use of SCV with a heterogeneous MoC employing MoC-specific kernels. Invocations of an FSM model, DE model and SDF models are also presented in this example. However, due to the lack of space the source code is made available at [36].

4. Model using CSP, FSM, SDF and DE kernels

Building on top of the example with an FSM based footman for the Dining Philosophers problem, we construct another model that uses the SDF and DE kernels along with the CSP and FSM. This example shows the use of all MoCs implemented in SystemC until now. In the model presented in Figure 9.6, we use the Sobel edge detection model as our SDF component, the RSA encryption algorithm and the Producer/Consumer FIFO example as the DE components (from SystemC distribution examples) and the footman as the FSM controller.

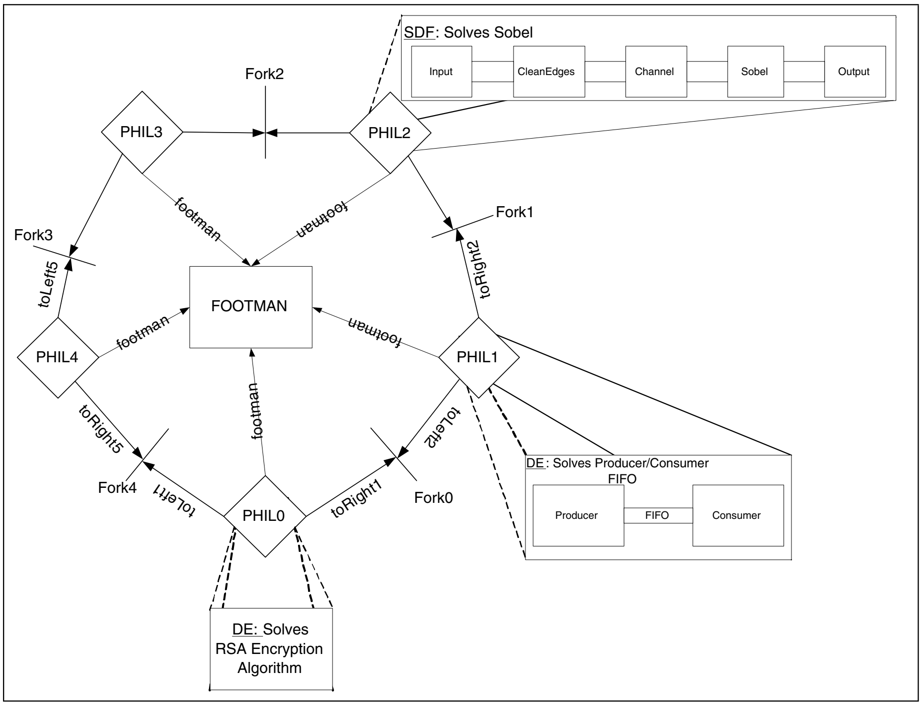

Figure 9.6. Truly Heterogeneous Dining Philosopher Model

The footman FSM controller is encapsulated in a CSP process but the DE and SDF models are not. The roles of the philosophers when in the thinking state have been slightly altered with this implementation and they are as follows: Phil0 solves an RSA encryption algorithm, Phil1 simulates the Producer/Consumer example using the FIFO implementation, Phil2 performs the Sobel edge detection, and Phil3 and Phil4 sit around just pretending that they are thinking. Listing 9.4 shows the *PHIL* module entry function with the added invocations to the DE and SDF kernels.

Listing 9.4. Entry function for PHIL module

```
void PHIL::soln() {
 int duration = _timeToLive;
 int eatCount = 0;
 int totalHungryTime = 0;
 int becameHungryTime;
 int startTime = msecond();
 while ( 1  ) {
   seatAvailable[id] = false;
   int got = footman.push(csp);
   if ((seatAvailable[id] == true) && ((state[id] != 0) || (
        state[id] != 6)) )  {
     becameHungryTime = msecond();
     print_states();
     getfork();
     print_states();
     totalHungryTime += ( msecond() - becameHungryTime );
     eatCount++;
     state[id] = 3;
     usleep( 1000L * random_int( MeanEatTime ) );
     print_states();
     dropfork();
     usleep( 1000L * random_int( MeanThinkTime ) );
     print_states();
     state[id] = 6;
     if (id == 1 )  {
       rsa(-1);
       sc_de_init("rsa");
       sc_de_trigger(-1, "rsa");
     }
     if (id == 2 )  {
       top top1("Top1");
       sc_de_init("Top1");
       sc_de_trigger(-1, "Top1");
     }
     if (id == 3) {
       toplevel sdftop("sdftop");
       sc_de_init("sdftop");
       sc_sdf_start(1,"sdftop");
     };
     usleep( 1000L * random_int( MeanThinkTime ) );
     state[id] = 0;
     print_states();
     --space;
     --seatsTaken;
     csp.reschedule();

   }  else  {
       csp.reschedule();
   }
 }
 state[id] = 7;
 };
```

We compare *id* to allow for a specific philosopher to perform certain tasks during their thinking phase. We mandate that every model must be encapsulated with a toplevel process. We use *SC_METHOD()* processes since we want single execution of the model. Our SDF kernel is an untimed MoC implementation, however DE examples can span for a certain length of time. To support this we added global functions *sc_de_trigger(...)* and *sc_sdf_trigger(...)*. Both these functions take in the duration of the execution and the name of the toplevel process that is to be executed. For the SDF, the duration has no effect. For the DE, the simulation runs for the specified duration after which control returns to the calling thread. The *sc_de_init(...)* initializes the model specified by the argument passed into the function to be executed. This same initializer function is used to insert the SDF model.