SystemJ Programming Manual

Avinash Malik[1], Heejong Park and Zoran Salcic
University of Auckland
Department of Electrical and Computer Engineering

February 2, 2017

[1]Author names are in Alphabetical order

Contents

Preface

SystemJ is a new system level design language, which targets highly concurrent and distributed systems requiring both complex control and data-driven processing. The SystemJ language extends the Java language with synchronous and asynchronous concurrency together with constructs for programming reactive systems. SystemJ is based on rigorous mathematical semantics and hence is amenable to formal verification. This document is a manual for programming in SystemJ. It assumes that the reader is familiar with the underlying theoretical concepts of concurrency and reactivity and is positioned as a reference manual for learning the SystemJ development environment. The reader is urged to refer the various public publications [1, 2, 3, 4, 5] underlying SystemJ's model of computation before going through this documentation and developing SystemJ models and programs. Besides showing how SystemJ programs are developed, the manual also illustrates an important feature of SystemJ, namely the ability to write executable test-benches as concurrent programs. These speed-up the development process and assist in faster verification of developed programs and systems. We also highlight how SystemJ may be used for effective coding of critical sections as separate asynchronous threads called clock-domains (see the PCABRO example in 2).

SystemJ targets different execution platforms. Examples of platforms are servers and desktop computers that require a Java Virtual Machine (JVM), embedded microcomputers that require at least a smaller version of JVM (J2ME) and Google's Android Platform, special embedded processors and microprocessors on chip customised for SystemJ execution. Even pure hardware platforms can execute SystemJ with or without an operating system, depending on level of performance and system resources that are required to execute final application.

Chapter 1

Introduction to SystemJ

In this chapter we will introduce the tools needed in developing SystemJ programs. We need to explain some SystemJ specific terminology which will be used in rest of this manual.

1.1 What is SystemJ

Design of highly concurrent and distributed systems has been a major software engineering challenge due to the need for using multi-threaded programming. A paper by Lee [6] highlights the problems associated with *threads*: "They discard the most essential and appealing properties of sequential computation: understandability, predictability , and determinism". Understandability is lost since the programmer has the responsibility of ensuring correctness through complex synchronisation mechanisms provided by the RTOS. Predictability is sacrificed since concurrency is emulated through RTOS scheduling that is inherently non-deterministic.

SystemJ is a system-level design language [2, 7, 3] for demystifying the programming of highly concurrent and possibly distributed systems. It combines the synchronous elegance of the Esterel language [8, 9], with the asynchronous concurrency of CSP [10] and, the object-oriented data encapsulation capability of Java. This enables the programming of highly concurrent and distributed systems with ease. The language is created by extending the Java language with a few syntactic extensions to enable very high-level modelling of a system under development. Also, a compositional semantic [3] is proposed to enable effective compilation and formal analysis. SystemJ is an ideal language for the design of Globally Asynchronous and Locally Synchronous (GALS) systems[3].

A SystemJ program (also known as a *system*) consists of one or more

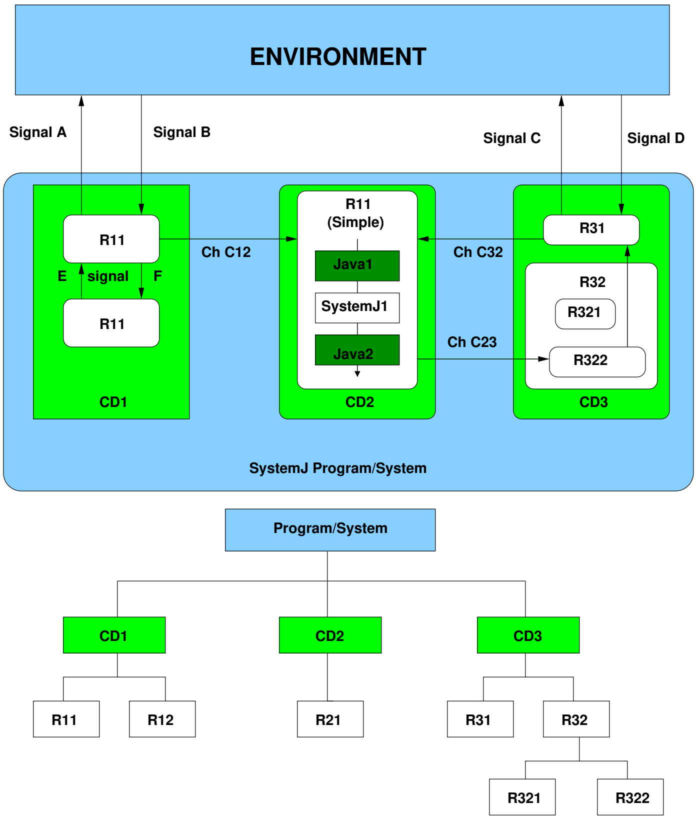

Figure 1.1: A typical SystemJ program's graphical illustration

clock-domains. Clock-domains execute asynchronously (each at its own logical speed) and sometimes communication with each other using message passing. Communication between clock-domains is facilitated using entities called *channels*, which allow point-to-point communication. A clock-domain, in turn, may consist of multiple *reactions* all of which are synchronized with respect to a logical clock. The reactions move in lock-step synchronously with respect to a logical clock. Thus, synchrony within a clock-domain is achieved by ensuring that all reactions progress using a single logical clock. Asynchrony among clock-domains is achieved using a different logical clock for each clock-domain. Communication between the reactions is facilitated using entities called *signals*, which are broadcasted across the reactions of a given clock-domain. These concurrency constructs facilitate the high-level modelling of the reactive control aspects while all data computations are managed using Java classes.

A graphical illustration of a typical SystemJ program is given in Figure 1.1. This example program has three clock-domains named CD1, CD2 and CD3 respectively. The clock-domain CD1 has two synchronous parallel reactions R11 and R12, while the clock-domain CD2 has a single reaction R21 and, the clock-domain CD3 has two reactions R31 and R32. SystemJ allows structural hierarchy and hence the reaction R32 also has two children reactions R321 and R322 respectively. The structural hierarchy of reactions may be presented as hierarchical tree shown on lower part of Figure 1.1.

Chapter 2

SystemJ by Example

In this chapter we introduce the SystemJ programming style through three different examples. First the ABRO example by Berry [8] is adapted for motivating the synchronous reactive (SR) programming style [11] of a single clock domain. We then present the AABRO example to illustrate how two asynchronous clock domains execute. Finally, we present the PCABRO example, where we illustrate how Java code can be incorporated within SystemJ code to facilitate the creation of complex multi-threaded programs that share data without the need for complex critical sections.

2.1 SR Programming using ABRO

eactive systems continuously interact with their environment at a speed determined by the environment. Synchronous reactive (SR) programming style [11] was introduced in the early 80's to facilitate the modelling of these systems without the need for the use of an operating system to capture the inherent concurrency and reactivity. A key distinguishing feature of this modelling style is that all *correct* synchronous programs are guaranteed to be *deterministic* and *reactive*. Informally, determinism implies that a system produces the same output trace in response to a given input trace. Reactivity implies that the system remains responsive to valid external stimulus. These key properties are the corner stone of the SR programming paradigm.

A key feature of the SR paradigm is that programs are inherently concurrent and all concurrent threads progress in lockstep relative to the ticks of a logical, global clock. The *synchrony hypothesis*, based on which all synchronous languages operate, states that inputs and the corresponding outputs have identical time stamp i.e, when an input happens the corresponding output is generated instantaneously. This hypothesis holds when the idealized reactive system executes infinitely fast compared to its environment. When a synchronous program is implemented on a physical device, in order to ensure that the synchrony hypothesis is respected, we have to ensure that inputs arrive at a rate that is slower than the processing time of the computation of any given reaction. Reaction, in this context refers to the amount of computation a system must complete within one tick (this is not to be confused with the concept of a *reaction* in SystemJ, which essentially represents a synchronous thread).

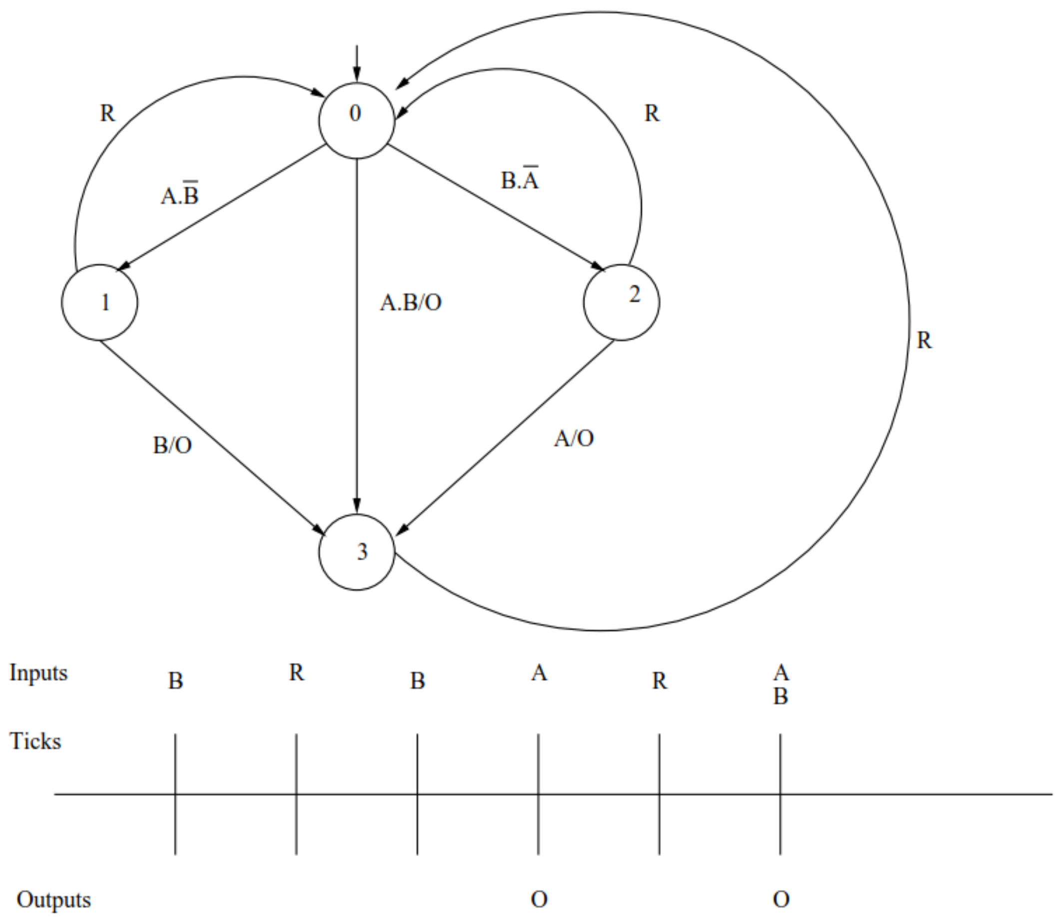

Figure 2.1: ABRO as a FSM and one behaviour trace

We illustrate this programming style using the ABRO program as shown in Listing 2.1. ABRO is like the *hello world* of SR programming and was first introduced by Berry [8]. This reactive program waits for the presence of the *signals* A and B in the environment. When both of them have happened, the program emits an O to the environment. This behaviour is reset and restarted every time the input R has happened. Otherwise, the program just waits for R to happen. A finite state machine (FSM) capturing this reactive behaviour is shown in Figure 2.1 and a sample trace is also shown in the same figure below the FSM. In state 0, the machine waits for inputs. If only A happens then a transition to state 1 is made. After this, it waits for B or R to happen. If R happens the behaviour resets to state 0 again. If, on the other hand, B happens then the output O is emitted and a transition to state 3 is made. Here the FSM waits until R happens to reset the behaviour to state 0. Similarly, from state 0 the other possibility would be for B to happen followed by A where O is emitted and control reaches state 3 from 2. The final possibility is that both A and B happen together and hence a transition is made directly from state 0 to state 3 with the emission of O.

Listing 2.1: ABRO in SystemJ

```
1  ABROCD(
2    input signal A, B, R;
3    output signal O;
4  )->{
5    while(true){
6      abort(R){
7        {await(A);}{await(B);}
8        emit O;
9        while(true){
10         pause;
11       }
12     }
13   }
14 }
```

The SystemJ code for this application is shown in the Listing 2.1. Since the program is reactive, you start by defining the *interface* of the program (lines 2 to 3) where all input and output channels and signals are defined. While channels are objects using which point-to-point communication between clock domains is established, signals are used for synchronous communication between reactions. In the ABRO program, we only have input signals A, B, R which are received from the environment and output signal O that is emitted to the environment. The ABRO program has a single clock domain enclosed between braces on line 4 to 14 respectively.

ABRO behaviour being reactive starts with an infinite loop on line 5. At the start of this behaviour is a preemption construct called *abort* (on line 6) which has a body marked between lines 7 to 11. The condition for taking this preemption is the signal R. This preemption statement will kill the body whenever the R input is present (true) in the environment, except in the first instance of execution. First instance of execution is defined to be the instant when control reaches the body. Also, note that preemption of the body will happen automatically in the instance control passes the last statement of the body (line 12).

The first statement of the abort body at line 7 is the synchronous parallel (`||`) composition of two await statements that wait for the inputs A and B respectively. The `||` operator executes two synchronous threads in parallel every tick until both branches have terminated. Only then the `||` terminates. In this example, the `||` will terminate only when both A and B have happened in the environment within one tick (transition 0 to 3 in the ABRO FSM in Figure 2.1) or have happened one followed by the other in different ticks (either transitions 0 to 1 followed by 1 to 3, or 0 to 2 followed by 2 to 3 in the ABRO FSM in Figure 2.1). Only after the `||` terminates, the program will emit the output O (line 8).

Note that ABRO behaviour demands that only after the occurrence of R the ABRO behaviour can restart from line 5. However, the abort will be taken (even when R is not present) when the body is finished (after the emission of O). To prevent this, we have an infinite delay loop to halt control within the body until R happens. Here *pause* (line 10) is a delay statement that delays for one tick. By enclosing the pause within an infinite loop, we effective have a halt. The behaviour of the ABRO will be restarted as soon as R happens. This will resume the abort body thus starting the two concurrent await statements.

The ABRO example highlights several features of SystemJ and in particular the features it inherits from the SR style that is a subset of the language. It highlights *synchronous preemption* using the *abort*, concurrency using the `||`, sequencing by ensuring that the emission O follows the occurrence of A and B and, signal emission by the *emit* statement and, delay using the *pause* and *await* statements. More details on these synchronous constructs are described in Chapter 3. In the next section we will illustrate the asynchronous composition of SystemJ using the AABRO example.

2.2 GALS modelling in AABRO

The Asynchronous ABRO (AABRO) example shown in Listing 2.2 highlights the GALS modelling style of SystemJ. In SystemJ reactions could be either named or unnamed. In the ABRO program, we had a single unnamed reactions. In this program, we have a named reaction called abro defined on line 1. Named reactions may be parametrized by passing different arguments. In the current example, we have effectively created two clock domains by passing signals A1, B1, R1 and O1 to the first reaction and A2, B2, R2 and O2 to the second reaction respectively. Then by instantiating these reactions in two different clock-domains, we have created a simple GALS program. While the first ABRO clock domain will emit a O1 whenever inputs A1 and B1 have happened in the environment, the second one will emit O2 in response to A2 and B2.

Listing 2.2: AABRO in SystemJ

```
reaction abro(: input signal A, input signal B, input signal R, output
signal O){
  while(true){
    abort(R){
      {await(A);}||{await(B);}
      emit O;
      while(true){
        pause;
      }
    }
  }
}

ABROCD1(
  input signal A1, B1, R1;
  output signal O1;)->{
    abro(:A1,B1,R1,O1) || { /* empty reaction */ }
}

ABROCD2(
  input signal A2, B2, R2;
  output signal O2;)->{
    abro(:A2,B2,R2,O1) || { /* empty reaction */ }
}
```

A sample trace of the behaviour of the AABRO program is shown in the Figure 2.2. In this program, the two clock domains are completely independent. Also, both the examples presented in the previous two sections illustrate pure control flow but involve no data. In the next section we present an example that involves the synchronization of clock domains while also using the data encapsulation capability of Java to develop an interesting producer consumer application that is thread-safe by construction.

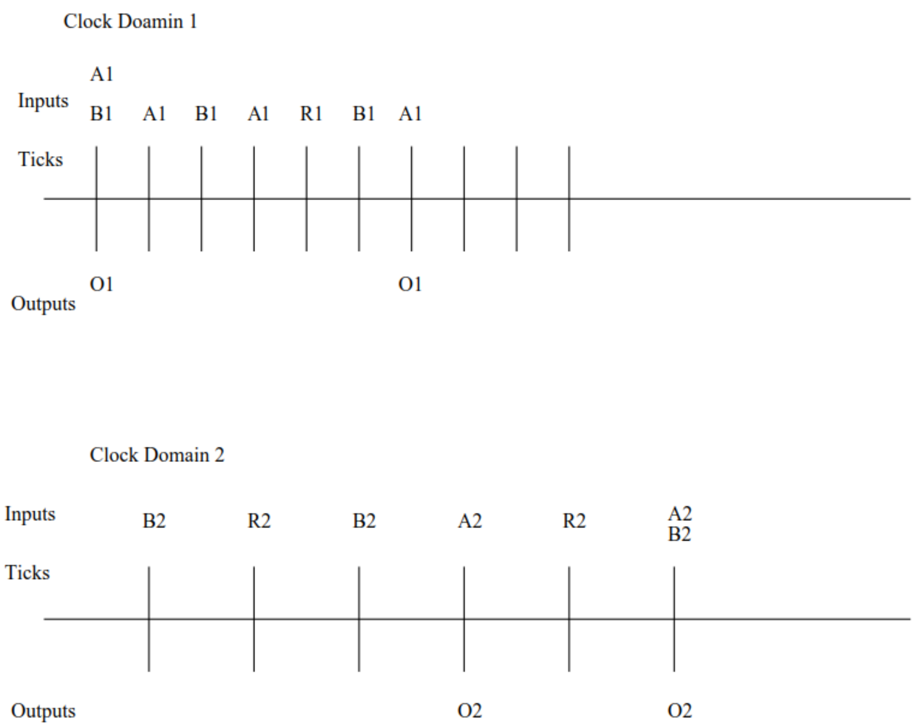

Figure 2.2: Sample behaviour trace of AABRO

2.3 PCABRO: GALS program with synchronization

Listing 2.3 consists of three clock domains. The first two PABRO and CABRO are the ABRO clock domains enhanced to act as producer and consumer, respectively. These two clock domains run concurrently with a third clock domain, BUFFER, which acts as the intermediary between the two. The BUFFER clock domain implements a circular buffer. The PABRO clock domain produces a series of Fibonacci numbers and passes them onto the circular buffer using rendezvous on channel producer (lines 8- 15). The CABRO clock domain receives the data from the PABRO clock domain via the circular buffer (lines 25). The BUFFER clock domain runs three reactions concurrently the first reaction (line 41) receives the data from the producer, the second reaction (line 60) sends the data to the CABRO clock domain via channel consumer, and lastly the third reaction (lines 67- 81) maintains the circular buffer using the mix of Java and the SystemJ control constructs. Note that the clock-domains in this example are instantiated using the notation `CD(..)->R` (lines 84- 86), which indicates a body of the clock-domain `CD` is defined by a body the reaction declared as `R`.

Listing 2.3: PCABRO in SystemJ

```
import buffer.*;
import fibonacci.*;

reaction PABRO(: input signal A1, input signal B1, input signal R1,
output signal O1, output int channel producerChannel){
  FibonacciGenerator f = new FibonacciGenerator();
  while(true){
    abort(R1){
      {await (A1);} || {await (B1);}
      emit O1;
      send producerChannel(f.getNext());
      while(true){
        pause;
      }
    }
  }
}

reaction CABRO(: input signal A2, input signal B2, input signal R2,
output signal O2, input int channel consumerChannel){
  while(true){
    abort(R){
      {await (A2);} || {await (B2);}
      emit O2;
      receive consumerChannel;
      int data = (Integer)#consumerChannel;
      System.out.println("PC-ABRO Received next fibonacci number: " +
          data);
      while(true){
        pause;
      }
    }
  }
}

reaction BUFFER(: input int channel producerChannel, output int channel
    consumerChannel){
  signal bufferNotFull, bufferNotEmpty, requestData;
  Integer signal toBuffer, fromBuffer;
  {
    while(true){
      present(bufferNotFull){
        receive producerChannel;
        if(#producerChannel != null){
          int data = (Integer)#producerChannel;
          emit toBuffer(data);
          pause;
        }
      }
      pause;
    }
  }
  ||
  {
    while(true){
      present(bufferNotEmpty){
        emit requestData;
```

```
56           pause;
57           pause;
58           present(fromBuffer){
59             int data = (Integer)#fromBuffer;
60             send consumerChannel(data);
61           }
62         }
63         pause;
64       }
65     }
66     ||
67     {
68       Buffer myBuffer = new Buffer(100);
69       int data =0;
70       while(true){
71         present(toBuffer){myBuffer.push((Integer)#toBuffer);}
72         present(requestData){data = ((Integer)myBuffer.pop()).intValue();
               emit fromBuffer(data);}
73         if(!myBuffer.isFull()){
74           emit bufferNotFull;
75         }
76         if(!myBuffer.isEmpty()){
77           emit bufferNotEmpty;
78         }
79         pause;
80       }
81     }
82   }
83
84   PABROCD(..)->PABRO
85   CABROCD(..)->CABRO
86   BUFFERCD(..)->BUFFER
```

The trace in Figure 2.3 shows a snapshot of the running system. When the PABRO clock domain first receives input signals A1 and B1 it sends the first Fibonacci number via channel producer to the BUFFER. This rendezvous and data transfer is successful at some point of time (shown with the dashed line). As the circular buffer is no more empty the BUFFER clock domain tries to send the first Fibonacci number to the CABRO clock domain. This rendezvous through channel consumer is successful after the CABRO clock domain receives the signals A2 and B2 from the environment (the ABRO state machine is shown in Figure 2.1). Please note that the success of this rendezvous removes the first data value from the buffer and reduces the size of the buffer by one. Next we input signal R1 to the PABRO clock domain, this preempts the `send`/`receive` constructs working on the producer channel (note that the `abort(R1)` encapsulates the send producer line 11 and hence, preemption on R1 preempts the `send`). The corresponding `receive` is preempted on its own. Sending signal R2 to the CABRO clock domain preempts the `receive` construct on channel consumer. But, in the case of the consumer channel the corresponding `send` does not get preempted in the same logical tick. This is because the synchronous parallel reaction (lines 52- 65) does not enter the body of the `present` statement as the buffer

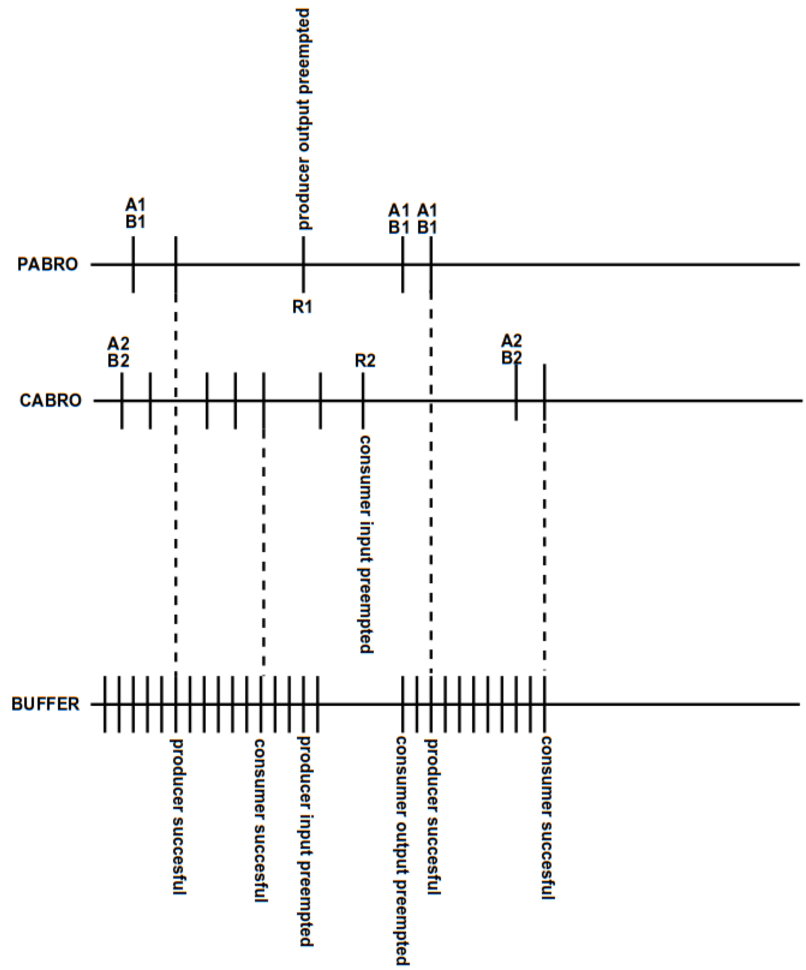

Figure 2.3: Trace for one run of PCABRO

is empty. When signals A1 and B1 are again sent to the PABRO clock domain the second Fibonacci number is generated and sent to the BUFFER clock domain. This in turn makes the buffer non empty and thus, the second synchronous parallel branch (lines 52- 65) of the BUFFER clock domain enter the `present` branch and this time around preempts the `send` construct on the channel consumer in response to the preemption of the corresponding `receive`. The system repeats this behavior in every iteration of input signals. Finally, it should be mentioned that this is only one possible behavior other type of behaviors are also possible depending upon the sequence of input signals received by the system. We will now discuss two key features of the

PCABRO program in the following:

1. *Synchronization between asynchronous clock-domains*: Asynchronous clock domains synchronize using *rendezvous* communication over point to point channels. A data producer clock domain sends the data over an output channel using the *send* statement. A matching receive statement over the same channel name (which is now an input channel) is used by a consumer clock domain to receive the data. Communication over channels is blocking. If the producer arrives at a send before the matching consumer is ready (hasn't reached its receive) then the producer blocks. Similarly, if the consumer arrives at its receive before the matching producer is ready, it blocks. This is unlike the synchronous broadcast communication within a clock domain using signals (see Chapter 3).

2. *Simple approach for coding a critical sections*: In the example in Listing 2.3, the producer (PABRO) and the consumer (CABRO) have a shared circular buffer (BUFFER). In a traditional concurrent program, the programmer has the responsibility of ensuring that critical sections (such as the shared buffer) is safely accessed. This is achieved through complex synchronization primitives such as mutexes or monitors. Such primitives have high programming and implementation overheads. In SystemJ by encapsulating the shared data in a separate clock domain, there is no need to code explicit critical sections. Also, synchronization using `send` and `receive` is much simpler than using OS synchronization primitives. Most importantly, the overall code is relatively easy to understand.

In this chapter we have illustrated the main features of SystemJ using three different pedagogic examples. In the next chapter, we will present the syntax and intuitive semantics of the SR constructs of SystemJ.

Chapter 3

Synchronous Reactive Programming in SystemJ

3.1 The pause statement

The `pause` statement indicates finishing of the logical tick (one time instant) in SystemJ. SystemJ communicates with the environment only after completion of a logical tick. Thus, every SystemJ reaction should end with a `pause` statement. The syntax of the pause statement is;

```
pause;
```

3.2 Signals

Signals form the main communication components in SystemJ. Signals are used to communicate with environment and also between synchronous parallel reactions within a clock-domain. Signals in SystemJ always have a status, a `true` status signal means that the signal is present, while a `false` status would represent an absent signal. Signals might also have a value, which can be any Java primitive type or object. The syntax for declaring a signal is;

```
input [type] signal <name>;
output [type] signal <name>;
[type] signal <name>;
```

Thus, signals have three different incarnations. The `input` and `output` signals can only be declared in the interface of the clock-domains. These are called interface signals and are used to communicate with the environment. The signals that are not declared with either the input or output qualifiers are called *local* or *internal* signals and are used to communicate between synchronous parallel reactions within a clock-domain. These are processes combined using the `||` operator and run concurrently and in lockstep. The `type` operator identifies the type of signal, it is an optional argument. A signal without a `type` is considered to be a pure signal i.e., it does not have any value. The `type` argument can be any Java type or object. Finally, the signal `name` is a mandatory argument. Signal names are unique in each clock-domain, i.e., a signal name cannot be repeated within a clock-domain.

3.3 The `emit` statement

The `emit` statement is used to broadcast a signal, i.e., the emitted signal is visible in all synchronous parallel reactions enclosed in the same clock-domain. This involves telling the program that a signal is present and also setting its value if it is a valued signal, i.e., it has a `type` declaration. Emitting a signal is simply done using the syntax;

```
emit <name> [(value)];
```

The `name` is a mandatory argument, while the `value` is an optional argument. It is never necessary to emit a value, even for a valued signal. The only signals that can be emitted are `output` signals and local signals. The `input` type signals cannot be emitted. Emitting an `input` signal will give a compiler error. Finally, it should also be pointed out that the status of the emitted signal is set high for only one logical instant of time, but, the emitted value is persistent over logical ticks, until rewritten by another emission of the same signal. Lastly, but most important, the emitted signals are only visible to all other reactions in *the next logical tick*, i.e., it is delayed by one tick (This is illustrated in the next section).

3.4 Obtaining the signal value

The `#` operator is used to get the emitted signal value. The emitted signal value, which can by any Java primitive type or object, needs proper casting when used with the `#` operator. The syntax is;

```
int my_variable = [(cast)] #<name>;
```

Here the compulsory `name` argument is the signal name whose value is extracted, while the `cast` is an optional argument used when the signal holds a Java object type value.

Listing 3.1 shows an example of using signals.

Listing 3.1: Signal example

```
1  int signal S;
2  ArrayList signal P;
3  emit S; //emitting the signal without a value
4  emit S(24); //emitting signal S with a value of 24
5  ArrayList list = new ArrayList();
6  list.add(24);
7  lit.add(25);
8  emit P(list); //emitting signal P with the value of
       ArrayList
9
10 pause; // Delayed by one tick
11 int t = #S; // obtaining the signal value
12 if(t == 24) {
13     .. // do something
14 }
15 ArrayList list2 = (ArrayList)#P; //obtaining the signal
       value P
16 // note the casting
17 if(list2.get(0) == 24){
18   .. //do something
19 }
```

In the code signal S (line 1) holds an integer value, while P (line 2) holds an ArrayList value. Signals S and P can be emitted both with or without a value. The values of these signals are obtained using the `#` operator as shown in lines 11 and 15. Note that as mentioned in Section 3.3, emitted signals are only visible in the next logical tick. Therefore one needs to insert the `pause` statement (line 10) before the values of the signals for S and P can be obtained. Please note that the signal values are persistent over time but statuses are not.

3.5 Synchronous conditional constructs

SystemJ provides a number of conditional constructs, which operate on signals for programming control-flow. This section describes all these conditional control-flow constructs

3.5.1 The `present` statement

The `present` statement checks if the signal is present in any given instant of time. For the signal to be present it either needs to be emitted or it needs to come in from the environment. The `present` statement can be used with all the different signals, `input`, `output` and local. The syntax for the present statement is

```
present(<name>){
  .. // computation
} else {
  .. // computation
}
```

The `present` statement only works on signals not on Java expressions. Using Java expressions in the `present` statement will give a compile time error. The Java conditional expressions can be tested using the normal Java `if-else` constructs. The `name` argument in the `present` statement can be a signal expression where signal names are combined using the Java logical operators.

3.5.2 The `abort` statement

The `abort` statement is used to preempt an ongoing computation if the signal is present. The `abort` statement can be considered equivalent to implementing an *Interrupt Service Routine* (ISR). The `abort` construct has the following syntax;

```
[weak] abort([immediate] <name>){
  .. // computation
}
```

In the above syntax the signal `name` is a compulsory argument. This name can be a signal expression of more than one signal name combined using Java's logical operators. The `immediate` construct is optional. If qualified with the `immediate` construct then the `abort` statement checks for the `name` expression to be present from the very first instant of time, else the check for the `name` expression is delayed by one logical tick. The `weak` qualifier is also optional. The `weak` qualifier preempts the enclosed computation after a logical tick is over. Without the `weak` qualifier the preemption will happen even before entering the computational node. Listing 3.2 shows the implementation of an ISR with different behaviours.

Listing 3.2: Implementing ISRs with `abort`

```
/* ------- The simplest form of abort statement ------- */
emit S;
pause; // delayed
abort(S || P){
  .. // some computation
```

```
6     pause;
7   }
8   present(S){
9     .. // ISR handler for signal S
10  } else {
11    present(P) {
12      .. // ISR handler if signal P is present
13    }
14  }
15  /* ----- The immediate form of abort statement ----- */
16  emit S;
17  pause; // delayed
18  abort(immediate (S || P)){
19    .. // some computation
20    pause;
21  }
22  present(S){
23    .. // ISR handler for signal S
24  } else {
25    present(P){
26      .. // ISR handler if signal P is present
27    }
28  }
29  /* ----- The weak form of abort statement ----- */
30  emit S;
31  pause; // delayed
32  weak abort(immediate (S || P)){
33    .. // some computation
34    pause;
35  }
36  present(S){
37    .. // ISR handler for signal S
38  } else {
39    present(P){
40      .. // ISR handler if signal P is present
41    }
42  }
```

In the simplest case first signal S is emitted, the `abort` statement does not check if the expression S`||`P is true and the computation is carried out until the `pause` statement (line 6). Thus, there is no preemption in the first instant of time, the `abort` statement starts checking if the expression S`||`P is true only from the second instant of time.

In the `immediate` case (lines 16 to 28) the `abort` statement checks if the expression S`||`P is true from the very first instant of time. Hence, the `abort` statement preempts the computation, in-fact no computation is carried out at all. Next, the `present(S)` statement succeeds and the ISR for signal S is invoked (line 23)

In the final case (lines 30 to 42) the `abort` statement does preempt the computation but only after the `pause` statement is hit at line 34. After the preemption, the `present(S)` statement succeeds and the ISR for signal S is invoked just like in the previous case.

Finally, a designer can create prioritised preemptive control-flow using nested `abort` statements.

3.5.3 The `suspend` statement

The `suspend` statement is another form of preemption. While the `abort` statement completely aborts the enclosing computation, the `suspend` statement preempts the enclosing computation for one logical instant of time. In the next logical instant of time the enclosing computation proceeds further. The syntax for the `suspend` statement is;

```
[weak] suspend([immediate] <name>){
  .. // some computation
}
```

Just like the `abort` statement the `suspend` statement can be qualified with the `immediate` and `weak` primitives. The compulsory name argument can be an expression of signal names combined using the Java logical operators.

3.5.4 The `await` statement

The `await` statement is provided for convenience. The `await` statement waits for the signals to be present before proceeding further. It is a blocking construct effectively implementing polling on a signal status. The syntax for the `await` statement is;

```
await([immediate] <name>);
```

A programmer can also implement the `await` statement as follows;

```
abort([immediate] <name>){
  while(true){
    pause;
  }
}
```

Where `name` can be a signal expression combined using the Java logical operators.

3.6 Looping constructs in SystemJ

SystemJ only allows the infinite temporal loop, which lasts for one logical tick for each iteration. Other loops, which count on variables and then break (the normal Java style loops), can only contain Java computations and no control-flow construct can be present within such loops. While each iteration of the SystemJ loop takes one logical instant of time, all iterations of the Java style loops take a single logical instant. The syntax for the SystemJ loop is;

```
while(true){
  .. // control-flow constructs
  pause; // This is required
}
```

An incorrect SystemJ loop would be like this;

```
for(int y=0 ; y<89; ++y){
  .. // control-flow constructs like pause, signal
     etc.
}
```

The `while` loop above will compile fine but the for loop will give a compile time error. But a counting loop with Java only constructs would be fine. For example, the code below will compile fine,

```
for(int y=0 ; y<89; ++y){
  ++y;
  .. // other Java only computations
}
```

Please note that every loop body having control-flow constructs should finish with the `pause` statement, for example the `while` loop above.

3.7 User controlled preemptions: the `trap` and `exit` statements

The previously introduced preemption constructs abort and suspend work with signals. SystemJ also provides a user controlled preemption construct called the `trap` statement. The `trap` statement is akin to Java's `try-catch` statement. SystemJ provides its own preemption constructs since the `try-catch` constructs have different semantics, which are incompatible with SystemJ semantics. The syntax for the `trap` statement is;

```
trap(T){
  .. //some computation
```

```
  exit(T);
}
```

The **trap** statement encloses the body of the computation, which can be preempted by the `exit` statement. Trap statements can be nested. In a nested scenario the outermost `trap` construct has the highest priority. The `trap` and `exit` constructs help the programmers implement counting loops with control-flow. Listing 3.3 shows an example use of `trap` statement for implementing a Java style counting loop.

Listing 3.3: Java-style counting loop using `trap`

```
int counter = 0;
trap(T1){
  while(true){
    emit S;
    ++counter;
    if (counter == 89)
      exit(T1);
    pause;
  }
}
```

Signal S will be emitted 89 times (in 89 logical ticks) and then the while loop will be terminated because of the `exit` statement.

3.8 Synchronous concurrency in SystemJ

SystemJ uses synchronous concurrency primitive construct to describe and run reactions (both named and unnamed) concurrently and in lockstep ,i.e. the reactions running in parallel with each other will wait for each other to complete a logical tick before proceeding further. The syntax is;

```
p1 || p2
```

Here `p1` and `p2` are two reactions, which can be named or unnamed.

Now that the synchronous concurrency primitive has been defined we look at an example of nested traps with synchronous parallel concurrency to show the importance of `trap` priorities.

Listing 3.4: Trap priorities in ynchronous concurrency

```
int counter = 0;
trap(T1){
```

```
  trap(T2){
    {exit(T1);}||{exit(T2);}
  }
  emit S;
}
emit P;
```

In Listing 3.4 there are two `trap` statements nested together. Please note that when nesting `trap` statements the `trap` identifiers (in this case `T1` and `T2`) cannot have the same name. There are two unnamed synchronous parallel reactions running concurrently, both preempt the `trap` construct using their respective `exit` statements. In this situation the outermost `trap` statement takes priority and hence the signal P is emitted, not S. The programmer can create priorities using this mechanism.

3.9 Delayed signal semantics

The original SystemJ communication model between synchronous parallel reactions is based on instantaneous emission of signals and reactions on their presence [3]. Hence, any SystemJ reactive statements registered with signals must react to the change of their status in the same logical instant (tick). Consider two synchronous reactions forked at some arbitrary instant as shown in the following example program (Listing 3.5). Note that the clock-domain is said to be *closed* when there are no interface ports, i.e. no I/O signals and channels, connected to the external environment which, in this case, the parentheses used for declaring signals and channels is omitted as shown in line 1.

Listing 3.5: SystemJ program with cyclic (signal) dependency

```
1  CD->{ // no parentheses
2    signal A,B,C;
3    { // Reaction 1
4      emit A;
5      present(B)
6        emit C;
7      pause;
8    }
9    ||
10   { // Reaction 2
11     present(A)
12       emit B;
```

```
13          pause;
14      }
15  }
```

During compilation of this program, concurrency is removed by scheduling reactions in sequential order (cyclic scheduling). Suppose control flow first enters Reaction 1 where it emits signal A (line 4) and checks for the status of the signal B at `present(B)` statement (line 5). However, at this point of execution, the status cannot be determined as it can potentially be emitted from other synchronous reactions within the same instant. As a result, the Reaction 1 is blocked until the signal dependency is resolved and the control flow jumps to the Reaction 2. As signal A is already emitted from the first reaction, the presence check at `present(A)` (line 11) results in emission of signal B (line 12). Reaction 2 then finishes its current tick at the `pause` statement (line 13) and will be terminated in the next instant. When control flow returns to Reaction 1 to continue execution, it sees that the signal status for B is now resolved and emits the signal C (line 6).

The algorithm used to resolve signal dependencies presented in the previous example assumes that neighbouring reactions are only signal emitters that can change execution flow of the SystemJ program, which is not always true. Consider Listing 3.6 below.

Listing 3.6: A single reaction having cyclic (signal) dependency

```
1  CD->{
2    signal A,B,C;
3    emit A;
4    present(B)
5      emit C;
6    present(A)
7      emit B;
8    pause;
9  }
```

In this example, two synchronous reactions from Listing 3.5 are merged into a single reaction. When this program is executed, control flow blocks at `present(B)` (line 4) after emitting the signal A (line 3). However, as there are no other reactions in this clock-domain and control flow cannot proceed further to execute `present(A)` (line 6), the program is blocked indefinitely at `present(B)` statement.

Last program shown in Listing 3.7 is logically incorrect. Regardless of which execution path the control flow chooses (present or absent branch), the result of the program is inconsistent with its decision (e.g. presence or absence of the signal A).

Listing 3.7: An incorrect SystemJ program

```
CD->{
  signal A;
  present(A){
    ; // dummy statement
  } else
    emit A;
  pause;
}
```

To address such compilation challenges, signal communication model in SystemJ language is modified. In the new approach, every signal emitted in a particular instant is only visible to neighbouring synchronous reactions (reactions in the same clock-domain) after a single logical tick is elapsed. Consider the example program below.

Listing 3.8: SystemJ program with different execution behaviour

```
1  CD->{
2    signal A,B,O;
3    while(true){
4      emit A;
5      present(A){
6        emit O;
7        pause;
8      } else
9        emit B;
10     pause;
11   }
12 }
```

In original semantics, the signal A emitted in the while loop will be instantaneously visible to the `present(A)` statement at line 5 and emits the signal O in the true branch. However, in delayed semantics, the signal A in the first instant is not visible until the next instant. The trace of signal emissions from this program is shown in Figure 3.1.

If one desires to achieve in the delayed semantics similar behaviour as in the original one, an additional `pause` statement needs to be added between line 5 and line 4, which gives the program one instant delay to the emitted signal. However, one must be aware that the signals A and B will not be emitted within the same instant anymore.

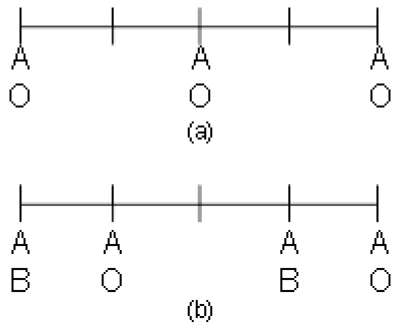

Figure 3.1: Result of the program execution in (a) original and (b) delayed semantics

3.10 Local variables

In SystemJ, all Java variables are local within the reaction where they are declared. Hence, writing the following program is prohibited.

Listing 3.9: Incorrect SystemJ program: variable `var` is shared among three reactions

```
CD->{ // Reaction 1
  int var = 0;
  { // Reaction 2
    var = 20;
  }
  ||
  { // Reaction 3
    System.out.println(var);
  }
}
```

In order to pass the integer value 20 to the Reaction 3, the programmer now has to use SystemJ signal as shown below in Listing 3.10. The parentheses used with the emit statement will set the value of the signal with the integer variable var which will be captured in the Reaction 3 in the next instant (delayed). The value is read by using the hash (#) operator followed by the name of the signal (line 10). Note that, since any variables declared inside a particular reaction are local to that reaction, declaring duplicated variable names is possible (line 4 and 10). The compiler will treat such declarations as declarations of two different variables, each visible only within the reaction in which it is declared.

Listing 3.10: Correct SystemJ program: data is passed through SystemJ signal

```
CD->{ // Reaction 1
  int signal A;
  { // Reaction 2
    int var = 20;
    emit A(var);
  }
  ||
  { // Reaction 3
    await(A);
    int var = #A;
    System.out.println(var);
  }
}
```

Chapter 4

Asynchronous Programming in SystemJ

4.1 Clock-domain communicating using channels

CSP style rendezvous (i.e., a complete handshake) over channels is the only means of communication between clock-domains. The `send` and `receive` operators are used over channels for communication. Channels are point-to-point i.e., a send and receive over a channel 'C' cannot be done simultaneously in multiple reactions. The syntax for channel declarations is;

```
input <type> channel <name>;
output <type> channel <name>;
```

Channel declarations do not have any optional qualifiers. The `input` and `output` qualifiers are compulsory, they indicate two separate points of the same channel.

As shown in Figure 4.1, channels are declared in the interface (i.e. parentheses) of the clock-domain (lines 1 and 4).

Listing 4.1: Declaring channels

```
1 CD1(input boolean channel C;)->
2 { .. /* some computation */ }
3
4 CD2(output boolean channel C;)->
5 { .. /* some computation */ }
```

The `receive` statement works on the input point, while the `send` statement works on the output point of the channel. The syntax for `receiving` and sending are;

```
receive C;
send C(value);
```

where `value` can be any Java primitive or object type. Thus, for the `boolean` channel 'C' declared above the sending and receiving code would be;

Listing 4.2: Asynchronous `send` and `receive`

```
CD1(output boolean channel C;)->
{ send C(true); }

CD2(input boolean channel C;)->
{ receive C; boolean t = #C; }
```

The value of channel 'C' can be obtained after receiving it using the `#` operator. The syntax and semantics of the `#` operator have been described previously.

This finishes the overview of kernel SystemJ language and its programming practice. The reader should refer to the compiler, which should have been provided to look at the various examples.

Chapter 5

The SystemJ Tools and Runtime Environment

The purpose of this chapter is to provide an overview of the SystemJ compilation flow as well as guidance on extending SystemJ Runtime Environment (RTE). Upon completion of reading this document developers will be able to compile and run SystemJ programs on various types of execution platforms. This document assumes that the reader already understands basic terminologies of the SystemJ programming language.

5.1 Compilation flow

An overview of SystemJ compilation flow is shown in Figure 5.1. Currently one of two target platforms can be chosen when compiling SystemJ programs: standard Java Virtual Machine (JVM) or TP-JOP embedded platform [12]. For a standard JVM, the SystemJ compiler generates a java source file (.java) for every clock-domain in a SystemJ program (left branch of the compiler switch in Figure 1). These source files are then compiled using javac which produces bytecode class files (.class). On the other hand, the compiler is also able to produce more efficient executable code by splitting control and data computations in a program. This code can be compiled and deployed on a time-predictable execution platform called TP-JOP [12]. As shown in Figure 1, TP-JOP consists of two processor cores: ReCOP, which leads control-flow of a SystemJ program, and JOP [13], which executes, on demand (by ReCOP), data computations described in Java. Initially, ReCOP starts executing control-flow of the program (i.e. SystemJ kernel statements) until it reaches any Java statement. ReCOP then requests JOP for execution of the corresponding Java statement via a simple message passing mechanism.

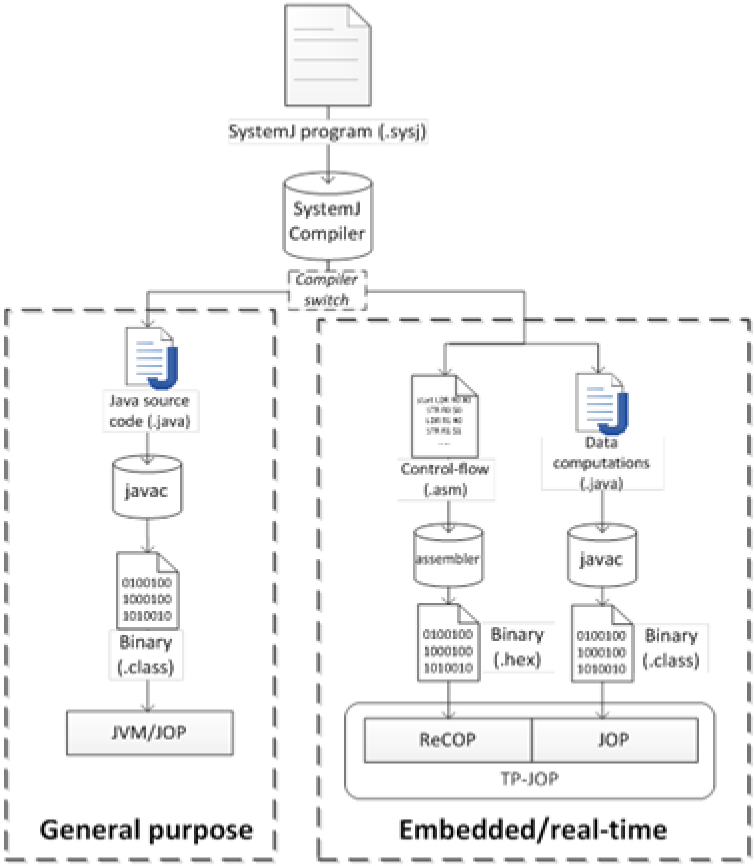

Figure 5.1: SystemJ compilation flow

JOP sends a message back to ReCOP upon completion, allowing ReCOP to continue its execution.

5.2 Compiling SystemJ programs

The System tool consists of three components: (1) the compiler, (2) runtime environment (RTE), and (3) the JDOM XML parser [14]. The compiler produces files for JVM or TP-JOP execution platform as explained in Section 5.1. The SystemJ RTE is needed in order to deploy the SystemJ program on various types of execution platforms. While the core of the RTE is designed to be compatible with any versions of JVM, it may be required to be extended in order to support platform specific features such as accessing I/O peripherals, and implementing signals and channels using the hardware resources provided by a platform. Lastly, the XML parser is needed to initialize a SystemJ program.

Assume the current working directory is in `$HOME/workspace` and the SystemJ tools (.jar files) are located in `$HOME/workspace/tools`. A SystemJ program (e.g. test.sysj) then can be compiled via the following command:

```
user@hostname ~/workspace $ java -cp "tools/*"
    JavaPrettyPrinter test.sysj
```

In case when the compiler warns that it cannot find the correct jdk path, make sure you have jdk installed instead of jre. For Windows, set the environment variable `$JAVA_HOME` to the directory where the jdk is installed.

Here, the compiler is invoked by running java with classpath including all the jar files of the SystemJ tools. The main method of the compiler is called `JavaPrettyPrinter`. The program is a simple Hello World program as shown below:

```
CD->{System.out.println("Hello World!"); pause;}
```

When the program is compiled both a clock-domain Java source code and the corresponding .class files are generated:

```
user@hostname ~/workspace $ java -cp "tools/*"
    JavaPrettyPrinter test.sysj

user@hostname ~/workspace $ ls
CD.class CD.java test.sysj tools/
user@hostname ~/workspace $ _
```

To run the clock-domain, the SystemJ RTE needs to be invoked with appropriate settings via *a configuration file*:

```
<System xmlns="http://systemjtechnology.com">
  <SubSystem Name="mySS" Local="true">
    <ClockDomain Name="myCD" Class="CD"/>
  </SubSystem>
</System>
```

Consider this file is named as `test.xml`, then the clock-domain ‘CD’ can be executed using the following command:

```
# Change colon (:) to semi-colon (;) for ".:tools/*" on Windows/Cygwin
user@hostname ~/workspace $ javac -cp ".:tools/*" systemj.
    bootstrap.SystemJRunner test.xml

... Some debugging messages ...
Hello World
Finished CD
```

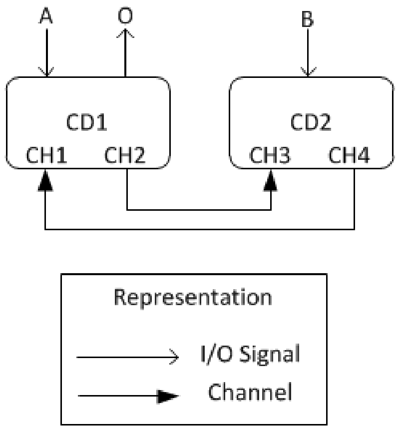

Figure 5.2: System configuration

5.3 Writing a configuration file

As already shown in Section 5.2, a user needs to write a configuration file in order to run the SystemJ program. The purpose of the configuration file is to provide the following information to RTE:

- Which subsystem (and thus clock-domains) to load (called *Local*).
- Interconnections with the external environment as well as other clock-domains via I/O signals, and channels, respectively.
- Locations of the remote machines, and how they can be reached through *Links*.

Listing 5.1: A simple SystemJ program

```
CD1(
  input signal A;
  output signal O;
  input String channel CH1;
  output String channel CH2;
)->
{ /* Program logic for CD1 */   }

CD2(
  input signal B;
  input String channel CH3;
  output String channel CH4;
```

```
)->
{ /* Program logic for CD2 */   }
```

We are going to use an example shown in Listing 5.1. It consists of two clock-domains that communicate with their environment via signals. These clock-domains are also interconnected via channels. Consider a designer wants to configure his/her system as shown in Figure 5.2. Here, the output channel CH2 of CD1 is connected to the input channel CH3 of CD2. Similarly, the output channel CH4 of CD2 is connected to the input channel CH1 of CD1. CD1 has one input and one output signal called A and O, respectively, while CD2 has only one input signal called B. Considering both clock-domains are executed on a same machine, one possible way to write a configuration file is:

```
<System xmlns="http://systemjtechnology.com">
  <SubSystem Name="mySS" Local="true">
    <ClockDomain Name="myCD1" Class="CD1">
      <iSignal Name="A" IP="192.168.1.25" Class="com.systemj.ipc.TCPReceiver
          " Port="80"/>
      <oSignal Name="O" IP="231.241.232.11" Class="com.systemj.ipc.TCPSender
          " Port="70"/>
      <oChannel Name="CH2" To="myCD2.CH3"/>
      <iChannel Name="CH1" From="myCD2.CH4"/>
    </ClockDomain>

    <ClockDomain Name="myCD2" Class="CD2">
      <iSignal Name="B" IP="192.168.1.25" Class="com.systemj.ipc.TCPReceiver
          " Port="81"/>
      <oChannel Name="CH4" To="myCD1.CH1"/>
      <iChannel Name="CH3" From="myCD1.CH2"/>
    </ClockDomain>
  </SubSystem>
</System>
```

The subsystem called `mySS` has additional attribute Local which is set to `true`, indicating this subsystem will be executed on a current RTE. The `SubSystem` can be nested with one or more `ClockDomain` elements meaning that the subsystem consists of (thus executes) the corresponding clock-domains. For example, clock-domains CD1 and CD2, compiled to class files CD1.class and CD2.class, are to be loaded and labelled as `myCD1` and `myCD2`, respectively. All interface signals and channels are declared as nested elements to `ClockDomain`. Input signals (`iSignal`) have four attributes: `Name`, `IP`, `Class`, and `Port`. The value of Name should be same as the signal name declared in the SystemJ program. The `Class` attribute specifies underlying protocol to be used when communicating with the environment. In this case, input signal A acts as a TCP server, which is implemented in a Java class called `com.systemj.ipc.TCPReceiver`. Attributes `IP` and `Port`, which corresponds to local IP and port numbers, respectively, are passed to the `TCPReceiver` instance as arguments during initialization phase of the program. On the other hand, output signal O acts as a TCP client, which is implemented in `com.systemj.ipc.TCPSender`. For output signals (`oSignal`) the environment's IP and port numbers are used.

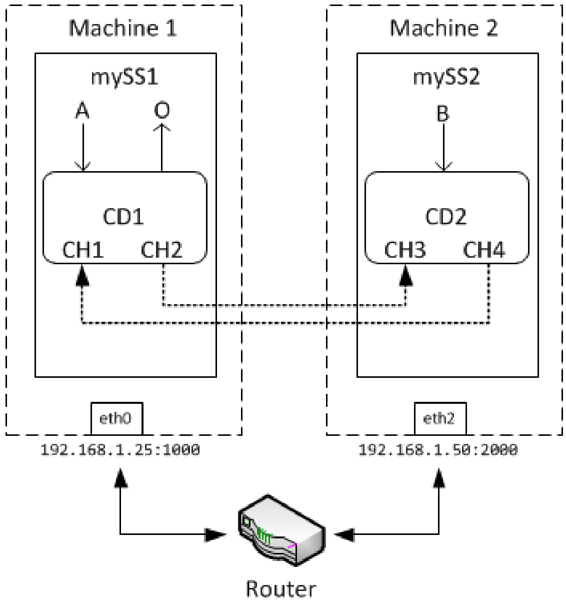

Figure 5.3: Each subsystem running on a different machine

When clock-domains are executed on a same subsystem, they use shared heap to exchange channel data. For example, the channels CH2 and CH3 establish a shared data structure in a heap in order to transfer a message via rendezvous mechanism. On the other hand, locations of the subsystems need to be provided in a configuration file when each of the clock-domains runs on a different machine as shown in Figure 5.3. In this example, CD1, which resides in `mySS1`, runs on a Machine 1 with IP address `192.168.1.25`. CD2, on the other hand, runs on a Machine 2 with IP address `192.168.1.50`. For the sake of simplicity, assume these machines are in a same local network. Then a configuration file for each of these subsystems can be written as shown below.

```
<!-- Configuration file for mySS1 -->
<System xmlns="http://systemjtechnology.com">
<Interconnection>
    <Link Type="Destination">
      <Interface SubSystem="mySS1" Class="com.systemj.ipc.TCPIPInterface"
          Args="192.168.1.25:1112"/>
      <Interface SubSystem="mySS2" Class="com.systemj.ipc.TCPIPInterface"
          Args="192.168.1.50:1113"/>
    </Link>
</Interconnection>
```

```
  <SubSystem Name="mySS1" Local="true">
    <ClockDomain Name="myCD1" Class="CD1">
      <iSignal Name="A" IP="192.168.1.25" Class="com.systemj.ipc.TCPReceiver
          " Port="80"/>
      <oSignal Name="O" IP="231.241.232.11" Class="com.systemj.ipc.TCPSender
          " Port="70"/>
      <oChannel Name="CH2" To="myCD2.CH3"/>
      <iChannel Name="CH1" From="myCD2.CH4"/>
    </ClockDomain>
  </SubSystem>

  <SubSystem Name="mySS2">
    <ClockDomain Name="myCD2" Class="CD2"/>
  </SubSystem>
</System>
```

```
<!-- Configuration file for mySS2 -->
<System xmlns="http://systemjtechnology.com">
<Interconnection>
    <Link Type="Destination">
      <Interface SubSystem="mySS1" Class="systemj.desktop.TCPIPInterface"
          Interface=""
          Args="127.0.0.1:1112"/>
      <Interface SubSystem="mySS2" Class="systemj.desktop.TCPIPInterface"
          Interface=""
          Args="127.0.0.1:1113"/>
    </Link>
</Interconnection>

  <SubSystem Name="mySS1">
    <ClockDomain Name="myCD1" Class="CD1"/>
  </SubSystem>

  <SubSystem Name="mySS2" Local="true">
    <ClockDomain Name="myCD1" Class="CD1">
      <iSignal Name="A" IP="192.168.1.50" Class="com.systemj.ipc.TCPReceiver
          " Port="80"/>
      <oSignal Name="O" IP="231.241.232.11" Class="com.systemj.ipc.TCPSender
          " Port="70"/>
      <oChannel Name="CH2" To="myCD2.CH3"/>
      <iChannel Name="CH1" From="myCD2.CH4"/>
    </ClockDomain>
  </SubSystem>
</System>
```

5.4 Generating a configuration file via compiler option

The compiler can automatically generate a configuration skeleton file based on the declared clock-domains, I/O signals, and channels, when the option "`--config-gen`" is passed:

```
user@hostname ~/workspace $ java -cp "tools/*"
   JavaPrettyPrinter --config-gen test.sysj
```

```
user@hostname ~/workspace $ ls
CD.class CD.java test.xml test.sysj tools/
user@hostname ~/workspace $ _
```

As previously mentioned, the file generated is only a skeleton; programmers still need to fill in missing attribute values in the file, e.g. channel connections and signal port numbers.

5.5 An overview of elements used in a configuration file

The following tables provide a short description of each element used in a configuration file.

Element	Description	Nested elements
`System`	A top-level entity describing a SystemJ system. Every configuration file should have a single System as a root element.	`Interconnection*`[1], `SubSystem+`[2]
`Interconnection`	`Interconnection` is a collection of `Links`	`Link*`
`Link`	`Link` is a collection of `Interfaces` that provides a way to exchange SystemJ channel data between subsystems running on remote machines.	`Interface+`
`Interface`	Provides a physical location of subsystems.	–
`SubSystem`	Consists of `ClockDomains`	`ClockDomain+`, `Scheduler*`
`ClockDomain`	SystemJ clock-domain	`iSignal*`, `oSignal*`, `iChannel*`, `oChannel*`
`iSignal`	Input signal	–
`oSignal`	Output signal	–
`iChannel`	Input channel	–
`oChannel`	Output channel	–
`Scheduler`	Schedules clock-domain execution	`ClockDomain*`

5.5.1 List of attributes

System

Attribute	Description	Required
`xmlns`	Specifies a namespace for the elements used in the configuration file. Should be set to "`http://systemjtechnology.com`"	Yes

Interconnection

None

[1]zero or more elements
[2]one or more elements

Link

Attribute	Description	Required
`Type`	Specifies a type of link. Use `Destination` if the remote machine is addressable using a global identifier (e.g. IP address). Use `Local` if the remote machine is connected via a port such as `COM1`.	Yes

Interface

Attribute	Description	Required
`SubSystem`	Name of a subsystem connected via a link.	Yes
`Class`	Fully qualified Java class name that implements this interface	Yes
`Args`	An argument passed to the class file specified in `Class`.	No

SubSystem

Attribute	Description	Required
`Name`	Name of this subsystem.	Yes
`Local`	If true, load and execute all clock-domains that belong to this subsystem.	No

ClockDomain

Attribute	Description	Required
`Name`	Name of this clock-domain.	Yes
`Class`	Fully qualified Java class name of this clock-domain.	Yes

iSignal

Attribute	Description	Required
`Name`	Name of the signal.	Yes
`Class`	Fully qualified Java class name that captures inputs from environment.	Yes
`IP`	(If `Class` is equal to `TCPReceiver`) Local IP address that the server will bind to.	Yes for `TCPReceiver`
`Port`	(If `Class` is equal to `TCPReceiver`) Port number of the server	Yes for `TCPReceiver`

oSignal

Attribute	Description	Required
`Name`	Name of the signal.	Yes
`Class`	Fully qualified Java class name that provides an output to environment.	Yes
`IP`	(If `Class` is equal to `TCPSender`) Local IP address of the remote machine	Yes for `TCPSender`
`Port`	(If `Class` is equal to `TCPSender`) Connects to this port number on the remote machine.	Yes for `TCPSender`

iChannel

Attribute	Description	Required
`Name`	Name of the input channel.	Yes
`To`	Name of the output channel that this input channel is connected to. The format is `<ClockDomainName>.<ChannelName>`, e.g. `CD1.CH`.	Yes

oChannel

Attribute	Description	Required
`Name`	Name of the input channel.	Yes
`From`	Name of the input channel that this output channel is connected to. The format is `<ClockDomainName>.<ChannelName>`, e.g. `CD2.CH`.	Yes

Scheduler

Attribute	Description	Required
`Class`	Fully qualified Java class name that schedules clock-domains. (e.g. `com.systemj.ThreadScheduler`)	Yes
`Args`	An argument passed to the class file specified in `Class`.	No

5.6 Implementing input and output signal classes

In order to introduce new input/output signal classes, you need to create a Java class inheriting either one of the following abstract classes:

- `com.systemj.ipc.GenericSignalSender` for output signals
- `com.systemj.ipc.GenericSignalReceiver` for input signals

`GenericSignalReceiver` class has two abstract methods that need to be implemented by its subclasses, and two already implemented methods, which can be overridden if needed:

1. `public abstract void configure (Hashtable data)` – This method is called only once during initialization phase of the SystemJ program. The parameter data contains a set `<Key,Value>`, which maps attribute name (`Key`) to its value (`Value`) in a configuration file.

2. `public void getBuffer(Object[] obj)` – This method is called for every clock-domain end-of-tick (EOT). The runtime expects a signal status and a value to be retrieved from `obj[0]` (`java.lang.Boolean`) and `obj[1]` (`java.lang.Object`), respectively. This method can be overridden if required.

3. `public void setBuffer(Object[] obj)` – This method is called for every clock-domain End of Tick (EOT). The default implementation passes a signal status (`obj[0]`) and a value (`obj[1]`) to the current instance of the signal class. This is can be overridden if required.

4. `public abstract void run()` – During the initialization phase, the runtime invokes `GenericSignalReceiver.start()` to spawn a thread that runs `GenericSignalReciever.run()`. Spawning a new thread may be required if `GenericSignalReciever.run()` consists of blocking operations (e.g. `ServerSocket.accept()`). Otherwise, leave this as empty method.

On the other hand, `GenericSignalSender` class has the following methods:

1. `public abstract void configure(Hashtable data)` – See `GenericSignalReceiver`

2. `public boolean setup(Object[] data) throws RuntimeException` – This method is called when the signal is emitted in the previous tick, and should return either true or false (see below). `data[0]` always contains `java.lang.Boolean.TRUE` and `data[1]` contains a signal value (if there is any) which is any Java object. This method can be overridden if required.

3. `public abstract void run()` – This method is called for every clock-domain end-of-tick (EOT) only if previously called `setup(Object[] data)` returned true.

4. `public void arun()` – This method is called when the signal is not emitted (absent) in the previous tick. This method is empty and can be overridden if required.

Both `GenericSignalSender` and `GenericSignalReceiver` classes can be found in the runtime jar file. When working with signal classes, you must to import this jar file to the classpath of your development environment. The following diagram illustrates the interaction between a SystemJ program and the input/output signal instances.

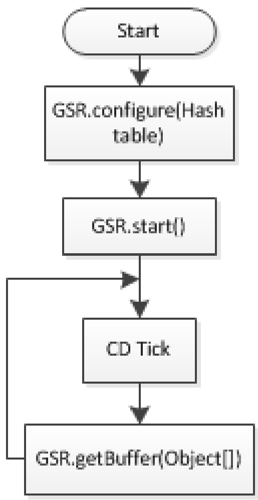

Figure 5.4: Interaction between `GenericSignalReceiver` and a SystemJ program

5.6.1 Example – InputFileReader

When a programmer uses `InputFilerReader` for an input signal A as follows:

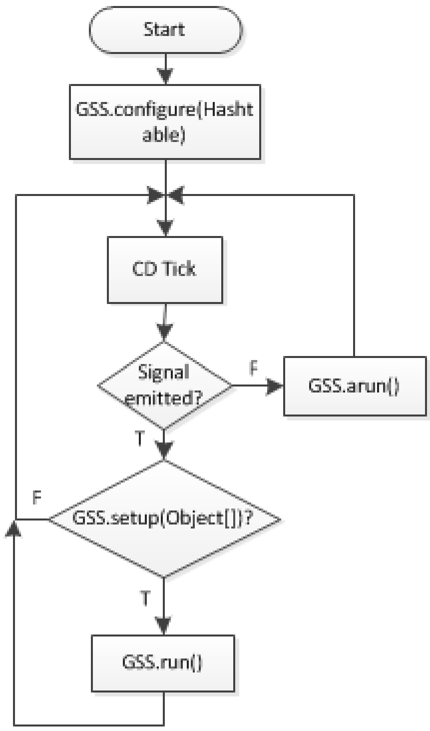

Figure 5.5: Interaction between `GenericSignalSender` and a SystemJ program

```
<iSignal name="A" Class="com.systemj.ipc.InputFileReader"
   File="myfile"/>
```

and the program reads a signal value from A, it will read a text file named “myfile” line by line.

Implementing `configure`

This method simply opens the file using the filename specified in the configuration file:

```
@Override
public void configure(Hashtable data) throws
    RuntimeException{
  filename = (String)data.get("File");
  try{
    br = new BufferedReader(new FileReader(filename));
  }catch(FileNotFoundException e){e.printStackTrace();}
}
```

Implementing getBuffer

This method reads the file line by line, and re-opens the file when it reaches EOF (returns null). Note that the signal is considered absent (i.e. `obj[0] = false`) when the line only consists of a character ';' otherwise present (i.e. `obj[0]= true`). `obj[1]` is assigned with a signal value (i.e. read text).

```
@Override
public void getBuffer(Object[] obj){
  try{
    String line = br.readLine();
    if(line == null){
      br.close();
      br = new BufferedReader(new FileReader(filename));
      line = br.readLine();
    }
    if(line.trim().equals(";")){
      obj[0] = Boolean.FALSE;
    }
    else{
      obj[0] = Boolean.TRUE;
      obj[1] = line;
    }
  }catch(Exception e){e.printStackTrace();}
}
```

Implementing run

Since reading the file can be done for each time `getBuffer` is called, no need to keep this thread alive (i.e. empty method).

```
@Override
public void run() { }
```

Lastly, make sure you call the constructor of `GenericSignalReceiver` inside the constructor of your class, if you are going to use the default `getBuffer` or `setBuffer` method:

```
public InputFileReader(){
  super(); // Initializes the buffer
}
```

5.6.2 Example – OutputFileWriter

When a programmer uses `OutputFilerWriter` for an output signal O as follows:

```
<oSignal name="O" Class="com.systemj.ipc.OutputFileWriter"
    File="myfileout"/>
```

and the program emits the signal with a value of type `java.lang.String`, it will write the string to the file named “myfileout”.

Implementing `configure`

Similar to `InputFileReader`, this method also needs to open a file with the name specified in the element’s attribute.

```
@Override
public void configure(Hashtable data) throws RuntimeException {
  if(data.containsKey("Name")){
    signalName = (String)data.get("Name");
  } else throw new RuntimeException("The configuration parameter 'Name'
      is required!");

  if(data.containsKey("File")){
    dir = (String) data.get("File");
  } else throw new RuntimeException("The configuration parameter 'Path'
      is required!");

  try{
    String fileName = new String(dir);
    file =  new File(fileName);
    file.createNewFile();
  }catch(IOException ioe){ioe.printStackTrace();}
}
```

Implementing `setup`

We are going to use default `setup` method in `GenericSignalSender`. Make sure to call the parent’s constructor:

```
public OutputFileWriter(){
  super();
}
```

Implementing `run`

In this method, we are going to retrieve a signal value (String) from `GenericSignalSender.buffer` and write it to the file that was previously opened in the configure method.

```
@Override
public void run() {
  Object[] obj = super.buffer;
  String data = (String) obj[1];

  try{
    writer = new BufferedWriter(new FileWriter(file,true));
    writer.write(data,0,data.length());
    System.out.println(data);
    writer.write("\n", 0, 1);
```

```
        writer.flush();
        writer.close();
      } catch(IOException e){
        e.printStackTrace();
      }
    }
```

References

[1] A. Malik, Z. Salcic, and P. S. Roop, "SystemJ compilation using the tandem virtual machine approach," *ACM Transactions on the Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 3, pp. 1–37, 2009.

[2] F. Gruian, P. Roop, Z. Slacic, and I. Radojevic, "The SystemJ approach to System-Level Design," in *The 4th International Conference on Formal Methods and Models for Codesign(MEMOCODE)*, July 2006.

[3] A. Malik, Z. Salcic, P. S. Roop, and A. Girault, "SystemJ: A GALS language for system level design," *Elsevier Journal on Computer Languages*, 2010.

[4] H. Park, Z. Salcic, K. I.-K. Wang, U. D. Atmojo, W.-T. Sun, and A. Malik, "A New Design Paradigm for Designing Reactive Pervasive Concurrent Systems with an Ambient Intelligence Example," in *ISPA, 2013 IEEE 11th International Symposium, Melbourne, Australia*, 2013.

[5] H. Park, A. Malik, M. Nadeem, and Z. A. Salcic, "The Cardiac Pacemaker: SystemJ versus Safety Critical Java," in *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES 2014, Niagara Falls, NY, USA, October 13-14, 2014*, p. 37, 2014.

[6] E. A. Lee, "The problem with threads," *IEEE Computer*, vol. 39, pp. 33–42, May 2006.

[7] A. Malik, Z. Salcic, and P. S. Roop, "SystemJ compilation using the Tandem Virtual Machine Approach," *ACM Transaction on Design Automation of Electronic Systems*, vol. 14, no. 3, pp. 1–37, 2009.

[8] G. Berry, "The Esterel v5 Language Primer - Version 5.10, release 2.0."

[9] D. Potop-Butucaru, S. A. Edwards, and G. Berry, *Compiling Esterel*. Springer Verlag, May 2007.

[10] P. Welch and F. Barnes, "Communicating Mobile Processes: introducing occam-pi," in *25 Years of CSP* (A. Abdallah, C. Jones, and J. Sanders, eds.), vol. 3525 of *Lecture Notes in Computer Science*, pp. 175–210, Springer Verlag, April 2005.

[11] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. Simone, "The synchronous language twelve years later," in *Proceedings of the IEEE*, pp. 64–83, 2003.

[12] M. Nadeem, M. Biglari-Abhari, and Z. Salcic, "Gals-jop: A java embedded processor for gals reactive programs," in *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pp. 292–299, 2011.

[13] M. Schoberl, "JOP: A java optimized processor," in *Workshop on Java Technologies for Real-Time and Embedded Systems*, November 2003.

[14] "Jdom xml parser." `http://www.jdom.org`, 2016. Last accessed: 03/04/2016.