SystemJ Programming Manual

Robert Connolly [1], Avinash Malik, Partha S Roop,
Zoran Salcic and Jeremy Stott
University of Auckland
Department of Electrical and Computer Engineering
Auckland Centre for Embedded Intelligence

[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. 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 publications [8, 3, 7, 6, 9] available from [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 clocks domains (see the PCABRO example in Chapter 2).

SystemJ targets different execution platforms. Examples of platforms are servers and desktop computers that require a Java Virtual Machine (JVM), embedded microprocessors that require at least a smaller version of JVM (J2ME or KVM) and Google's Android Platform, special embedded processors and multiprocessors on chip customised for SystemJ execution. Even pure hardware can also be synthesised from SystemJ specifications. These execution 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 the SystemJ IDE

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 recent paper by Lee [4] 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 nondeterministic.

SystemJ is a recent language [3, 10, 11] for demystifying the programming of highly concurrent and possibly distributed systems. It combines the synchronous elegance of the Esterel language [2, 12], with the asynchronous concurrency of CSP [13] and, the object-oriented data encapsulation capability of Java. This enables the programming of highly concurrent and distributed systems with ease. The new 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 [11] is proposed to enable effective compilation and formal analysis. SystemJ is an ideal language for the design of globally asynchronous and locally syn-

chronous (GALS) systems [11].

A SystemJ program (also known as a *system*) consists of one or more *clock domains*. Clock domains execute asynchronously (each at its own logical speed) and sometimes communicate 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 lockstep synchronously based on the *ticks* of this 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 a hierarchical tree shown on lower part of Figure 1.1.

Figure 1.1 also gives the graphical notation for GALS systems that all SystemJ programs comply with. On the top level are clock domains which consist of synchronous reactions. In this example, the clock domain CD1 communicates with the clock domain CD2 using channel C12 and clock domains CD1 and CD3 do not communicate directly at all while the clock domains CD2 and CD3 communicate via channels C23 and C32 respectively. Arrows show the direction of communication. Also, Figure 2 graphically shows which reactions within clock domains communicate through signals. Signal based communication is done using *synchronous broadcast* [2] which is explained in detail in Chapter 2. Arrows in Figure 1.1 show the emitters and receivers of a given signal. while there is always a single emitter, there can be more than one receiver of a given signal. For example, reaction R11 emits signal E, which is used in reaction R12. Similarly, reaction R12 emits signal F, which is used to control computation in reaction R11.

SystemJ programs are inherently concurrent and also continuously react with the adjoining *environment* at a speed determined by the environment. Hence, SystemJ is an ideal language for programming reactive systems [1].

We have graphically presented the environment of a SystemJ program with which the program exchanges information using inputs and outputs. SystemJ program allows communication with the environment only through special kind of signals, called input and output signals. The environment is very abstract for SystemJ programs. It can be physical environment with which SystemJ program interacts thorough electronic signals, it can be communication line through which messages are exchanged between the program and the environment or can be another program written in Java or any other programming language which understands the signal abstraction.

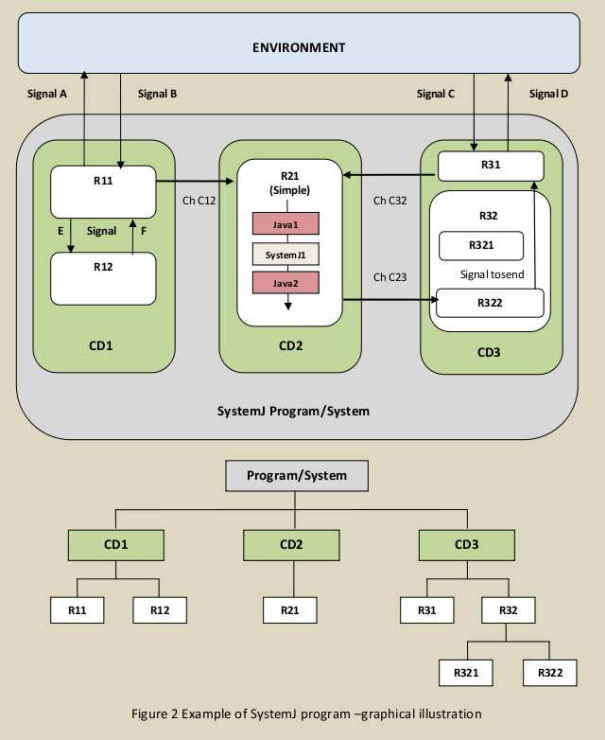

Figure 1.1: A typical SystemJ program's graphical illustration

1.2 The SystemJ IDE

SystemJ does not have it's own specific IDE since it utilises the framework provided by the well known Eclipse Java IDE. Eclipse is extremely powerful with a plug-in model, which supports enhancing the capabilities of software as and when needed. A complete description of the Eclipse IDE is out of the scope of this book and the user is assumed to have general knowledge

about IDEs. This chapter shows how to create a new SystemJ project, edit your SystemJ code and compile the program for debugging all within Eclipse using the SystemJ Development Tools. We demonstrate this with an example project called PC-ABRO (Producer Consumer ABRO), which will be used as example code throughout this manual.

The SystemJ Development Tools consists of a set of Eclipse plugins which should be installed in your version of Eclipse (currently supported are versions greater than 3.5.1) in order to allow you to compile and run SystemJ programs. A guide on the installation process for the latest version of these plugins is maintained on the SystemJ website `www.systemjtechnology.com`. This manual assumes that you have the SystemJ Development tools installed on your system.

Once the SystemJ Development Tools are successfully installed, you will be able to create a SystemJ specific project in your workspace, via the File ->New ->Other... menu option. This will open a dialogue window, as shown in the Figure 1.2. Scroll down to the SystemJ entry and select 'SystemJ Project' from the drop down list. Then proceed by clicking next.

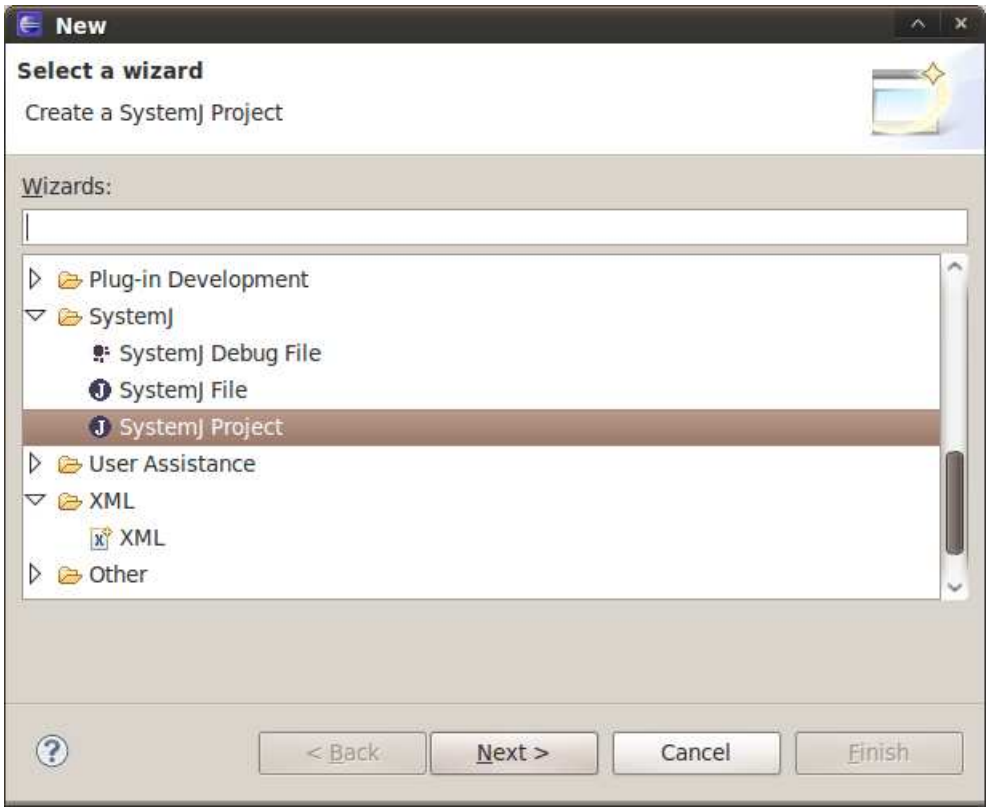

Figure 1.2: New Project Wizard - First Screen

In the next step of the wizard, you can enter a descriptive name for your project. For our example, you may enter the string 'PC_ABRO' (having replaced the hyphen with an underscore as hyphen is a reserved character in SystemJ and Java). After having selected your project name, click 'Next'. If you are new to SystemJ and Eclipse you can safely accept the default setting of the next screen in the wizard and click 'Next' again (advanced user may wish to add custom libraries here). The last screen in Figure 1.3 enables you to select a project template. W e will select the default template for now (other templates contain specific examples which may be of interest to you in your own experimentation). In the 'File Generation' section of the window you can select the default files to be generated and there are three options:

- 'Generate a .sysj file' - Generates a SystemJ source code file. You almost always want this selected.
- 'Generate a .sjio file' - Generates a SystemJ debugger scenario file. You will usually want this selected.
- 'Generate a .xml file' - Generates a SystemJ signal mapping XML file. You should select this if you wish to run your program in standalone mode (i.e. without the SystemJ debugger).

The details of what each of these files do and their respective formats will be explored as we come across them in the course of our example. For now it is sufficient to leave the default selections in place and click 'Finish'. After finishing the wizard you may be asked to open the 'SystemJ Perspective' of Eclipse. You should answer 'Yes' when prompted to do so.

Once you have completed the new project wizard, Eclipse will create your project and open the .sysj and .sjio files in the editor (in our case these are named PC_ABRO.sysj and PC_ABRO.sjio).

We can now begin looking around the features provided by the SystemJ Development Tools. The first feature to note is the SystemJ editor (see Figure 1.4), which provides syntax highlighting features for the SystemJ programmer. You will see from the default example code which is automatically generated by the IDE, that the editor highlights the SystemJ specific keywords and operators.

The next and most important part of the SystemJ Development Tools is it's integration with the run and debug profiles within Eclipse. These can be accessed in various ways, including from the main toolbar icons. Perhaps the easiest way to run or debug a SystemJ program is to right click on the .sysj file in question and use the entries 'Run As...' or 'Debug As...'. This will

automatically create the correct profile for your program with the default settings, which you can then modify later.

During development, the best way to run your program is via the SystemJ debugger (shown in Figure 1.5). By invoking a debug profile in Eclipse the SystemJ compiler will be automatically called to generate debugging code and then the debugger will then be launched. Currently, this debugger is external to Eclipse. In a future version of the SystemJ Development Tools it will be embedded as an integral part of the IDE.

The debugger is largely equivalent to a simulator which allows easy prototyping of the system and following of the internals of the system (such as the state of signals and channels in the system). Debugging is facilitated by allowing the user to enable / disable the inputs from the environment in every instance of the execution of a given clock domain. In response to a set of inputs provided in an instance, the debugger produces a set of outputs that are generated by the program and also highlights the current state of the program.

Debugging may also be done without user guidance by using the .sjio file which provides an input trace to the debugger automatically. You will understand this better after reading the following chapters of this document. You will notice that in the default .sjio provided the input 'pause' is given every three ticks over the course of the simulation.

When you have finished developing and testing one SystemJ system in isolation, you will come to requiring the run profile (rather than the previous debug profile). The main difference here is that the compiler is called to generate code which is suitable for distribution within a larger super-system. This is where the aforementioned SystemJ XML files come in. These provide signal mappings for the interface of your system, so that it may interact with other systems over a variety of transports. More details on the XML format used are given in Chapter 5.

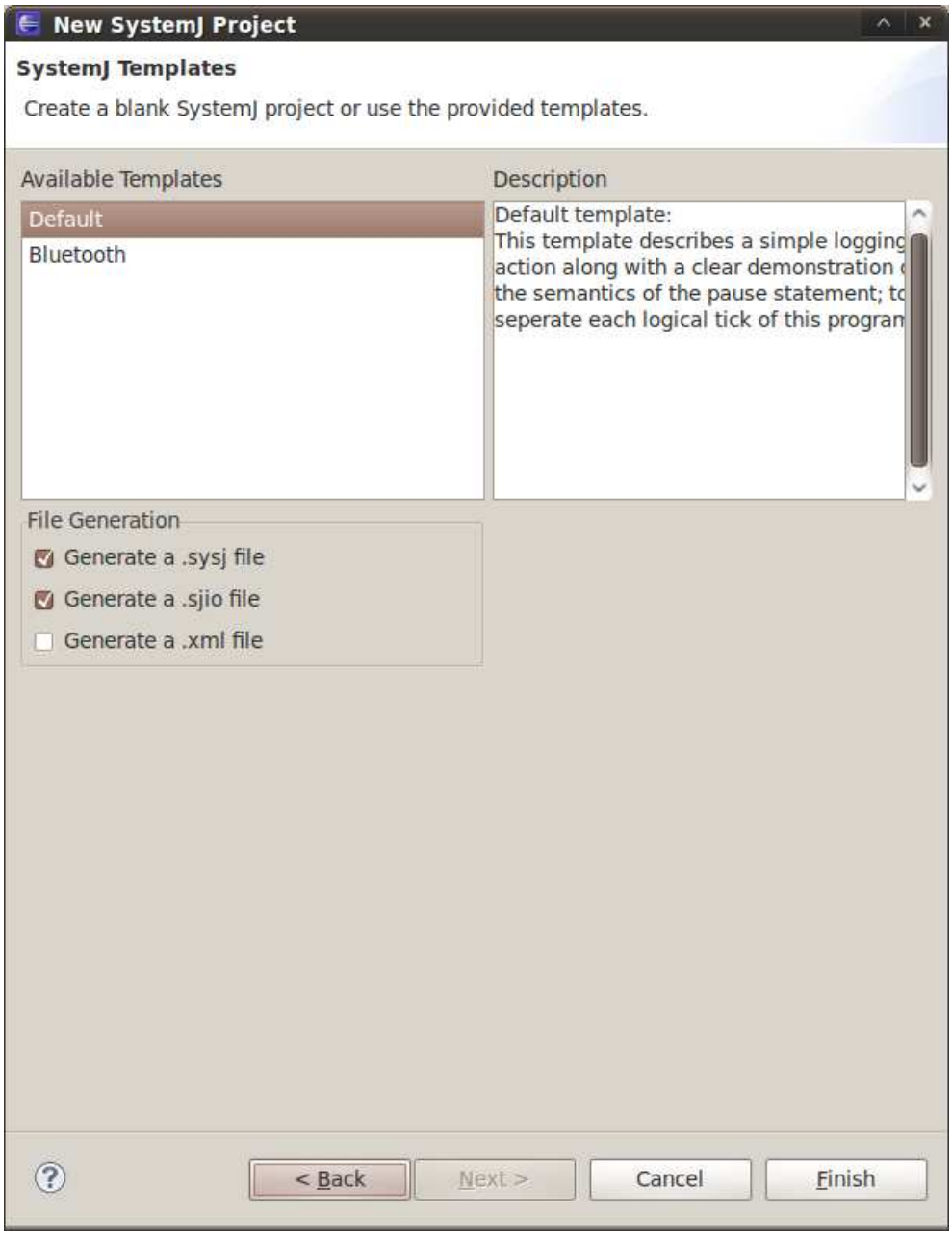

Figure 1.3: New Project Wizard - Final Screen

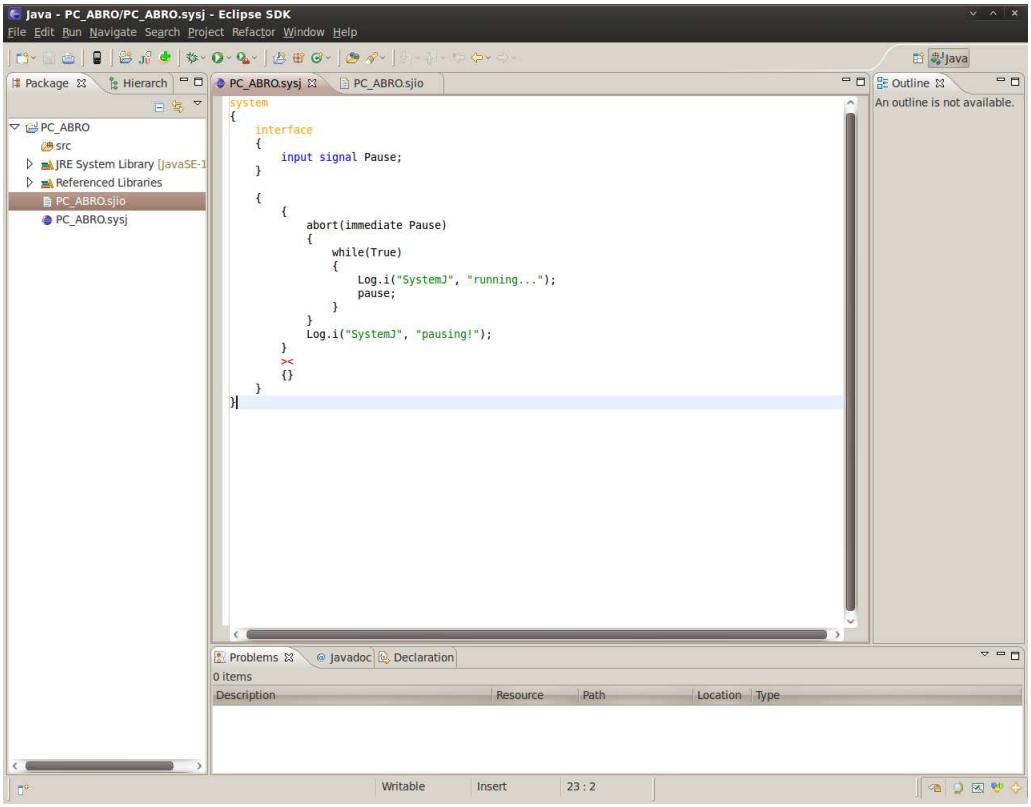

Figure 1.4: The SystemJ Editor Window.

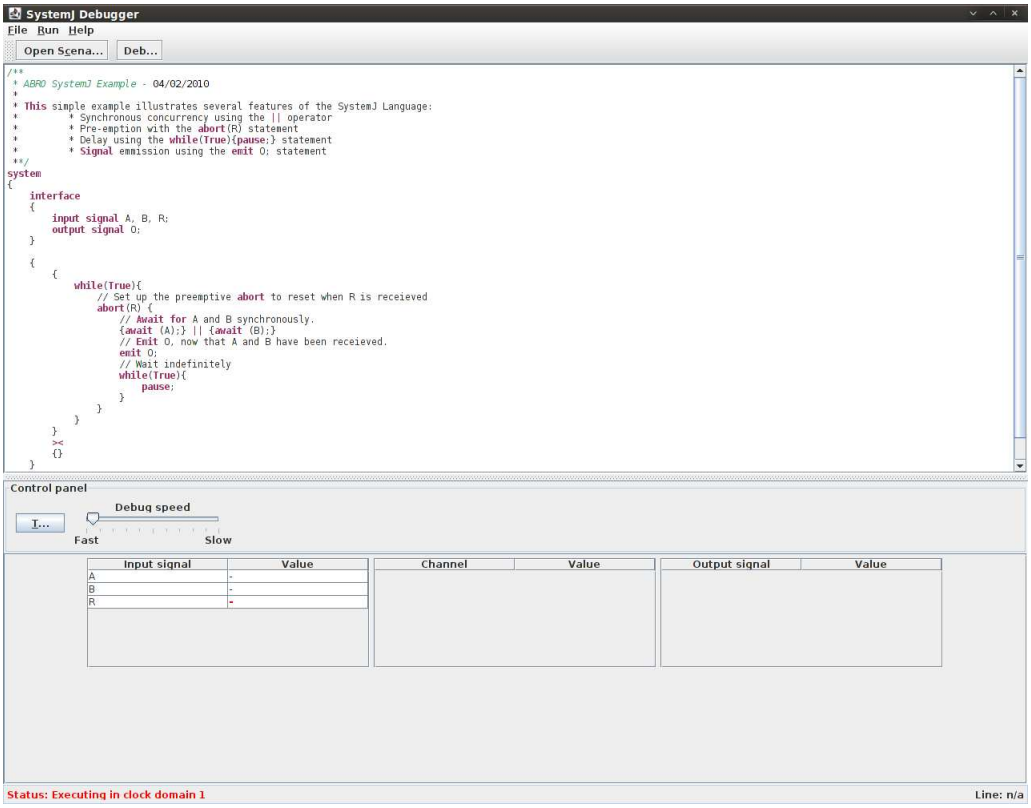

Figure 1.5: The Current SystemJ Debugger.

Chapter 2

SystemJ by Example

In this chapter we introduce the SystemJ programming style through three different examples. First the ABRO example by Berry [2] is adapted for motivating the synchronous reactive (SR) programming style [1] 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

Reactive systems continuously interact with their environment at a speed determined by the environment. Synchronous reactive (SR) programming style [1] 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).

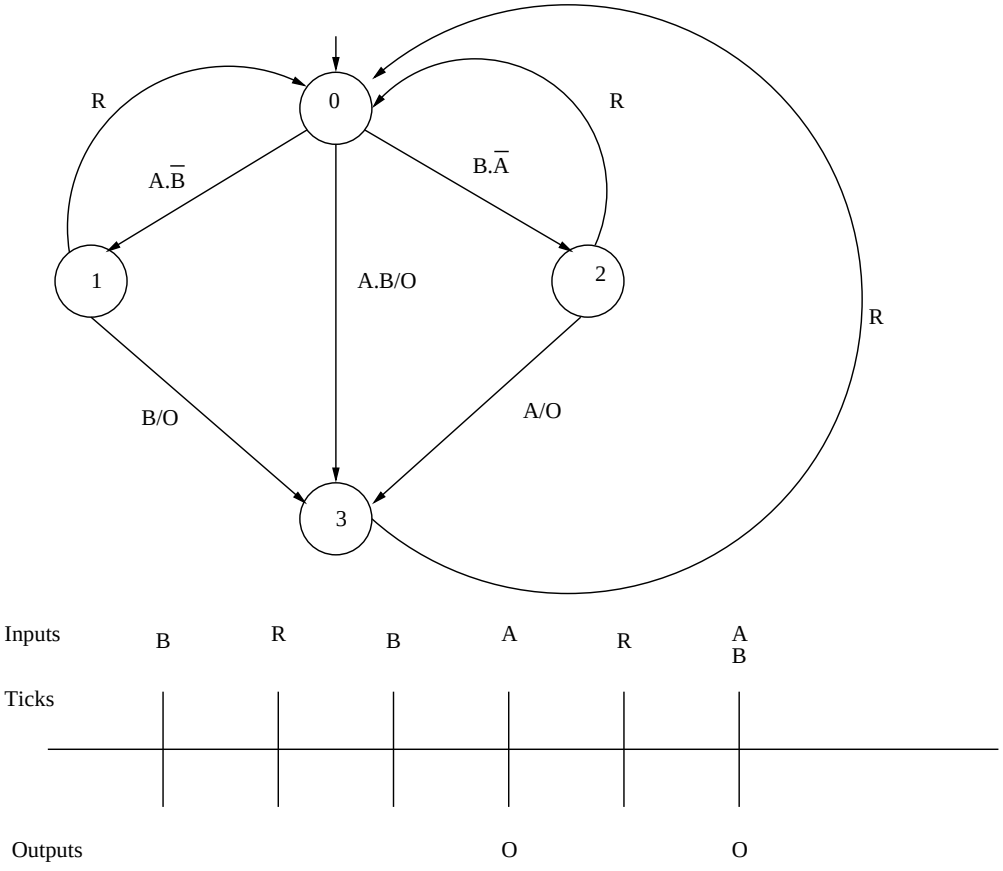

Figure 2.1: ABRO as a FSM and one behaviour trace

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 [2]. 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  system
2  {
3      interface
4      {h
5          input signal A, B, R;
6          output signal O;
7      }
8
9      {
10         {
11             while(True){
12
13                 abort(R) {
14                     {await (A);} || {await (B);}
15                     emit O;
16                     while(True){
17                         pause;
18                     }
19                 }
20             }
21         }
22         ><
23         {}
24     }
25 }
```

The SystemJ code for this application is shown in the Listing 2.1. A SystemJ program is defined using the keyword *system* to capture the overall GALS system (see Chapter 1). Since the program is reactive, you start by defining the *interface* of the program (lines 3 to 7) where all input and out-

put 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 10 to 21 respectively. Since SystemJ in general, is GALS we have the asynchronous composition of two clock domains (using the SystemJ asynchronous parallel operator >< on line 22). Since ABRO is just a SR program without asynchrony, the second clock domain shown on line 22 is empty.

The actual behaviour of ABRO is captured between lines 11 to 21. ABRO behaviour being reactive starts with an infinite loop on line 11. At the start of this behaviour is a preemption construct called *abort* (on line 13) which has a body marked between lines 14 to 19. 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 19).

The first statement of the abort body at line 14 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 15).

Note that ABRO behaviour demands that only after the occurrence of R the ABRO behavior can restart from line 11. 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 17) is a delay statement that delays for one tick. By enclosing the pause within an infinite loop, we effective have a halt. The behavior 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 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 using the >< to compose these two reactions asynchronously, we have created a simple GALS system. The use of the >< operator will force the reactions run at different rates. 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. More details on the asynchronous operator are described in Chapter 4.

Listing 2.2: AABRO in SystemJ

```
1  reaction abro(: input signal A, input signal B,
2   input signal R, output signal O){
3      while(True){
4          abort(R) {
5              {await (A);} || {await (B);}
6                     emit O;
7                     while(True){
8                       pause;
9                     }
10         }
11     }
12 }
13
14 system
15 {
16     interface
```

```
17      {
18          input signal A1, B1, R1;
19          input signal A2, B2, R2;
20          output signal O1, O2;
21      }
22
23      {
24                  abro(A1,B1,R1,O1)
25          ><
26                  abro(A2,B2,R2,O2)
27      }
28  }
```

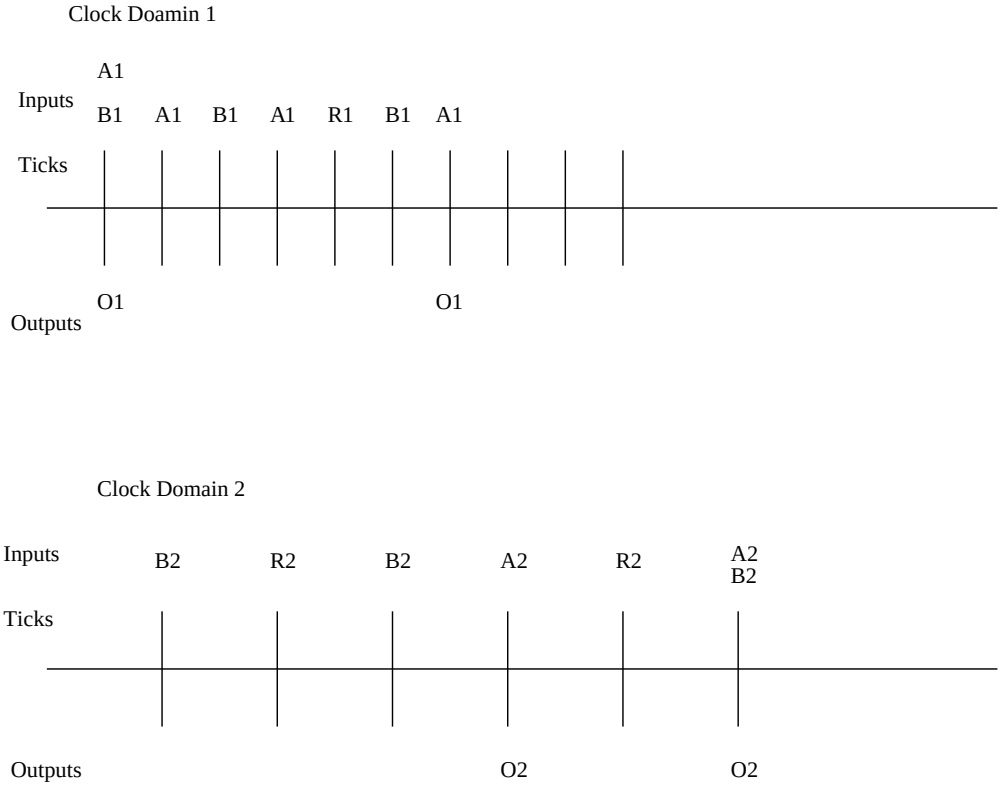

Figure 2.2: Sample behaviour trace of AABRO

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.

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 8-14). The CABRO clock domain receives the data from the PABRO clock domain via the circular buffer (lines 30). The BUFFER clock domain runs two reaction concurrently one (line 48) receives the data from the producer while the other (line 64) sends the data to the CABRO clock domain via channel consumer.

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(R)` encapsulates the `send producer` line 14 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 60-70) always loops into the else branch as the buffer 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 60-70) of the BUFFER clock domain enter the if 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:

- *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 that is instantaneous (see Chapter 3).

- *Simple approach for coding 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.

Listing 2.3: PCABRO in SystemJ

```
1 import buffer.*;
2 import Fibonacci.*;
3
4 reaction Pabro(: input signal A, input signal B,
5 input signal R, output signal O,
```

```
 6 output int channel producerChannel){
 7     while(True){
 8         abort(R) {
 9             {await (A);} || {await (B);}
10             FibonaciGenerator f = new FibonaciGenerator();
11             int data = 0;
12             emit O;
13             data = f.getNext();
14             send producerChannel(data);
15             while(True){
16                 pause;
17             }
18         }
19     }
20 }
21
22 reaction Cabro(: input signal A, input signal B,
23  input signal R, output signal O,
24  input int channel consumerChannel){
25     while(True){
26                 int data = 0;
27         abort(R) {
28             {await (A);} || {await (B);}
29             emit O;
30             receive consumerChannel;
31             data = #consumerChannel;
32             Log.i("PC-ABRO",
33             "Received next Fibonacci number: " + data);
34             while(True){
35                 pause;
36             }
37         }
38     }
39 }
40
41 reaction Buffer(: input int channel producerChannel,
42 output int channel consumerChannel){
43     CircularBuffer myBuffer = new CircularBuffer(100);
44     int data = 0;
45     {
46         while(True){
```

```
47              if (!myBuffer.isFull()) {
48                  receive producerChannel;
49                  data = (Integer)#producerChannel;
50                  if(data != null){
51                      myBuffer.push(data);
52                  }
53              }
54              else {
55                  pause;
56              }
57          }
58      }
59      ||
60      {
61          while(True){
62              if (!myBuffer.isEmpty()) {
63                 data = myBuffer.pop();
64                 send consumerChannel(data);
65              }
66              else {
67                  pause;
68              }
69          }
70      }
71  }
72
73  system
74  {
75      interface
76      {
77          input signal A1, B1, R1;
78          input signal A2, B2, R2;
79          output signal O1, O2;
80          input int channel producerChannel;
81          output int channel producerChannel;
82          input int channel consumerChannel;
83          output int channel consumerChannel;
84      }
85
86      {
87          Pabro(A1,B1,R1,O1, producerChannel)
```

```
88              ><
89              Cabro(A2,B2,R2,O2, consumerChannel)
90              ><
91              Buffer(producerChannel, consumerChannel)
92          }
93      }
```

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.

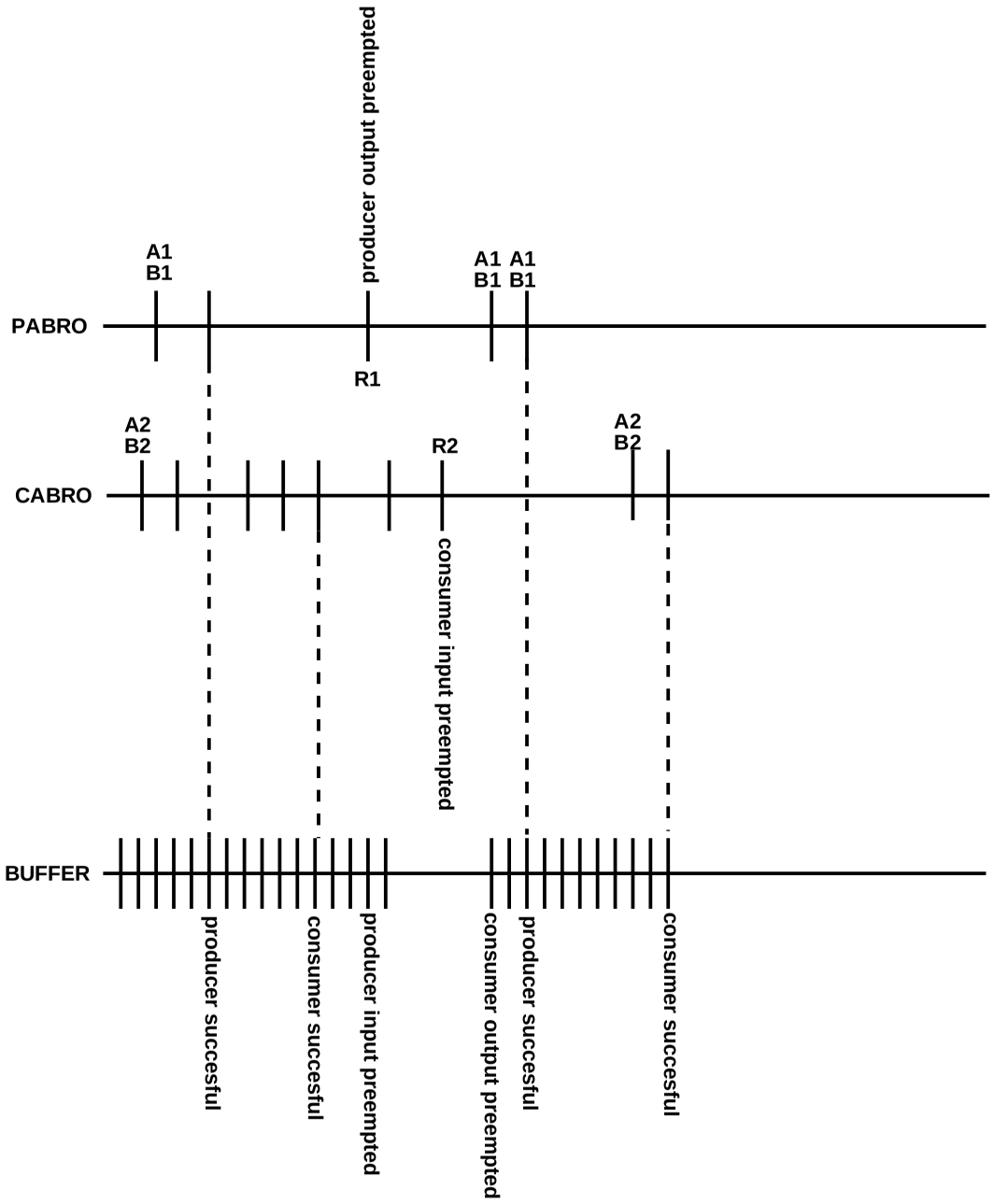

Figure 2.3: Trace for one run of PCABRO

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 `system`. 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 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, i.e., a signal name cannot be repeated in SystemJ.

3.3 The `emit` statement

The `emit` statement is used to broadcast a signal, i.e., the emitted signal is instantaneously visible in all synchronous parallel reactions together on emission. 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.

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(new Integer(24));
7  list.add(new Integer(25));
8  emit P(list); //emitting signal P with the value of ArrayList
9
10 int t = #S; //obtaining the signal value
11 if( t == 24)
12 {
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 {
19     .. //do something
20 }
```

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. When extracting the value from a primitive type signal (in this case S) there is no need to cast it (line 10). Obtaining the value from an object type signal needs appropriate casting. For example, when using the # operator in signal P, it needs to be cast into an ArrayList type (line 15). 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

```
1  //The simplest form of abort statement
2  emit S;
3  abort(S || P)
4  {
5      .. //some computation
6      pause;
7  }
8  present(S)
9  {
10     .. // ISR handler for signal S
11 }
12 else
13 {
14      present(P)
15      {
16          .. //ISR handler if signal P is present
17      }
18 }
19 //The immediate form of abort statement
20 emit S;
21 abort(immediate (S || P))
22 {
23     .. //some computation
24     pause;
25 }
26 present(S)
27 {
28     .. // ISR handler for signal S
29 }
30 else
31 {
32     present(P)
33     {
34         ..//ISR handler if signal P is present
35     }
36 }
37 //The weak form of abort statement
38 emit S;
39 weak abort(immediate (S || P))
```

```
40 {
41     ..//some computation
42     pause;
43 }
44 present(S)
45 {
46     ..// ISR handler for signal S
47 }
48 else
49 {
50     present(P)
51     {
52         ..//ISR handler if signal P is present
53     }
54 }
```

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 19 to 36) 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 28).

In the final case (lines 37 to 54) the `abort` statement does preempt the computation but only after the `pause` statement is hit at line 42. 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, 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

```
1  int counter=0;
2  trap(T1)
3  {
4      while(true)
5      {
6          emit S;
7          ++counter;
8          if(counter == 89)
9          {
10             exit(T1);
11         }
12         pause;
13     }
14 }
```

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. Listings in chapter **??** show the examples of || usage. Please note that internally the compiler will always execute the unnamed reactions before executing the named reactions. If there are more than one named or unnamed reactions composed by the || operator then the order in which the reactions are composed will be the order in which they will be executed. Thus, in the above

example if both `p1` and `p2` are unnamed reactions then `p1` will be scheduled for execution before `p2`. If `p1` is named, while `p2` is unnamed then `p2` will be executed before `p1`. This scheduling information can be used to optimize programs.

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 Synchronous Concurrency

```
1  int counter=0;
2  trap(T1)
3  {
4      trap(T2)
5      {
6          { exit(T1); } || { exit(T2); }
7      }
8      emit S;
9  }
10 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.

On a final note, the name spaces for Java, signals and `trap` identifiers are separated and hence the same name can be used for all, although for clarity reasons this is not recommended! Listing 3.5 shows such a usage.

Listing 3.5: Trap priorities in Synchronous Concurrency

```
1  int y=0;        //value y
2  int signal y; //signal y
3  trap(y)         //trap y
4  {
5      while(true)
6      {
7          if(y == 3)
8          {
9              exit(y);
10         }
```

```
11          else
12          {
13              ++y;
14              emit y(y);    //emitting signal y with value y
15          }
16          pause;
17      }
18  }
```

3.9 Java variables in SystemJ

It is possible to use both global and local variables in SystemJ. The use of global variables is discouraged but such variables are often needed to reduce the programming burden. Listing 3.6 shows an example of using both global and local variables in SystemJ.

Listing 3.6: Java Variable Usage

```
1  //Reaction reac2 can use the e variable because
2  //all variables declared are global variables
3  reaction reac2(:)
4  {
5      Log.i("SystemJ", "e is: "+e);
6  }
7
8  reaction reac1(int y, double t:)
9  {
10     int e=0;
11     Log.i("Y is: "+y+" T is: "+t);
12     reac2()||{++e}
13 }
14 system
15 {
16     interface{}
17     {
18         //All local variables are declared here
19         //and passed into the clock-domains
20         int y1=0,y2=5;
21         double t1=9.2,t2=8.5;
22         reac1(y1,t1)
23         ><
```

```
24              reac1(y2,t2)
25          }
26  }
```

As can be seen from the above example any variable declared is a global variable (in the above example variable `e` is declared in `reac1` but used in `reac2`. Thus, after declaring a variable it can be used anywhere as desired. This is for convenience and it should not normally be used by the designer. The local variables are always declared before the clock-domain initializations. Local variables are passed in the clock-domains as arguments.

Chapter 4

Asynchronous programming in SystemJ

4.1 The asynchronous composition operator ><

Reactions in SystemJ can be combined to run concurrently and asynchronously. Unlike the || operator the combined reactions do not wait for each other to complete. The reactions combined using the >< operator are called clock-domains. Clock-domains themselves can be a composition of one or more reactions combined using the || operator. The >< operator can only be used at the `system` level, not within a clock-domain. The syntax for using the >< operator is;

```
p1 >< p2
```

Where `p1` and `p2` are named or unnamed reactions. The scheduling order of named and unnamed reactions is similar to the || operator described previously. A concrete example is shown in listing 4.1.

Listing 4.1: Asyncronous Concurrency Example

```
1 system
2 {
3     interface
4     {
5     }
6     {
7         reac1()
8         ><
```

```
9           reac2()
10          ><
11          {}
12      }
13  }
```

4.2 Clock-domain communication 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' can only be written once in the whole SystemJ program. Also, a channel 'C' can only be declared once in the whole program. 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. Thus, every `input` channel needs to have an `output` channel partner and vice-versa. Missing a partner declaration will give a compile time error.

Channels are declared in the `interface` of the `system`.

Listing 4.2: Declaring Channels

```
1  system
2  {
3      interface
4      {
5          input boolean channel C;
6          output boolean channel C;
7      }
8      {
9          .. //some computation
10     }
11 }
```

In listing 4.2 the two points (input and output) of the same channel 'C' are declared in the `interface`.

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 4.3: Asyncronous Send and Receive

```
1 {
2     send C(true);
3 }
4 ><
5 {
6       receive C();
7       boolean t = #C;
8 }
```

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.

4.3 Using asynchrony to validate a system design

The asynchronous Model of Computation (MoC) in SystemJ can be used to design both distributed systems as well as testing and validating a system design. This section presents validating a system design, which might be synchronous or asynchronous using a clock-domain based test-bench.

Figure 4.1 shows an asynchronous protocol stack. There are two clock-domains, the first clock-domain is the test-bench and it models the packet generation from a network device. The second clock-domain is the protocol stack itself (this clock-domain models a partial protocol stack, it can be extend to model a complete communication stack like TCP/IP). The protocol stack consists of three synchronous parallel reactions. The first reaction

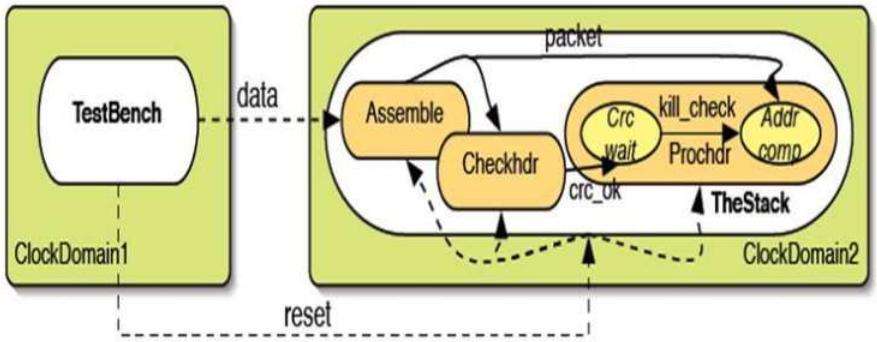

Figure 4.1: Validating a protocol stack

(Assemble) as the name suggests assembles the incoming packets. The second reaction (Checkhdr) checks the header information of the sent packet. Particularly, this reaction does a *Cyclic Redundancy Check* (CRC) on the packet obtained. The CRC check information is then propagated (using signal crc_ok) to the third reaction, which is the real stack. The stack reaction parses the packets and does computations on them like processing the header etc.

The code below shows the test-bench clock-domain. The stack itself has been abstracted out since the SystemJ code for the stack is generic.

Listing 4.4: Asyncronous System Validation

```
1  {
2      //The test-bench that emulates a network sending packets
3      send reset1(1); //reset the protocol stack first
4      pause;
5      byte tosend[] = {13,73,127,100,55,77};
6      int y=0;//a counter
7      trap(y)
8      {
9          while(true)
10         {
11             if(y == tosend.length)
12             {
13                 //It all packets are sent then just exit
14                 exit(y);
15             }
```

```
16                  else
17                  {
18                      //Send the packet
19                      send data1(tosend[y]);
20                      ++y;
21                  }
22                  pause;
23              }
24      }
25      Log.i("SystemJ", "Packet sent.");
26  }
27  ><
28  Protocol_Stack() //The abstracted protocol stack
```

The test-bench first resets the running protocol stack. Then it initializes a `byte` array (`tosend`) with some packets. This byte array can also be initialized in a loop to make a more complex packet stream. Next the test-bench just sends to packets to the protocol stack. This simple test-bench example shows the power of SystemJ in writing test-benches for validating complex systems. Since the designer can use the complete SystemJ language for writing test-benches very complex test cases can be written with ease. The complete asynchronous protocol stack example is available with the compiler.

Chapter 5

Communicating Between Multiple SystemJ Systems.

As stated previously, SystemJ programs communicate with the environment using signals while communication between reactions in different clock-domains requires channels. The signals and channels are an abstract means for communication and system specification. At runtime these signals and channels need to utilize some underlying physical layer for actual communication to take place.

The SystemJ Runtime supports communication via arbitrarily specified protocols. An XML file is used to bind signals and channels in SystemJ to the underlying communication methods/drivers. In addition, an interface exists to allow for custom serialization of data objects being sent via signals or channels. An example of the XML dialect used is presented in listing 5.1.

Listing 5.1: Signal Mapping Via XML

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <SystemJProgram>
3      <ClockDomain>
4          <Inputs>
5              <Signal
6                  Name=''A''
7                  Type=''mypackage.myCustomObject''
8                  IP=''224.0.0.0''
9                  Port=''44448''
10                 SocketClass=''systemj.signals.network.TCPReceiver''
11                 TimeOutValue=''1000''
12                 Serializer=''mypackage.MyCustomSerializer''
13                 BufferSize=''150''
```

```
14              />
15          </Inputs>
16          <Outputs>
17              <Signal
18                  Name=``O''
19                  Type=``String''
20                  IP=``224.0.0.0''
21                  Port=``44449''
22                  SocketClass=``systemj.signals.network.UDPSender''
23                  TimeOutValue=``1000''
24              />
25          </Outputs>
26      </ClockDomain>
27  </SystemJProgram>
```

First we present binding signals via XML, binding channels to the underlying physical layer is a little more complex and hence, deferred until Section 5.4. The `Inputs` and `Outputs` nodes in the XML file define the input and output signals in the SystemJ program (as specified in the interface section), respectively. These then contain the `Signal` nodes, which define the options for each signal by way of their attributes. There are several standard/required attributes, which are described in detail below:

- `Name`: The signal name as defined in the interface section of the SystemJ program.
- `Type`: The (Java) data type of the signal in question. N.B. This is not required for non-typed signals.
- `SocketClass`: The type of underlying communication layer to use, specified as a path to the Java class (a Sender or Receiver class) representing the communication method to use. The SystemJ Environment library provides some implementations of the interfaces for networked communication (in the `systemj.signals.network` package). Support for arbitrary communication protocols can be added by implementing the `systemj.interfaces.GenericSignalSender` and `systemj.interfaces.GenericSignalReceiver` interfaces (more information on how to do this is provided in section XX).
- `Serializer`: This attribute defines the serializer class to be used for serializing and deserializing data when communicating over specified protocol. If a serializer is not specified then SystemJ's generic network

sockets will try to use Java's serialization protocols. Using Java's serialization is a good idea when communicating between SystemJ and other Java programs running on different machines. Communication with other languages requires designers to extend the `systemj.interfaces.Serializer` interface to implement their own serializer and deserializer, respectively. More information is provided in section XX.

- `BufferSize`: This attribute defines the buffer size that should be used by the serializer (generally only required for input signals).
- `TimeOutValue`: This is the signal timeout value in milliseconds (generally only required for input signals).

A SystemJ program and its related XML file do not necessarily need to have the same name, but the Eclipse based tools assume that they do for automatic run configurations. Once you have both files you can use the standard run profiles within Eclipse to start your application, your XML file will be parsed and used to determine the signals which, will be used to interface your system with the environment.

5.1 Custom Communication Mechanisms Using the SystemJ APIs.

In addition to the provided signal types, you can also utilise custom communication methods by implementing a simple Java class which utilises the SystemJ APIs to interface to the signals subsystem of SystemJ .

Communication via signals is split logically into a sending element and a receiving element (known respectively as Senders and Receivers). The `systemj.interfaces` package contains the Java interfaces which should be implemented in order to create one or the other. This section will briefly outline these interfaces.

Firstly, we have the `GenericSignalReceiver` interface:

```
public interface GenericSignalReceiver extends Runnable
{
    // simple member access methods
    public abstract void setDone(boolean flag);
    public abstract boolean getDone();
    public abstract Object getBuffer();
    public abstract void setBuffer(Object obj);
```

```
    public abstract int getTimeOut();

    // generic configuration method
    public abstract void configure(HashMap<String,String> data)
          throws RuntimeException;

    // standard run method to make object runnable
    public abstract void run();
}
```

As you can see, the interface contains several member access methods, which simply provide access to the internal variables that store the state of your Receiver object. A simple implementation of these methods is shown in listing 5.2.

Listing 5.2: Simple GenericSignalReceiver Implementation

```
1  pubic class SimpleReceiver implements GenericSignalReceiver
2  {
3      public void setDone(boolean flag)
4      {
5          this.done = flag;
6      }
7
8      public boolean getDone()
9      {
10         return this.done;
11     }
12
13     public Object getBuffer()
14     {
15         return this.buffer;
16     }
17
18     public void setBuffer(Object obj)
19     {
20         this.buffer = obj;
21     }
22
23     public int getTimeout()
24     {
25         return this.timeout;
```

```
26      }
27
28      // ... other methods here...
29
30      private boolean done = false;
31      private Object buffer = null;
32      private int timeout = 1000;
33  }
```

As this implementation will almost always be the same, you can simplify your implementation by utilising the `systemj.templates.ReceiverTemplate` base class, rather than directly implementing `GenericSignalReceiver`. This provides the standard methods you have seen above, whilst leaving the `run()` and `configure()` methods abstract, for you to implement.

The `configure()` method is responsible for loading the configuration parameters for the signal in question. In order to maintain a flexible infrastructure allowing arbitrary communication methods, each Sender or Receiver class is responsible for it's own configuration and must throw a RuntimeException if a required parameter is not present. Configuration parameters are passed into the `configure()` method as a Hash map containing string keys and values as strings. This maps directly to the attributes given to the corresponding `<signal>` node in the XML signal mappings. Parsing these parameters is a matter of checking for the correct keys in the map and converting and assigning the values to internal variables, e.g:

Listing 5.3: Configuration Method Example

```
1  public void configure(HashMap<String,String> data)
2     throws RuntimeException
3  {
4      if(data.containsKey(''attributename''))
5      {
6          this.myattribute = data.get(''attributename'')
7      }
8      // if the parameter is required, throw a
9      // RuntimeException when we don't have it
10     else
11     {
12         throw new
13             RuntimeException(''Couldn't find attributename!'');
14     }
15 }
```

The standard attributes; the Serializer, the TimeOutValue and the BufferSize will require parsing These can be parsed as non required, i.e., no Exception is thrown if these attributes are absent, as long as you can set a suitable default value to be used when they are not provided (most of the built in signals mark them as required).

The last method to implement is the `run()` method. This performs the actual implementation of your communication protocol. Generally this involves receiving and deserializing Java objects coming through some channel. To provide plugable serialization, SystemJ defines an API that can be used to define a Serializer class, which can be used with any signal. In the run method you can make use of this Serializer object if one was provided in the XML configuration or fall back to using standard Java serialization methods otherwise. The only other thing to note about the `run()` method is that you should call the `setBuffer()` and `setDone(true)`. The buffer should be set to a Java `ArrayList<Object>` instance containing two entries, a Boolean value representing the status of the signal (present or absent) and a Java object representing its value signal is a valued signal.

The GenericSignalSender interface is similar to the GenericSignalReceiver interface, however it does not require the myriad of member access methods, instead having just one method to perform a signal setup:

```
public interface GenericSignalSender extends Runnable
{
    // standard methods from existing implementation
    public abstract void setup(Object obj) throws Exception;

    // generic configuration method
    public abstract void configure(HashMap<String,String> data)
          throws RuntimeException;

    // standard run method
    public abstract void run();
}
```

Here the `setup()` method sets an internal buffer variable using the object passed to it and performs some setup of the underlying physical communication layer. This means that the setup method is generally reimplemented for ever Sender class and no template base class is provided. The other methods are largely the same as those documented above for the Receiver interface (with the `run()` method serializing and sending data rather than receiving it).

5.2 Custom Data Serialization

As you have seen in the previous sections, SystemJ has a mechanism for providing customised serialization of data objects, which are being communicated over signals. This is achieved by creating a class that implements the `systemj.interfaces.Serializer` interface and specifying the class name in the XML signal mappings:

```
public interface Serializer
{
    public Object deserialize(byte[] b,int length);

    public byte[] serialize(Object ob);
}
```

You can see that this simple interface provides methods for serializing a Java object to a byte array and conversely deserializing a byte array into a Java object. Several standard serializers are provided in the `systemj.serializer` package and you can quickly implement your own schemes using this API.

5.3 The SystemJ `IOLogger` Interface

SystemJ contains a standard method for logging output, which is consistent across platforms, and can be used for debugging purposes. This takes the form of the `IOLogger` interface plus an automatically selected imlementation for each platform. The `IOLogger` interface (listing 5.4) is similar to the logging interface available on Google's Android mobile Operating System.

Listing 5.4: `IOLogger` Interface

```
1  public interface IOLogger
2  {
3          // Verbose logging messages
4          public void v(String tag, String msg, Throwable tr)
5          public void v(String tag, String msg)
6
7          // Debugging messages
8          public void d(String tag, String msg, Throwable tr)
9          public void d(String tag, String msg)
10
11         // Informational (normal) messages
```

```
12          public void i(String tag, String msg, Throwable tr)
13          public void i(String tag, String msg)
14
15          // Warning messages
16          public void w(String tag, String msg, Throwable tr)
17          public void w(String tag, String msg)
18
19          // Error messages
20          public void e(String tag, String msg, Throwable tr)
21          public void e(String tag, String msg)
22  }
```

As you can see from listing 5.4 logging messages have different severities (with increasing serverity as you go down) depending on the logging level. In the future the SystemJ compiler may have the capability to compile under a certain logging level, but currenlty all messages are printed. It is also intended that a future version of the IDE tools will include a utility for filtering log messages by level or tag.

To use the logging functionality just make a call to the predefined static `Log` class, with the method corresponding to your logging level and the relevant arguments. The arguments take the form of a tag, which is usually the name of your program or specific clock domain or reaction (to aid in debugging large systems), a log message and an optional Java `Throwable` object (which will trigger the logging of the relevant JVM stack trace in order to aid debugging). Listing 5.5 gives an example of possible usage.

Listing 5.5: Usage of the `IOLogger`

```
1  system
2  {
3      interface
4      {}
5      {
6          // log a debugging message
7          Log.d("ClockDomain1", "This is a debugging message.");
8      }
9      ><
10     {
11         // log an informational message
12         Log.i("ClockDomain2",
13             "Use informational messages for normal output");
14     }
15     ><
```

```
16      {
17          // log an error message with an exception
18          Exception ex = new RuntimeException("Something is broken!");
19          Log.e("ClockDomain3", "Help!", ex);
20      }
21  }
```

5.4 Multi-threaded execution of SystemJ and binding channels via the XML interface

The SystemJ compiler also supports a multi-threaded execution approach for clock-domains. In this case every clock-domain declared using the >< operator runs as a Java thread. Compared to the first approach where all the SystemJ code is compiled into a single threaded Java program this approach is more flexible. But, be warned that such multi-threaded execution is hard if not impossible to verify. Also, this form of execution would lead to non-deterministic program behaviour.

The multi-threaded execution program generated by the SystemJ compiler is called `<name-of-file>jcsmp`. In this approach channels like signals are also implemented using an underlying socket based physical communication layer. Thus, a designer needs to provide all the information for channels in addition to signals in the XML file when running multi-threaded SystemJ programs.

An example XML file used to specify channel based communication in the multi-threaded approach is shown in listing 5.6 with the corresponding SystemJ program in listing 5.7

Listing 5.6: Channel Mapping Via XML

```
1  <SystemJProgram>
2      <ClockDomain Name="Unnamed">
3          <Outputs>
4              <Channel
5                  Name="a_o"
6                  Type="Integer"
7                  IP="130.216.25.113"
8                  Port="44453"
9                  SocketClass="systemj.signals.network.TCPSender"
10             />
11         </Outputs>
```

```
12          <Inputs>
13              <Channel
14                  Name="a_op"
15                  Type="Integer"
16                  IP="130.216.25.113"
17                  Port="44454"
18                      SocketClass="systemj.signals.network.TCPReceiver"
19                      TimeOutValue="1000"
20                  />
21              <Signal
22                  Name="start"
23                  Type="Boolean"
24                  IP="130.216.25.113"
25                  Port="44455"
26                  SocketClass="systemj.signals.network.TCPReceiver"
27                  Serializer="systemj.serializer.StringSerializer"
28                  TimeOutValue="1000"
29                  BufferSize="1"
30              />
31          </Inputs>
32      </ClockDomain>
33      <ClockDomain Name="Unnamed">
34          <Inputs>
35              <Channel
36                  Name="a_in"
37                  Type="Integer"
38                  IP="130.216.25.113"
39                  Port="44453"
40                  SocketClass="systemj.signals.network.TCPReceiver"
41                  TimeOutValue="1000"
42                      />
43              </Inputs>
44          <Outputs>
45              <Channel
46                  Name="a_inp"
47                  Type="Integer"
48                  IP="130.216.25.113"
49                  Port="44454"
50                  SocketClass="systemj.signals.network.TCPSender"
51              />
52          </Outputs>
```

```
53      </ClockDomain>
54  </SystemJProgram>
```

Listing 5.7: SystemJ Program Corresonding to Listing 5.6

```
1  system
2  {
3      interface
4      {
5          input Integer channel a;
6          output Integer channel a;
7
8          input signal start;
9          output signal done;
10     }
11     {
12         {
13             while(true)
14             {
15                 await(start);
16                 Log.i("ClockDomain1", "Hello from me!");
17                 pause;
18                 send a(new Integer (1));
19                 pause;
20             }
21         }
22         ><
23         {
24             while(true)
25             {
26                 Log.i("ClockDomain2", "Hello from me!");
27                 pause;
28                 receive a;
29                 Integer t= new Integer(0);
30                 t = (Integer)#a;
31                 Log.i("ClockDomain2", "T is " + t);
32                 pause;
33                 emit done;
34             }
35         }
36     }
37 }
```

As seen in Listings 5.7, and 5.6, every channel port defined in the SystemJ interface needs a declaration in both the `Output` and `Input` nodes of the XML description. For example, input channel a declared on line 5 in Listing 5.7 has two nodes a_in and a_inp declared within input and output nodes, respectively (Lsisting 5.6, lines 34-52). Similarly, for output channel port a declared on line 6 in Listing 5.7 we need to declare two nodes a_o and a_op in the output and input nodes of the XML description, respectively (Listing 5.6 lines 3-20).

The designer needs to follow the naming convention as shown in the XML description. Hence, for every output channel port named `<name>` the XML nodes are called `<name>_o` and `<name>_op`, while for every input channel port named `<name>` the XML nodes are called `<name>_in` and `<name>_inp`, respectively. Every `<name>_o`, `<name>_op`, `<name>_in`, and `<name>_inp` XML nodes should also have matching `IP`, and `Type` attributes as the channels perform a rendezvous. Finally, every `<name>_o`, and `<name>_in` nodes should have matching `Port` attributes, same can be said for the `<name>_op` and `<name>_inp` nodes.

Bibliography

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, R. D. Simone, The synchronous language twelve years later, in: Proceedings of the IEEE, 2003. 1.1, 2, 2.1

[2] G. Berry, The Esterel v5 Language Primer - Version 5.10, release 2.0. URL `citeseer.ist.psu.edu/article/berry98esterel.html` 1.1, 2, 2.1

[3] F. Gruian, P. Roop, Z. Slacic, I. Radojevic, The SystemJ approach to System-Level Design, in: The 4th International Conference on Formal Methods and Models for Codesign(MEMOCODE), 2006. (document), 1.1

[4] E. A. Lee, The problem with threads, Computer 39 (5) (2006) 33–42. 1.1

[5] A. Malik, SystemJ Webpage, `http://www.ece.auckland.ac.nz/~amal029` [Last Accesses: 09/11/2008] (2008). (document)

[6] A. Malik, Z. Salcic, P. Roop, Efficient Compilation of GALS Language–SystemJ, Tech. Rep. 655, University of Auckland (2006). (document)

[7] A. Malik, Z. Salcic, P. Roop, SystemJ A GALS language for Embedded Systems and its Operational Semantics, Tech. Rep. 656, University of Auckland (2006). (document)

[8] A. Malik, Z. Salcic, P. S. Roop, An Efficient Execution Platform for GALS Language SystemJ, in: The 13th IEEE Asia Pacific Computer Systems Architecture Conference, 2008. (document)

[9] A. Malik, Z. Salcic, P. S. Roop, SystemJ Compilation using the Tandem Virtual Machine Appraoch, ACM. Transactions on Design Automation of Electronic Systems (2009) (in print). (document)

[10] A. Malik, Z. Salcic, P. S. Roop, SystemJ compilation using the tandem virtual machine approach, ACM Transactions on the Design Automation of Electronic Systems (TODAES) 14 (3) (2009) 1–37. 1.1

[11] A. Malik, Z. Salcic, P. S. Roop, A. Girault, SystemJ: A GALS language for system level design, Elsevier Journal on Computer Languages. 1.1

[12] D. Potop-Butucaru, S. A. Edwards, G. Berry, Compiling Esterel, Springer Verlag, 2007. 1.1

[13] P. Welch, F. Barnes, Communicating mobile processes: introducing occam-pi, in: A. Abdallah, C. Jones, J. Sanders (eds.), 25 Years of CSP, vol. 3525 of Lecture Notes in Computer Science, Springer Verlag, 2005.
URL `http://www.cs.kent.ac.uk/pubs/2005/2162` 1.1