SystemJ: A GALS language for system level design

Article *in* Computer Languages Systems & Structures · December 2010

DOI: 10.1016/j.cl.2010.01.001 · Source: DBLP

4 authors:

Avinash Malik
University of Auckland
78 PUBLICATIONS **392** CITATIONS
SEE PROFILE

Zoran Salcic
University of Auckland
337 PUBLICATIONS **2,380** CITATIONS
SEE PROFILE

Partha Roop
University of Auckland
191 PUBLICATIONS **1,329** CITATIONS
SEE PROFILE

Alain Girault
National Institute for Research in Computer Science and Control
135 PUBLICATIONS **2,001** CITATIONS
SEE PROFILE

SystemJ: A GALS Language for System Level Design

Avinash Malik[*,1], Zoran Salcic[**,1], Partha S. Roop[1], Alain Girault[b]

[a]*University of Auckland, New Zealand*
[b]*INRIA Rhône-Alpes, Grenoble, France*

Abstract

In this paper we present the syntax, semantics, and compilation of a new system-level programming language called SystemJ. SystemJ is a multiclock language supporting the Globally Asynchronous Locally Synchronous (GALS) model of computation. The synchronous reactive (SR) model is used for synchronous parts of the modelled system, and those parts, which represent individual clock-domains, are coupled asynchronously each to the other on the top-level of system design. SystemJ is based on Java language, which is used to describe "instantaneous" data transformations. Hence, SystemJ is well suited for both software-based embedded and distributed systems. SystemJ offers effective modelling of (1) data transformations through the power of Java, (2) control and synchronous concurrency through the SR paradigm and (3) asynchronous concurrency through clock domains and rendezvous. The language is based on semantics that is amenable to efficient code generation and partial automatic verification. The SystemJ micro-step semantics provide asynchronous and synchronous extensions over the semantics of other SR languages such as Esterel and provide an ideal platform for efficient software implementation.

1. Introduction and Overview

Hardware and software systems are increasingly becoming multiclock either due to the IP-based design paradigm [1] or due to their distributed nature [2]. Hence, there has been a recent surge in demand for tools that can be used for modelling, validation, and synthesis of such multiclock systems. Of particular interest has been a sub-class of multiclock systems, called *Globally Asynchronous Locally Synchronous* (GALS) systems [3, 4], which occur naturally in system-on-chips (SoC) and many other distributed systems.

System level design languages, including languages targeting multiclock design, have existed for some time. All current system level design languages can be classified into two separate domains, formal and informal. Formal languages like Esterel, CRP, SHIM and CRSM [5, 6, 7, 6] are based on rigorous mathematical foundations and are thus suitable for formal verification and compilation. Languages like SystemC [8] lack formal semantics and thus are harder

*Principal corresponding author
**Corresponding author
Email addresses: `amal029@aucklanduni.ac.nz` (Avinash Malik), `z.salcic@auckland.ac.nz` (Zoran Salcic), `p.roop@auckland.ac.nz` (Partha S. Roop), `alain.girault@inria.fr` (Alain Girault)

to compile and difficult to verify. The authors are of the opinion that formal semantics should be an integral part of any system level design language and hence comparison with informal languages is avoided.

The CRP language [9] combines the synchronous reactive Model of Computation (MoC) of Esterel [5] with the asynchronous coupling of CSP [10]. While being mathematically elegant, CRP never became a successful asynchronous language due to its reliance on Esterel (which has poor support for data-driven operations). Moreover, it offered an inefficient implementation of the rendezvous protocol through asynchronous coordinators. Later variants such as CRSM [9] and ECRSM [11] also inherited the same deficiencies of CRP. In addition to these, a multiclock extension to standard Esterel was proposed in [12] to offer the asynchronous semantics of VHDL within Esterel. This, however, made the language rather complicated (due to loss of abstraction) and difficult to use.

More recently, the multiclock Esterel (MC-Esterel) language has been developed by [13] targeting the design of digital hardware. The main idea was to keep the synchronous elegance of Esterel so that existing tools such as the Esterel prover (which performs observer-based verification using the synchronous semantics) could be used with the multiclock language. In this language, every Esterel module needs an explicit clock and the designer has to provide the design of low-level synchronizers to synchronize these *clocked-modules*. A multiclock program can be compiled into a single synchronous program using the *weak suspend* statement (which performs clock gating [14]). While MC-Esterel provides a powerful mechanism for modelling asynchronous and multi-rate systems, the main drawback lies in the design abstraction used; it is too low-level and the designer has to know hardware design quite well to be able to design synchronizers properly.

Like MC-Esterel, SHIM [15, 7] is another recent language for asynchronous system design. Here, the main idea is to decompose an embedded system into hardware and software functions using manual partitioning. Subsequently, both hardware and software functions are written as C-like functions and the communication between these asynchronous objects is achieved through rendezvous protocol that is mapped to a restricted Kahn Process Network (KPN) [16]. SHIM, thus, provides very high-level design abstraction for the designer. However, the major limitations of SHIM are the lack of support for modelling reactivity and its need for early manual hardware-software partitioning that prevents later design space exploration. SHIM also does not provide language based support for describing synchronous threads/processes and hence, designing GALS or synchronous systems in SHIM is much more involved compared to SystemJ.

In this paper we present the SystemJ language, its operational semantics and a new compilation approach, as a major step towards the design of a tool for specification, validation, and synthesis of software based GALS systems. Our main objectives in designing SystemJ were to meet criteria, which we consider essential but lacking in the current system-level design space. We elaborate these here:

1. *Need for a system-level design language capable of modelling multiple clock systems, particularly those that can be modelled by a* GALS *MoC*: SystemJ, unlike many current system-level languages is not targeted towards embedded systems alone, but is designed to be a general purpose system-level language. Like other languages, including CRP, CRSM and MC-

Esterel, it is capable of modelling systems using both pure *Synchronous Reactive* (SR), and GALS MoC. But, unlike these languages (especially MC-Esterel), the compilation of SystemJ is targeted towards generating efficient and portable software, in this case Java code. The resultant Java code is capable of executing on a range of different computing platforms. With Java as its compilation result, SystemJ is much more portable compared to any current system-level language. The portability of SystemJ is an important property as it allows designers to design SystemJ models and systems on a desktop and then perform automated deployment on other target architectures, like embedded systems, without the need for recompilation.

2. *Need for a system-level design language capable of modelling complex data-driven operations using high-level data abstractions*: Most of the system level languages currently provide minimal support for data-driven operations based on C. Also, many of these languages (including Esterel and MC-Esterel) separate control and data-driven computations, which is not the natural coding style for developers. SystemJ overcomes this limitation by extending (and embedding) Java itself with control (reactive and concurrency) statements, which drive the control flow. Such tight integration of control and data-driven operations reduces the source code size and provides a much more powerful abstraction for describing large and complex models.
3. *Suitability for automated formal verification:* System level languages are used for designing large and complex systems. Many of these systems are used in mission critical environments where incorrect system behaviour would result in undesired consequences. The current approach to designing such systems requires manually validating and debugging the designed system. This approach besides being tedious also does not guarantee complete state space exploration of the designed system, which depends upon the test scenarios the system is exposed to. Automatic formal verification on the other hand guarantees complete state space exploration of the designed system, which helps in ironing out all the bugs and guaranteeing correct system behaviour. The approach of formally verifying the system also helps in greatly reducing the designer productivity gap as the validation phase is completely automated, thereby freeing the designers to carry out other tasks.
4. *Appeal to software developers*: The syntax and semantics, which are more suitable for designing digital circuits, of the current system-level languages is the major reason for the slow adoption rate of these languages by software developers. SystemJ is designed with the aim to overcome this barrier. By using a syntax and design flow similar to general purpose programming languages like Java and C++.

An early version of SystemJ was presented in [17] together with its Java multithreaded library (TReK), which implements GALS MoC. This approach to translating and implementing a GALS model, besides lacking a formal semantical foundation, also suffered from a number of other drawbacks. The high level Java-based implementation utilizing Java-threads, templates, and Java-exception mechanisms introduce large run-time overheads, thereby making SystemJ implementation unsuitable for resource constrained embedded systems.

The Java-threading model also introduces non-determinism, usually due to the reliance on OS threading support, not intended by the designer. This in turn makes verifying SystemJ models harder [18].

The new version of SystemJ presented in this paper includes further extensions to the language, particularly those supporting strong preemptions at the asynchronous level. The proposed semantics lay down the foundation for a portable compilation approach targeting various processors. The focus of this paper is the semantics of the SystemJ language, which enables us to design an effective intermediate format for compilation and further back-end generation of executable Java code.

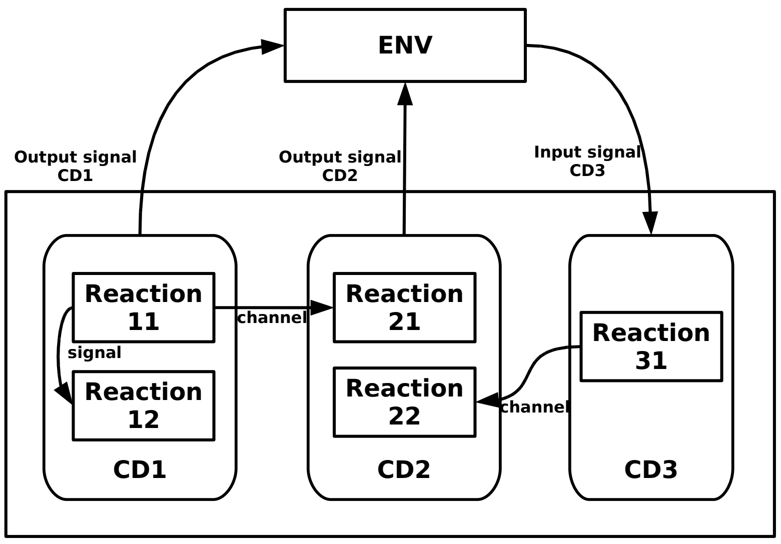
In short, a SystemJ program is a collection of asynchronous nodes called clock-domains (a GALS system) where each clock-domain is a purely synchronous node analogous to a SR program in a language like Esterel. Clocks are not modelled explicitly, unlike in MC-Esterel. Inter clock-domain communication is achieved through the asynchronous *send/receive* statements over point-to-point channels. SystemJ, thus, combines the Esterel-like reactivity with the asynchronous coupling of CSP and hence looks superficially similar to CRP [9]. However, unlike CRP, it offers powerful modelling of data-driven operations through Java objects and associated methods. In addition, rendezvous in SystemJ is robust, with clean semantics and with a simpler scheduling model compared to CRP and CRSM [6]. Finally, the micro-step semantics of SystemJ enables us to propose a new intermediate format called Asynchronous Graph Code (AGRC), which directly captures this operational semantics in a graph format.

This paper is organized as follows. Section 2 gives an informal introduction to the SystemJ programming model. Section 3 gives an example system designed with SystemJ for the reader to get familiar with the syntax of the language and also to highlight the strengths of SystemJ. Section 4 formalizes the most basic kernel syntax of SystemJ. Section 5 provides an informal presentation of the semantics of the language, including synchrony and asynchrony. Section 6 and Appendix A gives a formal presentation of the SystemJ semantics. Section 7 introduces the AGRC intermediate format for compiling SystemJ. Section 8 gives the details on the back-end compilation from AGRC intermediate format to Java. Section 9 gives quantitative results of comparing the AGRC compiler with the previous TReK based approach. Section 10 gives a detailed comparison of SystemJ with other system-level languages and reactive Java packages. Lastly, we end the paper with the conclusions and future work (Section 11).

2. The SystemJ programming model

A SystemJ program consists of a set of clock-domains executing at unrelated logical clock ticks (from here on referred to as tick), coupled together with the asynchronous parallel operator (><), and synchronizing and communicating each with the other using channels. SystemJ uses CSP [10] style rendezvous for synchronization and data transfer between clock-domains. Each clock-domain itself consists of one or more processes called reactions executing in lockstep, i.e., at the clock-domain's tick. The reactions are combined and controlled using the synchronous parallel operator (||). Reactions within the same clock-domain communicate using the synchronous broadcast mechanism over signals. Finally, a SystemJ program interacts with its environment through a set of input and output signals and operations on these signals. The synchronous, reactions and operations on signals, and asynchronous, clock-domains and channels, statements are together responsible for the control-flow of SystemJ programs. The data-driven computations and transformations are performed in Java.

Figure 1: A general representation of a SystemJ example



2.1. System

A top level SystemJ program is called a system; it consists of a collection of clock-domains composed with the asynchronous parallel operator (><). Within a system, each clock-domain executes at its own tick, and any two clock-domain ticks are unrelated. The system is used to declare clock-domains, input/output signals that are used for communicating with the environment, and channels for communicating between clock-domains. Figure 1 gives the pictorial representation of an example SystemJ system with three clock-domains CD1, CD2, and CD3.

2.2. Clock-domain

A clock-domain is a reaction or a collection of reactions composed using the synchronous parallel operator (||) executing in lockstep. Two clock-domains capable of synchronizing and communicating with each other are called *partner clock-domains*. A clock-domain can be either named or unnamed. Each clock-domain has its own set of signals and channels, which it uses to communicate with the environment and with other clock-domains, respectively. Within clock-domain CD1 Reaction11 and Reaction12 are combined using the synchronous parallel operator. Reactions within a clock-domain communicate using signals (the same as in Esterel [19]), while cross clock-domain communication requires channels.

2.3. Reaction

A reaction is a synchronous process consisting of reactive statements operating over signals and Java statements describing data computations and transformations. Reactions are composed into clock-domains using the synchronous parallel operator (||) and run synchronously, at the same tick. Reactions can also be used to implement behavioural hierarchy within a clock-domain, thus supporting the decomposition of the clock-domain functionality into smaller and manageable parts.

2.4. Channels

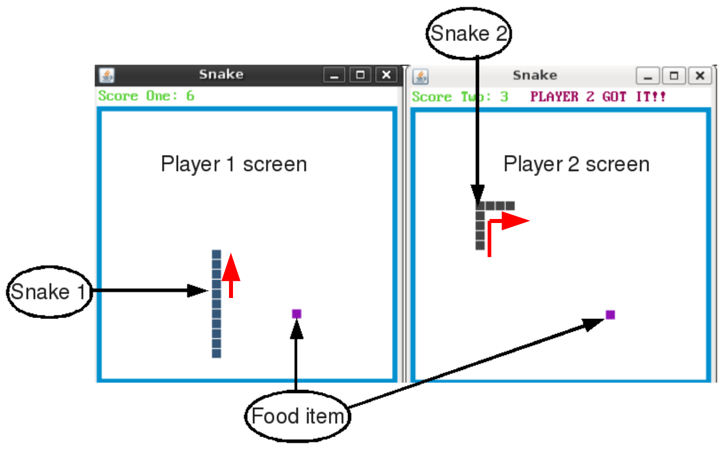
Channels are the only means of communication between clock-domains. Channels enable communication between reactions in different clock-domains. For example, in Figure 1 channels are used for communicating between Reaction 11 and Reaction 21 in clock-domains CD1 and CD2, respectively. They are used to synchronize, using a CSP style rendezvous, and to exchange data once the rendezvous is performed. Channels are fixed, point-to-point and unidirectional. The two rendezvous primitives are `send` and `receive`.

3. SystemJ Example

In this section we present a feature rich SystemJ example. The example is our own version of the game of snake. Although inspired from the classical snake game (available on cell-phones), the presented version is for multiple contesting players, and far superior in terms of player interactivity. The game consists of multiple players each able to control his/her own snake. The objective of a player is to compete with other players for food, while clearing various mazes at different stages of the game (reminiscent of the old arcade style gaming). This example for brevity presents a two-player snake game. SystemJ allows easy extension of this game to any number of players as we will see in the following paragraphs.

Figure 2 shows a screen shot of the two-player snake game, where, for simplicity, the mazes are the horizontal and vertical walls only. Each player is only able to view his/her own screen. The graphics, including the position of the food grain and mazes on each players screen, is identical, but the position and the size of the snake differ (as each snake is controlled individually by separate players). The arrows show the current movement and direction of the individual snakes and are super imposed on the original screen shot. Players earn a point by eating a grain of food, and at the same time each food grain increases the size of the snake by one. Bonus points are available for players if they are able to eat special food items. The appearance of special food items is in turn controlled by timers, unlike normal food grains, which are present on the screen until eaten; these special food items do not affect the size of the snake. At any instant of time, there can be at most one grain of normal and one grain of special food item on the screen. New food items appear randomly on the screen of each player. A player looses his/her life if the snake hits the maze (walls) or if the snake collides onto itself. Mazes become harder to navigate as the

Figure 2: Screen shot of the multi-player snake game



stages progress. The last player standing or the player with highest points after 5 stages is the winner. Individual snakes can be controlled using arrow keys or can be mapped to other keys on the keyboard as required by the players.

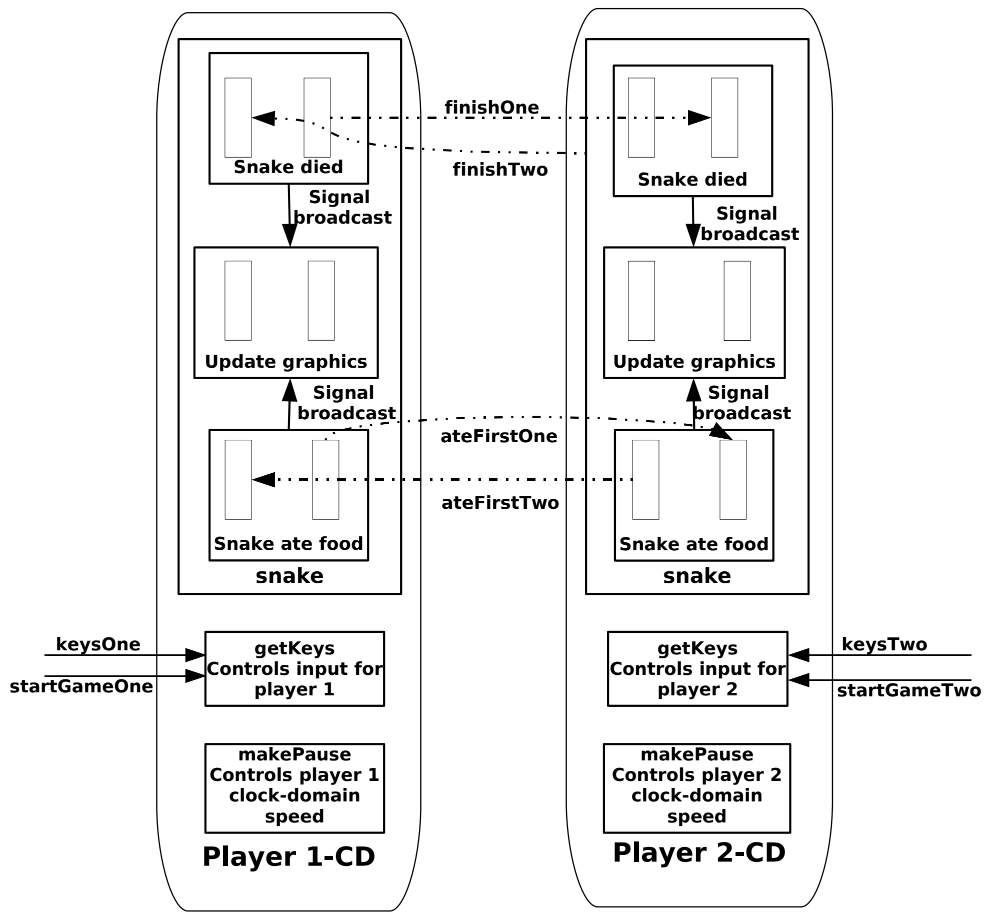
This example illustrates one of the most important features of SystemJ, its ability to appeal to software developers. SystemJ's intuitive design approach using reactive control flow statements makes it easier for programmers to describe control-flow in complex systems, compared to general purpose languages like Java or C/C++ alone.

Figure 3 gives an abstract representation of the structure of the SystemJ system used to implement our two player snake game. Each player's snake is controlled using a separate clock-domain. Figure 3 shows top three synchronous concurrent reactions. The top-most reaction "snake" is further composed of three main synchronous parallel reactions. The top-most reaction "snake died" within reaction snake is responsible for communicating the current alive or dead status of the snake. If the snake dies, the reaction communicates this fact to the partner clock-domain utilizing a channel (dotted lines). It also broadcasts a signal (solid lines) to the "update graphics" reaction, which is responsible for updating the graphics on the screen.

The bottom reaction "snake ate food" is responsible for maintaining a status of food grains and score for individual players. Since the food grains are shared amongst players, if player one eats the food grain, then this fact needs to be immediately communicated to the partner clock-domain in order to update the graphics on screen. This reaction also broadcasts the signal to its own "update graphics" reaction. Other reactions include "getKeys" and "makePause", which are responsible for controlling the movement of the snake and the speed of the game, respectively. These reactions achieve their objectives by broadcasting signals to the "snake" reaction.

Listings 1 and 2 show parts of the actual SystemJ code of the described system. For ease of understanding the complete system has been highly abstracted.

Figure 3: Pictorial representation of the SystemJ "system" for the two player snake game



First, we have the top-most "system", which is used to declare the signals and channels needed for communication (**interface**). Within the system the top level asynchronous clock-domains are also initialized. In this system, we have two clock-domains, one for each player. Adding more players is just a matter of declaring more channels in the interface as needed and instantiating more clock-domains with appropriate code for synchronizing the clock-domains using `send` and `receive`. The code reuse and the ease with which clock-domains can be added using asynchronous operator (><) makes SystemJ a highly scalable language. Each clock-domain uses a separate set of keys to interact with the environment (clock-domain 1 uses signal keysOne, while 2 uses keysTwo).

Listing 1: Top-system two player game specification

```

/**The top level system reaction which initialises the various asynchronous
   clock-domains. Also all the channels and signals needed to communicate with
   environment are all initialised in the interface
**/
system
{
 interface
 {
   int channel finishOne; //channel that indicates that the first snake died
   int channel finishTwo; //channel that indicates that the second snake died
   int channel beatOne; //channel that indicates that the first snake got beat
   int channel beatTwo; //channel that indicates that the second snake got beat
   int channel ateFirstOne; //channel that inidicates that the first snake ate food
   int channel ateFirstTwo; //channel that indicates that the second snake ate food
   int signal keysOne, startGameOne; //We use an int typed valued signal to represent the keys.
   int signal keysTwo, startGameTwo;
 }
 {
  {
  //Clock-domain 1 representing Player-1
    getKeys(keysOne, startGameOne) //used to communicate
                                   //with environment
    ||
    snake(keysOne, startGameOne, finishOne, finishTwo,
    beatOne, beatTwo, ateFirstOne, ateFirstTwo) //The main game controller
    ||
    makePause() //The reaction controlling the speed of the game using timer
  }
  ><
  {
  //Clock-domain 2 representing Player-2
    getKeys(keysTwo, startGameTwo)
    ||
    snake(keysTwo, startGameTwo, finishOne, finishTwo,
    beatOne, beatTwo, ateFirstOne, ateFirstTwo)
    ||
    makePause()
  }
 }
}
```

Listing 2 shows the main reaction responsible for controlling the graphics and the complete game play. There are two main things to note:

1. The tight integration of Java data-driven operations with complex control-flow. For example, in Listing 2 line 10 the signal `dirsig` is declared to be of type `Direction`, where `Direction` is a user defined Java class. This allows broadcast of the `Direction` class within the clock-domain using the `emit` statement.
2. The ease of use, especially with regards to the communication between partner clock-domains using `send`/`receive` over channels.

The listing below is well commented for ease of understanding and reflects the system described in Figure 3.

Listing 2: snake-reaction

```

/**This reaction acts as the main controller of the game.
   Many of the synchronous parallel reactions called by the snake reaction are
   not shown for brevity.
**/
reaction snake(:input int signal keys, input int signal startGameOne, int channel finishOne,
 int channel finishTwo, int channel beatOne, int channel beatTwo,
 int channel ateFirstOne, int channel ateFirstTwo){
 signal dead, newlevel, atefood, makebonus, atebonus, oneWon;
 Direction signal dirsig;
 int dieOne = 0; int winOne = 0; int twoDied = 0; int twoWins = 0; int finTwo = 0;

 Draw draw = null; draw = new Draw();
 draw.print(Draw.X/2−5*21, Draw.Y/2−5*6, "PRESS␣'ENTER'␣TO␣START␣GAME", 0x654321);
 await(startGameOne);

 draw.drawRect(0, 0, Draw.X, Draw.Y, Draw.BACKGROUND_COLOUR);
 draw.drawBorder();
 draw.drawInitialSnake();
 draw.repaint();

 trap(t)
 {
  draw.print(2, 2, "Score␣One:", 0x56D034);
  {
   while(true)
   {
    updateSnake(draw, dirsig, oneWon, dead, atefood)
    ||
    yay(draw, atefood)
    ||
    {
     while(true){
      //Awaiting until the atefood signal is received.
      //Once the signal is received we send this information to
      //the partner clock−domain using send and channel ateFirstOne.
      await(atefood); send ateFirstOne(1);
     }
    }
   }
  }
 ||
 keepScore(draw, atefood) //This is an example of named reaction
 ||
 {
  //The snake−ate food reaction.
  while(true){
   receive ateFirstTwo; draw.drawFoodTwo();
  }
 }
 ||
 {
  //Snake−died reaction
  //We receive the information if snake two is dead
  //by utilizing receive on channel finishTwo.
  receive finishTwo; twoDied = 1; exit(t);
 }
 ||
 {
 //Reaction used to decide which one is the winner
 //by looking at the score of each player, exit game if over.
  receive beatOne; finTwo = #beatOne;
  if (finTwo == 1){
   twoWins = 1; exit(t);
  }
 }
 ||
 {
   //Awaiting to receive the dead broadcast signal.
   //Once the snake dies we communicate this fact
   //to the partner clock−domain using send on channel finishOne
   await(dead); dieOne = 1; send finishOne(1); exit(t);
 }
 ||
 {
 //reaction to decide the winner and send the player score
 //also exit the system once winner is decided
  await(oneWon); winOne = 1; send beatTwo(1); exit(t);
 }
 ||
 processKeys(keys, dirsig)
 }
 //Updating graphics on screen.
 if( dieOne == 1){
  draw.drawRect(0, 0, Draw.X, Draw.Y, Draw.BACKGROUND_COLOUR);
  draw.print(Draw.X/2−5*18, Draw.Y/2−5*8, "PLAYER_ONE_KILLED_HIMSELF", 0x654321);
 }
 if (twoDied == 1){
  draw.drawRect(0, 0, Draw.X, Draw.Y, Draw.BACKGROUND_COLOUR);
  draw.print(Draw.X/2−5*12, Draw.Y/2−5*8, "PLAYER_ONE_WINS", 0x654321);
 }
 if (winOne == 1){
  draw.drawRect(0, 0, Draw.X, Draw.Y, Draw.BACKGROUND_COLOUR);
  draw.print(Draw.X/2−5*12, Draw.Y/2−5*8, "PLAYER_ONE_WINS", 0x654321);
 }
 if (twoWins == 1){
  draw.drawRect(0, 0, Draw.X, Draw.Y, Draw.BACKGROUND_COLOUR);
  draw.print(Draw.X/2−5*8, Draw.Y/2−5*8, "YOU_LOSE", 0x654321);
 }
}
```

The snake game shows the power of the SystemJ language in designing complex systems. The most obvious advantages of using SystemJ is the tight integration of data-driven and control-driven statements and synchronous and asynchronous concurrency. For example, the programmer just needs to compose the reactions with the || operator in order to make them run concurrently, as compared to using Java where the programmer needs to have detailed knowledge about the synchronization mechanisms between Java threads. Although Java threads inherently give the programmer fine grained control, the designed programs are prone to problems like deadlocks and race conditions. The SystemJ compiler, on the other hand, compiles away this parallelism at the back-end to produce single threaded Java code ready to be executed on any Java virtual machine (JVM). Thus, when using SystemJ a designer needs to concentrate only on the problem at hand leaving the critical implementation details to the compiler.

4. Kernel SystemJ syntax

SystemJ statements can be divided into two parts, the kernel statements and the derived statements. The kernel statements and operators are listed in Table 1. These kernel syntactic statements can be combined together to form more complex derived statements, which provide a further level of abstraction and syntactic sugar to reduce code size and programming effort.

Table 1: SystemJ kernel statements and their description

(a) SystemJ synchronous kernel statements and their description

SystemJ synchronous kernel statements	**Description**
`;`	dummy statement
`pause`	dummy statement
`[`*output*`] [`*input*`] [`*type*`] Signal S`	signal declaration
`emit S(exp)`	signal emission
`p1;p2`	sequential statement
`while(true) {p}`	infinite loop
`present(S) {p1} else {p2}`	conditional statement
`[weak] abort([immediate] S) {p}`	preempt watchdog
`[weak] suspend([immediate] S) {p}`	halt watchdog
`trap(T) {p}`	trap exception mechanism
`exit T`	exit from trap
`p1 || p2`	parallel statement
`jterm(p)`	Java data-driven computation
`jterm(p) = #S`	obtaining the signal value
`jterm(p) = #C`	obtaining a channel value

(b) SystemJ asynchronous kernel statements and their description

SystemJ asynchronous kernel statements	**Description**
`[`*type*`] channel C`	channel declaration
`p1 >< p2`	asynchronous parallel
`send C([exp])`	send data on channel
`receive C`	receive data on channel

The SystemJ kernel statements can be further divided into two sets, the synchronous statements, which represent the SR MoC and the asynchronous statements, which deal with the asynchronous MoC. The synchronous statements are shown in Table 1(a).

The `;` is a dummy statement that performs no operation. The `pause` statement also does not perform any operation but consumes one tick. The `signal` statement is used to declare signals in SystemJ. This statement can be prefixed by a ***type***, which can be any Java ***type***, to declare a valued signal. Further qualification of a signal declaration statement with either an ***output*** or an ***input*** can be used by the designer to declare signal as an interface signal for communication with the environment. Signals in SystemJ have a status and possibly a value. Signal statuses can be in three possible states, `unknown` ($\perp$), `true` ($+$) or `false` ($-$), following ternary logic [20]. The signal values can be any Java type. The `emit` statement sets the signal status to `true` and sets the value of the signal if any. Once emitted the signal status remains `true` until the end of the current tick, it gets reset to unknown at the start of the next tick, this is called signal *reincarnation.* The emitted signal value is persistent over ticks and is changed only when the same signal is emitted with a different signal value. SystemJ allows multiple emissions of the same signal in the same tick. SystemJ provides a *hook* to combine multiple signal values for a signal. The designer is required to provide an associative and commutative combinator function attached to this *hook*, which is called to set the value of the emitted signal at every emission.

Combining two synchronous reactive statements with the `;` operator makes them execute sequentially. SystemJ provides three exception mechanisms, the first two, `abort` and `suspend`, are based on signals, while the last one, `trap`, is similar to the Java's `try-catch` exception mechanism. The `abort` statement acts as a watchdog. The `abort` statement preempts a program if the status of the watchdog signal is `true` in any given tick. The language provides various types of `abort` statements. A `weak abort` statement allows the program it encapsulates to finish a single tick before preempting even in the presence of the watchdog signal. A strong `abort` immediately terminates the program if signal status is `true`. A `weak` or strong `abort` can be further combined with an `immediate` signal predicate. An `immediate abort` starts watching the signal from the very first tick, while a non `immediate abort` always watches the signal after the first tick has lapsed. The `suspend` statement is similar to the abort statement except, instead of preempting its body this statement pauses in the presence of signal `S`. The `trap` statement is a control flow based preemption statement similar to those found in modern imperative languages like Java/C++. The program execution terminates immediately once the corresponding `exit` statement is executed in the `trap` body.

SystemJ uses the Java `while` statement to provide an infinite looping statement. The synchronous parallel operator `||` runs two or more reactions concurrently and in lockstep. All reactions forked by the `||` operator start and finish together. The semantics of the `||` operator are described in detail in Section 5. Finally, the synchronous statements also consist of the `jterm(p)` statements, which can be any Java statement used for data-driven computations. Unlike Esterel, data-driven computation can be freely mixed with reactive control flow thereby providing a much more powerful programming abstraction.

The asynchronous statements presented in Table 1(b) are specific to SystemJ and represent the asynchronous MoC implemented in SystemJ. The first kernel statement is used to declare a `channel`, which is used to send data across clock-domains from the ***output port*** of the channel to a corresponding ***input port*** of the channel. These ***output*** and ***input*** ports of channels are intialized internally at channel declaration. An ***output port*** is used by the `send` statement to send data, while the ***input*** port is used by the `receive` statement to receive data. The `send` and `receive` statements are used to synchronize and send data over channels. SystemJ uses rendezvous for synchronization between sending and receiving reactions, within partner clock-domains, which communicate over a channel. Rendezvous in SystemJ is closely related to the scheduling of clock-domains and the asynchronous operator operator (`><`). The intuitive semantics of rendezvous and the `><` operator are presented in Section 5.

5. Intuitive SystemJ semantics

This section presents the intuitive semantics of the synchronous and asynchronous SystemJ statements presented in Table 1 and other semantic issues that arise when compiling SystemJ programs. A formal treatment of the SystemJ language is presented in the next section.

5.1. Intuitive synchronous semantics

A SystemJ clock-domain, which is a composition of one or more reactions, follows the synchronous MoC. A clock-domain samples the input signals from the environment at the start of its tick, all synchronous parallel reactions within the clock-domain *react* simultaneously and instantaneously computing and emitting their outputs in *zero time*, and then quiescent until the next tick. Thus, each clock-domain behaves like a classical *Finite State Machine* (FSM).

A clock-domain in SystemJ is deterministic and causal. Let p be a synchronous reactive term or a combination of such terms. We define determinism and causality over p.

Definition 1. *In any tick, given a set of sampled input signal events, p always produces the same set of output signals.*

Definition 2. *p is causal iff elementary computations of p can be totally ordered in* all *ticks.*

5.1.1. The `pause` and signal based statements

The `pause` statement indicates the tick boundary and is the only statement that consumes time (one logical tick), while all other statements complete instantaneously. Signals in SystemJ are composed of a status and possibly a value, where statuses follow ternary logic, while value can be of any Java data type. Depending upon the signal declaration, the signal can be a local signal or an environment interface signal. Local signals are used for communication between reactions within a clock-domain, while interface signals are used for communication with the environment.

```
signal S; //pure local signal (only has a status)
Image signal S; // Image type local signal,
//where Image is a user defined Java class
input int signal S; //a valued input signal
//from environment of type integer
output signal S; //pure output signal to the environment
```

The example code above shows different types of signals and their declarations. At declaration the signal status is set to `unknown` ($\perp$) and the value is initialized to `null`. A signal can be emitted using the `emit` statement and a signal can be checked for presence using the `present` statement. An emission of the signal, sets the signal status to `true` ($+$). The emission of the signal makes its status visible in all the reactions within a clock-domain, i.e., it is broadcasted. The status of the signal is set to `false` ($-$) at the end of the tick if all the emissions of the signals are ruled out by control-flow. The `emit` statement can also be used to emit a signal value. The sample code below shows the signal emission and presence check.

```
emit S; //emitting a pure signal
emit S(2); //emitting a signal with value 2
emit S(Image.img); //emitting an image type signal
present(S){p} else {q} //where p and q are
//synchronous reactive programs
present(S || S1){ p } // checking a signal expression
```

SystemJ allows emissions of the same signal more then once in the same tick. In case of valued signal emission, the values of the signal need to be combined using an associative and commutative function. The associative and commutative function guarantees that the order of emission of the valued signal does not effect the resultant value and hence, the determinism of a clock-domain. Consider the example code below. In this example the signal `S` is emitted twice from the first two concurrent reactions. The third concurrent reaction waits for the emission of signal `S` and then emits the value of signal `S` in signal `O`.

```
{ emit S(2);}  //reaction 1
||
{ emit S(3);} //reaction 2
||
{ //reaction 3
 await (S);
 emit O(#S); //The # operator reads the signal value
}
```

If the designer provides a commutative and associative function like addition to combine the emitted values 2 and 3 then signal `O` always emits 5, else, if the combinator function is subtraction then depending upon the emission order of signal `S`, `O` can emit either 1 or -1. The non-commutative and non-associative combinator function breaks the deterministic property of the clock-domain. This combinator function is a hook attached to every signal, which needs to be provided by the designer. More information about the combinator hook is provided in the compilation section, Section 8. The `abort` and `suspend` statements are used for signal based preemptions. Their behaviour has been described in Section 4. A statement like the `present` statement can check for the status of the signal expression, which consists of signal statuses combined using the Java logical operators.

5.1.2. The `while` looping statement

The `while(true)` statement in SystemJ is used to implement an infinite loop. Any loop encapsulating synchronous reactive statements is required to consume a single tick with every iteration. Such a behaviour is needed to avoid schizophrenic signal behaviour. Consider the incorrect SystemJ program below. In this program the control point enters the loop and declares signal `S`, which initializes its status to unknown. Next, signal `S` is emitted setting its status to `true`. After emission of signal `S`, the loop reiterates thereby redeclaring signal `S` again. Thus, in this case the status of signal `S` is schizophrenic, i.e., it is `unknown` and `true` at the same time.

```
while(true){
 signal S;
 emit S;
}
```

The example below shows the modified and correct SystemJ program. A `pause` statement needs to be inserted at the end of the loop. In the example below the signal declaration and its emission happen in different ticks and hence, the SystemJ program is correct.

```
while(true){
 signal S;
 emit S;
 pause;
}
```

5.1.3. The `trap` and `exit` preemption statements

The `trap` and `exit` statements work in pair. They together implement exception mechanism in SystemJ. The `trap-exit` exception mechanism differs from the normal Java `try-catch` statement. Consider the example program below. A `trap` statement immediately preempts its enclosing body if the corresponding `exit` statement is executed and continues processing after the ending of the `trap` scope. If the `trap` block does not execute the `exit` statement then the `trap` statement finishes processing the enclosing block and continues forward. In the example below if signal `I` is present then the higher priority trap `T1` is preempted and only signal `Q` is emitted. If signal `I1` is present then trap `T2` is preempted and signals `S` and `Q` are emitted. Otherwise all signals `O`, `S` and `Q` are emitted.

```
trap(T1){ //higher priority trap
 trap(T2){ //lower priority trap
 present(I)
  exit(T1);
 else present(I1)
  exit(T2);
 emit O;
 } //Trap T2 scope ends
 emit S;
} //Trap T1 scope ends
emit Q;
```

5.1.4. The synchronous parallel operator $\|$

The synchronous parallel operator $\|$ composes reactions together to run in parallel and in lockstep with the tick of the clock-domain. In the example program below, the clock-domain forks three synchronous parallel reactions and blocks for their completion before proceeding further. Termination codes, which can be attributed to Berry [19], are an easy way to synchronize the concurrent reactions and guarantee that all branches of the $\|$ statement will terminate, pause or exit some trap condition synchronously. Each reaction and the clock-domain finish with a termination code. A termination code is an integer value with an interval of [0,1,2,..,∞]. Here, a termination code of 0 represents completion of a reaction or clock-domain. A termination code of 1 represents that the reaction has paused. A termination code in the interval [2,3..,∞) is reserved for `trap` statements, where higher priority traps are assigned higher integer termination code. Finally, the termination code of ∞ shows unresolved signal dependencies and is used for cyclically scheduling synchronous parallel reactions [21]. If reaction $p1$ completes with a termination code of 'm' and $p2$ with 'n' then the composition $p1\|p2$ completes with a termination code of $max(m, n)$.

```
trap(T1){ //higher priority trap
 trap(T2){ //lower priority trap
  { //reaction-1
   present(S)
    exit(T2);
  }
  ||
  { //reaction-2
   emit S;
  }
  ||
  { //reaction-3
   exit(T1);
  }
  emit L;
 }
 emit N;
}
```

In the example above, a local signal `S` is checked for presence in the first synchronous parallel reaction and it is emitted from the second reaction. According to synchrony hypothesis all synchronous parallel reactions *react* concurrently and instantaneously. Thus, we can check for a signal presence only after we have guaranteed that no more emissions of that signal are possible from sibling synchronous parallel reactions. Hence, reaction-1 is said to have a signal dependency from reaction-2. The scheduling order of synchronous parallel reactions should not affect the deterministic behaviour of the system. We utilize the cyclic scheduling algorithm in [21] to guarantee such behaviour.

If during execution reaction-1 is scheduled before reaction-2, then reaction-1 finishes with a termination code of ∞. The second reaction emits signal `S` and finishes with a termination code of 0, while the third reaction executes the `exit` statement and finishes with a termination code of 3 (note `T1` is a higher priority trap statement). Next, the synchronizer calculates the maximum termination code, in this case ∞, and, thus, decides to rerun the first reaction as it finished with a termination code of ∞. In this iteration, the first synchronous parallel reaction checks signal `S` for presence, executes the `exit` statement and finishes with a termination code of 2. The synchronizer again calculates the maximum of the termination codes ($max(2, 0, 3)$) and signal `N` is emitted. Please note that the synchronous parallel reactions are allowed be inter-leaved in parallel. It is the synchronizer that guarantees that the resultant composition proceeds in locksteps.

The two further semantic rules, which one needs to consider when writing synchronous parallel reactions and Java data-driven computations deal with read-write and write-write concurrency, which we elaborate upon here.

1. *write-write concurrency of Java data-drive computations:* Write-write concurrency of Java data-driven computations is prohibited in SystemJ. Consider the program below.

   ```
   v = 1; || v = 2; || emit S(v);
   ```

   The first two synchronous parallel reactions update the Java variable `v`. The third reaction emits a signal `S` with the value v. Depending upon the scheduling order of reactions different values of signal `S` (1 or 2) can be emitted by the third reaction. This violates determinism as defined in Definition 1. The compiler determines such SystemJ programs as incorrect.
2. *read-write concurrency of Java data-driven computations:* Read-write concurrency is allowed in SystemJ. A single writer reaction and multiple readers are also allowed. Read-write concurrency is handled in the SystemJ compiler in the same way as signal dependencies. In fact, read-write concurrency leads to data-dependencies and cyclic scheduling described previously is utilized to resolve such dependencies. An example of read-write concurrency is shown in the example below.

   ```
   present(S) v= 2; else v = 3; || emit O(v);
   ```

   In the example above the first synchronous parallel reaction sets the value of `v` to either 2 or 3 in mutually exclusive branches. Thus, if signal `S` is present signal `O` is emitted with a value of 2 else, it is emitted with a value of 3.

It should be pointed out that write-write and read-write concurrency can also be created by multiple emissions of valued signals. Such programs are considered correct because the write-write concurrency of valued signal emissions are resolved using associative and commutative combinator functions. The read-write concurrency is resolved using cyclic scheduling. An example of a correct SystemJ program with write-write and read-write concurrency is presented below.

```
emit S(3); || emit S(4); || if(#S == 7) emit Q; else emit L;
```

Thus, the only way to emulate shared memory communication model in SystemJ is via valued signals.

5.1.5. Causality of SystemJ programs

Until now we have only looked at semantic rules that preserve determinism. Causality of programs is more involved than determinism and is related to signal statuses. Consider the causal program shown below.

```
present(S) emit O; || emit S;
```

In the above program the second synchronous parallel reaction emits signal `S`, which is registered in the first synchronous parallel reaction, which in turn leads to the emission of signal `O`. Thus, the *cause* of the emission of signal `O` is the emission of signal `S`. Now consider the non-causal program below.

```
present(S) emit S; else emit S;
```

In the above program the status of signal `S` cannot be determined by execution alone, i.e., speculation of whether signal `S` is present or not is required for the program to proceed further. Such programs are said to have *causal cycles.* Please note that such programs are however logically correct [19]. The next program is *logically non-deterministic* and incorrect.

```
present (S) emit S;
```

If we speculate that signal `S` is present then `S` is emitted else it is absent. Thus, the two assumptions lead to consistent execution paths. Finally, in the above program if we do not speculate the status of signal `S` then the program is *non-reactive*, it does not execute and is incorrect.

This finishes the presentation of the synchronous semantics. In the next section we present the informal semantics of asynchronous SystemJ statements.

5.2. Intuitive asynchronous semantics

SystemJ is a GALS language. Clock-domains are forked at the top level by the asynchronous operator $><$ and run at unrelated ticks. This section is dedicated to the discussion of the asynchronous operator, channels and the rendezvous as it pertains to SystemJ.

5.2.1. The asynchronous operator $><$

The asynchronous operator is used to combine clock-domains at the top level of a SystemJ program. The clock-domains composed using the $><$ operator run independently and concurrently, synchronizing with each other when they need to communicate. Each clock-domain has its own set of interface input and output signals, which are used to communicate with the environment. These sets of signals cannot be shared between clock-domains. Each clock-domain samples the input signals from the environment, *reacts* to these input signals and produces the output signals, which are emitted to the environment at the *End of Tick* (EOT). The asynchronous operator, like the synchronous operator, results in concurrent interleaved execution of clock-domains. However, there is no synchronization of the clock-domains at the end of their ticks. This makes SystemJ GALS programs non-deterministic. This non-determinism is an important property of SystemJ programs and makes SystemJ suitable for programming important non-deterministic systems like communication protocols.

```
system{
 interface{
  input signal I,I1;
  output signal O,O1;
  channel C;
 }
 {
   //The body of the system
   {
    //First clock-domain
    present(I)
     emit O;
    send C(); //synchronizing with the second clock-domain
   }
   ><
   {
    //Second clock-domain
    present(I1)
     emit O1;
    receive C; //synchronizing with the first clock-domain
   }
 }
```

The program above shows an example use of the asynchronous operator (><). The complete SystemJ program is called a `system`. A system consists of an `interface` and a body. The `interface` is used to declare the interface signals that communicate with the environment and channels for cross clock-domain communication. In this case two input signals `I` and `I1` are declared in the `interface`. Signal `I`, which is used in the first clock-domain cannot be used in the second clock-domain and vice-versa. The compiler makes sure that this requirement is followed. The same semantics apply for the output signals `O` and `O1`. SystemJ performs this separation of signals to guarantee that the channels are the only means of communication between clock-domains. In the example above, a `channel C` is declared and used to synchronize between the two reactions in the partner clock-domains.

The body of the `system` is used to initialize reactions as clock-domains using the asynchronous operator ><. The reactions initialized as clock-domains can be named or unnamed. Examples of named reactions initialized as clock-domain has been shown in Listing 1 lines 20-40. Unnamed reactions initialized as clock-domains is shown in the example above.

5.2.2. Channels in SystemJ

Channels are declared in the `interface` of a SystemJ program and are the only means of communication between reactions in separate partner clock-domains. Every declaration of a channel initializes an *`input port`* and an *`output port`* for that channel, respectively. These channel ports are used for inter clock-domain communication. The *`output port`* is used for sending data, which is in turn received by the corresponding *`input port`*. The `send` statement works on the *`output port`* to synchronize and send data across clock-domain to the *`input port`*, which is used by the `receive` statement. Channel communication is point-to-point, i.e., for every sending statement using a channel there needs to be a corresponding receiving statement. Finally, a sending or receiving statement can only be used once on a channel in any given clock-domain. Some examples of correct and incorrect SystemJ programs are presented below, which illustrate the channel semantics.

```
//Incorrect SystemJ program. Multiple declarations of the same
//channel is not allowed.
interface{
 channel C;
 int channel C;
}

//Incorrect SystemJ program. Channels are point to point
//fan ins and fan outs are not allowed when communicating with
//channels
{send C();} >< {receive C;} >< {send C();}

//Incorrect SystemJ program. Sending and receiving cannot
//be used twice on the same channel.
{send C()<++>; send C();} >< {receive C; receive C;}

//Correct SystemJ program
interface{
 int channel C;
 int channel C1;
}
{send C(3); receive C1;} >< {receive C; send C1(#C);}
```

In the correct SystemJ program above, two channel are explicitly declared to communicate and send and receive data between two clock-domains. The first clock-domain sends an integer value of 3 to the second clock-domain, which is returned from the second clock-domain back to the first clock-domain. Channel communication is uni-directional, i.e., data can only be transferred from the sending clock-domain to the receiving clock-domain and hence, two channels are required in the above example.

5.2.3. Rendezvous in SystemJ: the `send` and `receive` statements

The `send` and `receive` statements are used to synchronize and send data between reactions in partner clock-domains. SystemJ uses CSP [10] style rendezvous for this synchronization and data transfer. These statements can be used to synchronize between free running clock-domains and send data if any. In this section we present the algorithms used to implement `send` and `receive`. The SystemJ compiler rewrites the `send` and `receive` statements into a number of synchronous statements from Table 1(a).

Every channel is endowed with the following statuses and value buffers that are used to implement CSP style rendezvous communication:

- A data buffer O_cv for the output channel port, which holds the data to be sent. It can hold any Java data type.

- A data buffer I_cv for the input channel port, which holds the received data from the sending clock-domain. Again, this buffer can hold any Java data type.
- A write-sent status O_cw_s for the output channel port, this status indicates that the sending clock-domain is ready to rendezvous and send data if any. This status is implemented as an integer type.
- A read-received status I_cr_r for the input channel port, which holds the sampled write-sent status of the sending clock-domain, and informs the receiving clock-domain that the partner clock-domain is ready to rendezvous. Again, this status is of type integer.
- A read-sent status I_cr_s for input channel port, which sends back an acknowledgement from the receiving clock-domain to the sending one, informing that it is ready to rendezvous. This status is of type integer.
- A write-receive (integer type) status O_cw_r for the output channel port, which holds the sampled read-sent status from the receiving clock-domain.
- Preemption (integer type) statuses O_cp_s and I_cp_r for the output and input channel ports respectively, which deal with strong preemptions.

All integer statuses have an interval of $[0, 1, 2, .., n]$. Throughout the presentation of semantics, for brevity, we use just C to represent both output channel port O_c and input channel port I_c when the context is non ambiguous (example, Cr_s instead of I_cr_s). All channels statuses are initialized to a value of zero at channel declaration and the channel value buffers are initialized to null. It should be noted that channels never suffer from schizophrenia as they are declared only once in the `interface` of the SystemJ system.

The rendezvous implementation and semantics of SystemJ differ from all other existing asynchronous and GALS models/languages. SystemJ allows *weak* and *strong* preemption, due to `abort`, on rendezvous statements `send` and `receive`. Each clock-domain *stores* the sampled channel statuses at the start of the tick from its partner clock-domain, which are later used to deduce if the partner clock-domain is ready to rendezvous. A rendezvous is completed iff both the partner clock-domains are synchronized and the data transfer, if any, is successful. The signal based preemption of an ongoing rendezvous makes the rendezvous implementation non-trivial.

There are six possible scenarios, which the compiler needs to deal with when implementing rendezvous in SystemJ. (i) the sending clock-domain is in the rendezvous state while the receiving one is uninitialized when it gets preempted, (ii) the receiving clock-domain is in the rendezvous state and the sending is uninitialized when it gets preempted, (iii) the sending and receiving clock-domain are both in the rendezvous state when they both get preempted, (iv) the partner sending and receiving statements are uninitialized and both get preempted, (v) the sending and receiving clock-domain are both in the rendezvous state and only one gets preempted, and, (vi) the rendezvous completes without any preemption.

There are a few possible deductions one needs to make depending upon the channel statuses which we enumerate here:

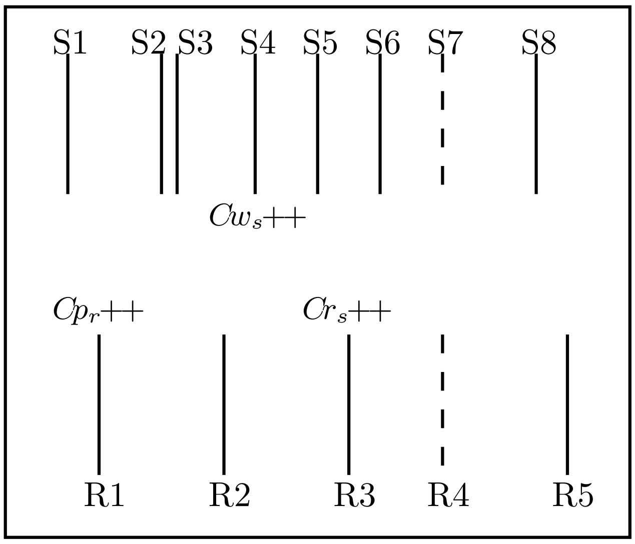
- For the output channel port, $Cw_s = Cw_r$ means that the buffer is empty and thus the data can be filled into the sending buffer. An empty buffer is possible if the earlier rendezvous was successful, if the earlier rendezvous was preempted, or if this is the very first attempt at rendezvous.

- For the input channel port, $Cr_r > Cr_s$ means that the sending and the receiving clock-domains are both in the rendezvous state and the sending buffer is full with data. This rendezvous would be successful in the next instant if there is no strong preemption in either of the clock-domains.

- For both the output and the input channels ports, $Cp_s \neq Cp_r$ means that the partner clock-domain is already preempted and thus the current clock-domain needs to respond accordingly.

The validity of these deductions becomes clear while stepping through the rendezvous algorithms as control point movement. We use Figure 4 to better

Figure 4: Intuitive rendezvous representation



explain the sending and receiving algorithms presented in Algorithms 1 and 2, respectively.

First, we consider the *operational semantics* of the `send` statement. A rendezvous is possible iff both the partner domains are ready to participate. In our implementation, a sending clock-domain can actually "send" the value without any concern if the receiving clock-domain is ready to "receive", provided that the output channel port buffer is empty. This behaviour arises from the fact that both `send` and `receive` have separate data-buffers.

The send algorithm fills the sending buffer (C_v) with data and increments the channel port status Cw_s. Next, the sending algorithm pauses waiting for an acknowledgement from the receiving clock-domain (Algorithm 1 lines 2-15). Send can now take multiple ticks to receive an acknowledgement from the receiving clock-domain. Send completes rendezvous when acknowledgement is received from the partner clock-domain. The write-received status (Cw_r) is updated with receiving clock-domain's read-sent status (Cr_s) at the start of the sending clock-domain's tick. The completion transition of the sending clock-domain is shown in Algorithm 1 lines 9-16.

Algorithm 1 Send Algorithm

```
1:  abort (immediate $Cp_r > Cp_s$)
2:    abort (immediate $Cw_s == Cw_r$)
3:      while (true) do
4:        pause ; //blocking while channel buffer is full
5:      end while
6:    end abort
7:    $Cw_s$++; //updating channel status (happens at tick S4)
8:    $Cv$ = data; //sending data (happens at tick S4)
9:    abort (immediate $Cw_s == Cw_r$)
10:     abort (immediate $I_{ps}$)
11:       while (true) do
12:         pause ; //blocking if no acknowledgement (ticks S4-S7 are generated)
13:       end while
14:     end abort
15:   end abort
16:   do
17:   reset($Cw_s$,$Cw_r$,$Cv$); //resetting if partner preempted
18:   $Cp_s = Cp_r$;
19: end abort
20: do
21: reset($Cw_s$,$Cw_r$,$Cv$); //resetting if partner preempted
22: $Cp_s = Cp_r$;
```

For example, in Figure 4 `send` is initialized at tick S4. At tick S4, the output channel port status Cw_s gets incremented and the sending buffer is filled with data. Next, `send` starts consuming ticks until it receives a reply from the receiving clock-domain or gets preempted. Thus, ticks S4-S7 are all generated internally by the rendezvous algorithm.

`Receive`, unlike `send`, cannot read its value buffer until both the clock-domains are synchronized (else it will read non-initialized data). Thus, `receive` acts as the true synchronizer. The receiving algorithm blocks waiting for `send` to be ready to synchronize (Algorithm 2 lines 2-6). The receiving clock-domain samples the sending clock-domain's write-sent status (Cw_s) at the start of its tick and stores it in the read-received status (Cr_r). Thus, a read-received status greater than read-sent status indicates that the sending clock-domain is ready to rendezvous. Remember that they are both initialized to zero. If the sending clock-domain is not ready then read-received and read-sent status are equal.

Once the sending clock-domain gets ready to synchronize, the receiving clock-domain increments the read-sent status (Cr_s), which is in turn sampled by the sending clock-domain as an acknowledgement. Next, it pauses (Algorithm 2 lines 8-9). In Figure 4 `receive` statement increments Cr_s at tick R3, since `send` is already ready to rendezvous. The next tick of both the clock-domains occur together (ticks S7 and R4) and data transfer takes place from the sending clock-domain's buffer to the receiving clock-domain's buffer (Algorithm 2 line 10). Once the rendezvous is complete the sending and receiving clock-domains are released to continue at their own pace.

The above steps are enough to guarantee a rendezvous between clock-domains (scenario (vi)). But, further channel status checks and steps are needed to deal with preemption scenarios (i) through to (v). From these preemption scenarios one can identify two main participants. One is the clock-domain that gets preempted by a synchronous statement in its body, and other is the partner clock-domain that needs to respond to this preemption. First, we define the operation of the preempting clock-domain, and then the semantics of the partner clock-domain.

Algorithm 2 Receive Algorithm

```
 1: abort (immediate $Cp_s > Cp_r$)
 2:   abort (immediate $Cr_r > Cr_s$)
 3:     while (true) do
 4:       pause ; //blocking if partner is not ready
 5:     end while
 6:   end abort
 7:   abort (immediate $I_{ps}$)
 8:     $Cr_s$++; //sending acknowledgement (happens at tick R3)
 9:     pause ; //tick R3
10:     $Cv$ = data; //receiving data (data received at tick R4)
11:   end abort
12:   do
13:   reset($Cr_s$,$Cr_r$,$Cv$); //resetting if partner preempted.
14:   $Cp_r = Cp_s$;
15: end abort
16: do
17: reset($Cr_s$,$Cr_r$,$Cv$); //resetting if partner preempted.
18: $Cp_r = Cp_s$;
```

In SystemJ every preemptive statement enclosing a `send` or `receive` statement has the capability of preempting an ongoing or uninitialized rendezvous. It also has the capability to call the `reset` function on the subset of channels bound to the preempted `send/receive` statements. The `reset` function resets every channel port status and value buffer to zero, except for the preemption statuses p_s and p_r, which are incremented. The partner clock-domain samples and samples the preemption statuses at the start of their ticks. If the partner clock-domain's preemption status is greater than its own, the clock-domain preempts the rendezvous. Algorithms 1 and 2 line 1 implement this behaviour: the immediate abort statement is checked at every tick before making any attempt at rendezvous. Algorithm 1 lines 21-22 and Algorithm 2 lines 17-18 call the reset function if the abort condition is true. In Figure 4, if the receiving clock-domain gets preempted at tick R1, then the `reset` function resets the value buffer and the receiving channel statuses (except the preemption status). At tick S4, the sending clock-domain, instead of incrementing its write sent status, responds to this preemption by resetting its own value buffer and statuses in accordance with Algorithm 1.

These preemption rules take care of all the possibilities except for scenarios (iii) and (v). Once in the rendezvous state (Figure 4 ticks {S7,R4}), either of the clock-domains might get preempted in the rendezvous instant itself (due to strong aborts), and thus the partner domain would not be able to respond. In order to deal with such a possibility, CRP [9] takes the approach of blocking the input signals from the environment in this instant of time. We use a different implementation technique, which maintains the autonomy of the clock-domains while resulting in the same semantic behaviour:

- For the preemptive scenario (iii), if both the clock-domains get preempted while in rendezvous state, then the reset function is called and there is no need for the partner clock-domain to respond since the preemptive statuses of the partner channels are unmodified.
- On the other hand, if, while performing the rendezvous only one clock-domain gets preempted (scenario (v)), then partner response becomes essential. This is taken care of by testing a shared input signal set I_{ps}, which represents all the possible input signals that can preempt the rendezvous in the partner clock-domains, for presence in both the clock-domains. Thus, the current scenario gets mapped to the aforementioned one (scenario (iii)). This behaviour is semantically correct since, while performing rendezvous, the two clock-domains act as a synchronous system and thus strong preemption when in rendezvous state can be defined as synchronization without data transfer. In Algorithm 1, lines 10 and 17-18 take care of aborting the ongoing data-transfer by checking the preemption set I_{ps}. Similar behaviour is implemented in Algorithm 2 lines 7 and 13-14. Finally, it should be noted that multiple copies of the shared input signal set I_{ps} are kept in different clock-domains. This allows for easier distribution of clock-domains.

Rendezvous guarantees many properties such as data delivery, synchronization and ordered data delivery. But, it is easy to introduce deadlocks with rendezvous. SystemJ compiler is deadlock aware. It helps the designers by giving a compile time error when deadlocks are introduced in the system due to `send` and `receive` statements. Due to the lack of space we refer the reader to [22] for a complete description of deadlock detection algorithm implemented in SystemJ.

5.2.4. Non determinism in SystemJ

In this section we concentrate on *non-determinism* as the final semantic issue that is of importance and can be a useful design tool for system designers.

The asynchronous interleaving makes SystemJ a non-deterministic language (just like all other asynchronous concurrent systems). Consider for instance Listing 3: the first clock-domain sends an integer value 2 on output port of channel 'C' (line 9). The second clock-domain uses two reactions, one receives the value on the input port of channel C (line 17) while the second checks if the value received is 2 (line 22). If the value is 2 and signal A is present, then it emits signal D, else it emits signal F. The time for inter clock-domain communication is not known a priori. If signal A is present in each instant of time, the second clock-domain can still emit either signal D or F depending upon the success of the synchronization of the two clock-domains. Thus, this system produces different output traces for the same input signal trace. In accordance with Definition 1 the SystemJ system is non-deterministic. Hence, a clock-domain in SystemJ is completely deterministic, but the asynchronous composition of clock-domains leads to non-deterministic system behaviour. This non-determinism is an essential feature for a designer who wants to model non-deterministic systems, such as communication protocols.

Listing 3: An example of inherent non-determinism in the SystemJ language

```
interface {
 input signal A;
 output signal D,F;
 channel C;
}
{//First asynchronous parallel reaction
 while(true){
  send C(2);
  pause;
 }
}
><
{//Second asynchronous parallel reaction
 //The two reactions reac−1 and reac−2 run in lockstep
 { //reac−1
  while(true)
   receive C;
 }
 || //synchronous operator
 { //reac−2
  while(true){
  //It should be noted that reading a channel value
  //is a java data−driven computation
  if(#C == 2 && A)
   emit D;
  else
   emit F;
  }
 }
}
```

6. Formal SystemJ semantics

The kernel SystemJ is the most basic subset of the complete SystemJ language. There are a myriad ways to represent the semantics of any given language. The structural operational semantics first proposed in [23] is not only a very efficient human-oriented method of specification for real languages, but also has been shown to produce very efficient software compilation techniques [24] for system-level languages. Thus, we have chosen the *constructive structural operational semantics* as the basis for SystemJ description. All the semantical rules presented utilize a structural translation scheme. We use one or more semantical rule(s) to rewrite the reactive control and Java data statements. Such a translation scheme helps us obtain a direct intermediate representation of the program from which back-end code can be efficiently generated. The semantical rewrite rules presented are very fine grained, being targeted towards compiler construction, and thus, we also call them micro-step kernel semantics.

In this section we present the general rule for a SystemJ micro-step transition and explain the terms involved in this rewrite rule. Due to lack of space we present only the asynchronous semantic rules in Appendix A. The synchronous rules, which form the majority of the SystemJ kernel statements can be obtained from [25, 24].

Let p be a SystemJ kernel statement, we write

$$term(p), data \xrightarrow[E,E_c]{k,e} term'(p), data' \tag{1}$$

Here $term(p)$ and $term'(p)$, represent the antecedent and consequent states of p respectively, during a micro-step transition. Term e represents the signals that are emitted during transition and if none are emitted then it takes the value of $\perp$. Term $data$ represents the value stores attached to the statement p before transition and $data'$ after the transition. Term k represents the completion code. It has a value of $\perp$ i.e., unknown, if, p does not generate a completion code after this transition, else, an integer value. Only `exit`, `pause`, and `;` terms are capable of generating a completion code. Term `;` is encoded with 0, `pause` with 1, and `exit T` with an integer greater than or equal to 2.

Other than the above mentioned terms, a termination code of ∞ can also be produced by a synchronous parallel reaction iff a signal or data dependency has not yet been resolved (cyclic scheduling). Input event E is the status of all the signals used in p, but declared somewhere else. E_c is the status of all the channel ports used in p but declared somewhere else. For n number of channels there are $2n$ number of input and output ports. Thus, the cardinality of set E_c is 2^{2n}. Hence, for a channel C, $E_c = \{\{Cw_s, Cw_r, Cp_s\}, \{Cr_s, Cr_r, Cp_r\}\}$. Here, the set's elements represent the output and input channel port statuses as described previously in Section 5.2.3. In the transition rules, for brevity we use the array indexing notation to refer to the channel and signal statuses. Hence, $E[Cw_s]$ represents the output channel port's write-sent status w_s, i.e., $Cw_s \in E_c$. Similar indexing notation is used for signal statuses.

A statement is also said to be *selected* iff a pause is hit during the execution of statement p. A *selected* statement is further decorated with a $\hat{p}$. We use the notation $\bar{p}$ to indicate that the selected state for term p is currently unknown. We refer the reader to [19] for long but simple definitions of $\hat{p}$ and $\bar{p}$.

The example below shows the micro-step semantics of `pause` execution during program flow. When a `pause` is hit for the first time (*also called the start rule*) the statement gets selected and the program ends with a completion code of 1. In the next instant (*also called the resumption rule*) the selected statement continues further and completes execution (completion code 0). The *selection* status is *upward-propagative*. Thus, any statement enclosing a `pause` is considered selected if the enclosed `pause` itself is currently selected. In the rules below data stores have been omitted since we are dealing with a pure synchronous statement:

$$\bullet pause \xrightarrow[E]{1,\perp} \widehat{pause}$$

$$\bullet \widehat{pause} \xrightarrow[E]{0,\perp} pause$$

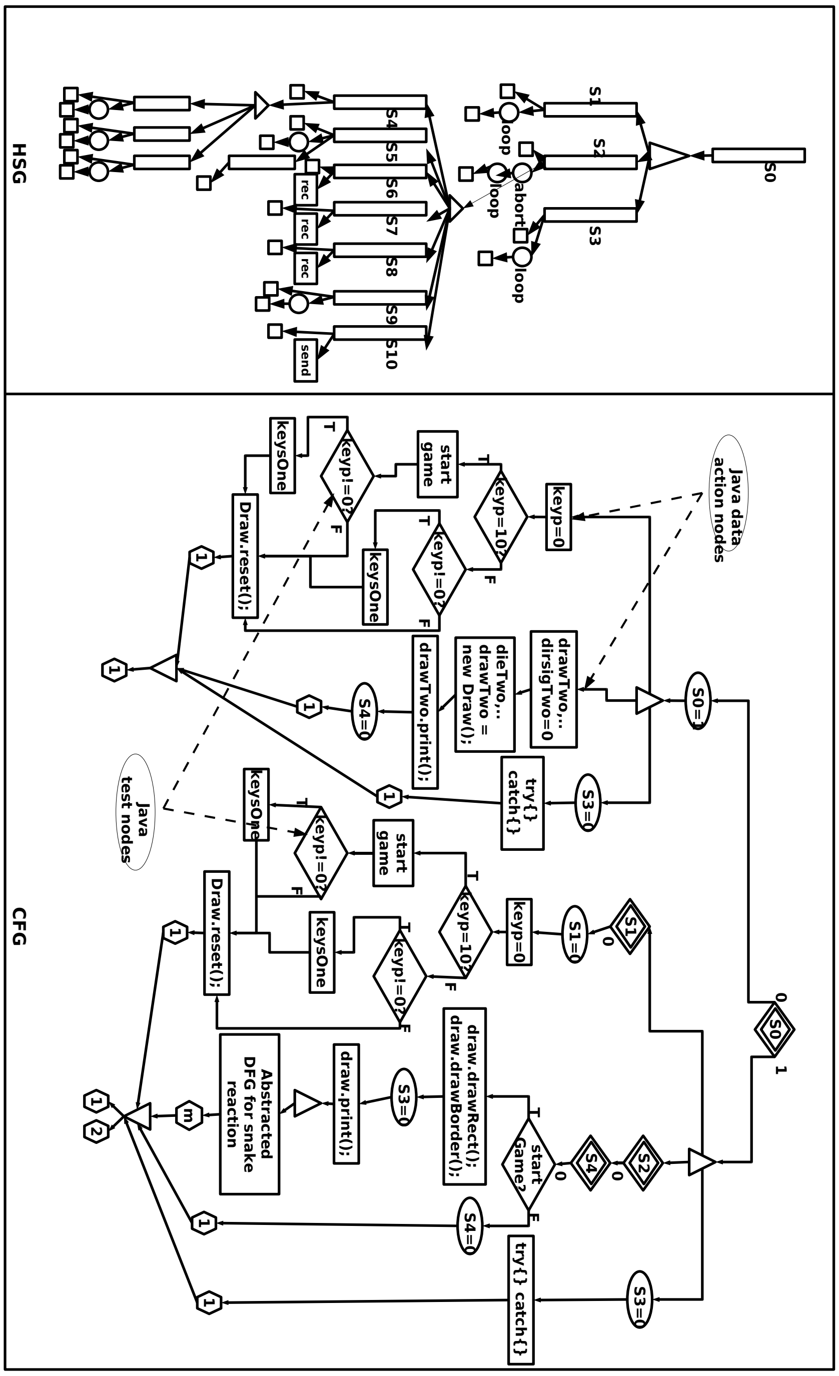


Figure 5: GRC representation of the Clock-domain 1 for the two-player snake game

The SystemJ compiler can be viewed as an extension of the GRC compiler approach [24] used to compile Esterel. GRC representation consists of two graphs; the *hierarchical state graph* (HSG), which preserves the structural information, and the *control flow graph* (CFG), which describes the operational aspects explicitly. These two structures handshake with each other to identify where control will be in a given instant. Figure 5 gives the GRC representation of the first clock-domain (*player-1*) in the game example from Section 3.

7.1. Hierarchical state graph (HSG)

The HSG outlines the structure of the synchronous reactions in terms of statements and threads of control. It consists of four different types of nodes. *Thread* nodes (rectangles) abstract the sequential composition of basic instruction inside a particular thread. *Parallel* nodes (triangles) represent concurrent threads. Meanwhile, loops, aborts, and other backtracking statements are represented as *compound* nodes (circles). Pauses are represented with *boundary* nodes (squares). The number of branches extending from a thread node indicates the number of possible states for that thread.

7.2. Control flow graph (CFG)

The CFG is the more expressive object of the GRC. It is the primary input for the code generation stage. The graph explicitly describes the complex control flow through seven primitive nodes. Signal emissions are represented using *action* nodes (rectangles) and state encoding with *enter* nodes (ellipses). On the other hand, signal tests are represented using *test* nodes (diamonds), while state selection is indicated using *switch* nodes (double-diamonds). Concurrency is delineated using *fork* (triangle) and *join* (inverted triangle) nodes. *Terminate* nodes (hexagon) mark the completion code of a given thread. The nothing and sequential statements are encoded using *action* nodes, while they are completely ignored in the HSG.

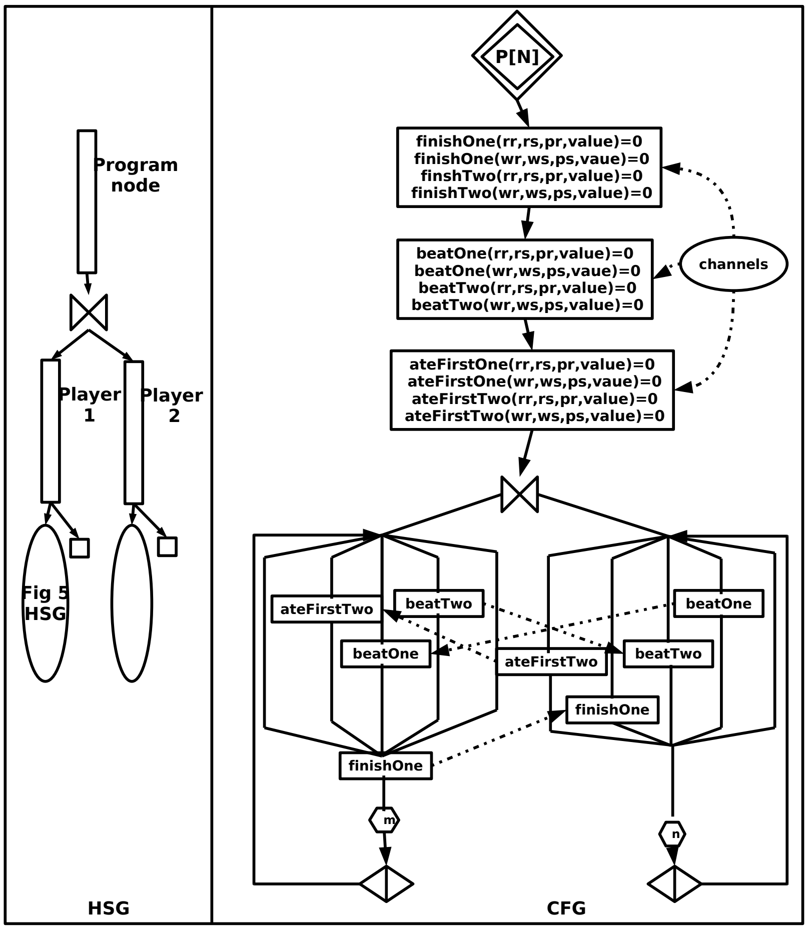
7.3. From GRC to AGRC

The GRC intermediate format is well suited for purely synchronous languages like Esterel or the synchronous sub-set of SystemJ programs, i.e., single clock-domain programs. But SystemJ further enhances synchronous reactions with asynchronous coupling, thereby making the current GRC format insufficient. In this section, we propose a number of changes that need to be incorporated into the current GRC format in order to make it suitable for compiling asynchronous MoC. The resulting *Asynchronous GRaph Code* (AGRC) format derived directly from SystemJ semantics (Section 6 and Appendix A) is the intermediate representation into which SystemJ is translated before being translated into low level Java back-end code.

7.4. Asynchronous modifications to GRC

The asynchronous statements `send`/`receive`, the >< operator, and the channels need to be incorporated into the GRC. This section describes the additional nodes supporting asynchrony and the translation of SystemJ into the resulting intermediate format, called AGRC.

Figure 6: AGRC representation of the Multi-player snake game example


P[N]
finishOne(rr,rs,pr,value)=0
finishOne(wr,ws,ps,vaue)=0
finshTwo(rr,rs,pr,value)=0
finishTwo(wr,ws,ps,value)=0
beatOne(rr,rs,pr,value)=0
beatOne(wr,ws,ps,vaue)=0
beatTwo(rr,rs,pr,value)=0
beatTwo(wr,ws,ps,value)=0
ateFirstOne(rr,rs,pr,value)=0
ateFirstOne(wr,ws,ps,vaue)=0
ateFirstTwo(rr,rs,pr,value)=0
ateFirstTwo(wr,ws,ps,value)=0
channels
Program node
Player 1
Player 2
Fig 5 HSG
ateFirstTwo
beatTwo
beatOne
ateFirstTwo
beatTwo
beatOne
finishOne
finishOne
m
n
HSG
CFG

7.4.1. Implementing channels

In accordance with the earlier defined model and semantics of channels, Section 5.2.2 and Appendix A, channels have their integer statuses and generic value buffers initialized to `null`. Also, unlike signals, channels do not suffer from schizophrenia since they are declared only in the interface of the system reaction. Thus, it suffices to initialise channels with action nodes at the start of the clock-domain. Figure 6 shows this channel implementation in the CFG part of the AGRC. Please note that for every channel (example `finishOne`) we initialize a pair of *`input`* and *`output`* channel port statuses.

7.4.2. Implementing Asynchronous operator

Figure 6 shows the *asynch-fork* (vertex joint triangles) and *asynch-join* (base joint triangles) used to represent forking and joining of clock-domains respectively. In accordance with the semantics of the >< operator (Section 5.2.1 and Appendix A rules 4 through to 5b), the asynch-fork node is traversed only once in the first tick to start all the clock-domains. The following ticks are all confined within the clock-domains. The asynch-join node is utilized for scheduling, to make sure that clock-domains do not finish with ∞ as their completion codes. It is also used to emit output signals to the environment and to read new inputs from the environment, in accordance with the previously defined semantics (Section 5.2.1).

7.5. Equivalence between system specification and implementation

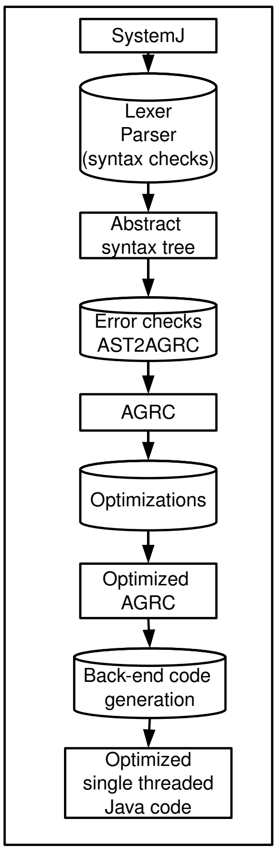
This section explains how the CFG, which is the main input to the back-end code generation stage preserves the semantics and the behaviour of a SystemJ program. We explain the equivalence of the CFG and the SystemJ program by visiting the nodes in the CFG for the two player snake game presented in Figure 5 as control point movement. The reader is advised to refer to Figure 5 and the corresponding SystemJ program code in Listings 1 and 2 respectively.

All the states in the CFG are initialized with a value of zero. The control point first enters the CFG and the state decoding node `S0` transfers the control to the left most branch. This branch represents the very first tick of program execution. In this branch first the state `S0` is encoded with 1 and then three synchronous parallel reactions are forked by the fork node. These three parallel reactions show the `getKeys`, `snake` and the `makePause` reactions from Listing 1 lines 21-29 respectively.

The `getKeys` reaction (not shown in details in the listings) initializes the key pressed variable and checks the value of the key pressed by the player. The signal `startGame` is emitted by this reaction if the player presses the `Return` key. If any key other than `Return` key is pressed (test node `keyp != 0`) then that value is emitted via signal `keysOne`. The `snake` reaction, second forked reaction in Figure 5, initializes a number of signals and variables as shown in Listing 2 lines 10-14 and pauses. Thus, in the very first instant of time the `await` statement, Listing 2 line 15, is not carried out. This behaviour is as expected. Since the `await` statement is not qualified with an `immediate` qualifier the checking for the `startGame` signal only happens from the second tick. The `makePuase` reaction, the third branch in Figure 5, simply sleeps for the specified amount of time using Java's `Thread` class. Once the sleep timer has expired this branch pauses (the hexagon with termination code of 1 shows this pausing). Once all the three branches have completed a tick the join node carries out the synchronization and finishes with a termination code of 1 (the maximum of all termination codes). The clock-domain finishes the tick and interacts with the environment, reading the next set of input signals and emitting any output signals from this tick.

In the next tick, the state `S0` gets decoded as one. The control enters the right branch of the state decoding node. In this tick, like in the first tick, three reactions are forked. In this iteration we only concentrate on the `snake` reaction, the second branch. In this iteration the `snake` reaction checks if the `startGame` signal is present. If the signal status is `true` then, the control proceeds further to carry out the rest of the program as shown in Listing 2 lines 17-100. In Figure 5 we have abstracted out this functionality to make it easier to understand. If signal `startGame` status is `false` then the reaction just pauses. In this way the CFG preserves the behaviour of the designed SystemJ system.

Figure 7: SystemJ compilation flow



8. Compiling SystemJ into Java

SystemJ's compilation target is Java source code. There are two main reasons we target Java:

1. Java is used for data-driven computations and transformations in SystemJ and, hence, Java is the most natural compilation target.
2. Java is highly portable. A SystemJ system designed on a desktop does not need to be recompiled to be run an another platform like an embedded system.
3. Java compilation lets us use the Java compiler for further optimizations and checking.

Compiling SystemJ into Java code is a multi-stage process as shown in Figure 7. In the first stage of lexical analysis and parsing we perform syntactic checks like multiple declarations of same named signals, or declaration of input/output signals and channels outside the interface scope. Next, we perform rigorous front-end error checking on the resultant abstract syntax tree, which includes checking if input signals are being shared in multiple clock-domains, if output and input channel ports have corresponding usage, and if the same channel name is being used multiple number of times. Next step is the translation into the AGRC format. Type checking in SystemJ is handled by the Java compiler when we compile the back-end generated Java code for execution.

The translation into AGRC is the most demanding step. During this stage, structural translation rules (Appendix A and [25]) are followed to translate each statement into one or multiple nodes of AGRC, as described in Section 7. The resulting sub-graphs are then composed together to build the complete AGRC representation of the SystemJ source. During this stage, the cyclic scheduling algorithm proposed in [21] and summarized in Section 5.1.4 is implemented. We chose this algorithm rather than statically scheduling synchronous threads in order to reduce code size and to make the back-end code portable to a multiple processor architecture at a later stage [26].

The optimization stage includes implementation of well known algorithms like redundancy elimination and information propagation. Lastly, the back-end code generation stage is carried out on the resulting AGRC. An example SystemJ program and its equivalent generated Java code is shown below.

SystemJ code

```
present(S){
 emit A(1);
}
pause;
emit B;
```

Generated back-end Java code

```
switch(S0){
 case 2: S0=2; break;
 case 0:
 A0.setClear();B0.setClear();
 if(S.getStatus())
  A0.setPresent();
  A0.hook(new Integer(1));
  ends[0]=1;S0=1;
  break;
 case 1:
  A1.setClear();B1.setClear();
  B1.setPresent();
  ends[0]=0; S0=2;
  break;
}
```

In the back-end, we use Java switch statements instead of Java threads to emulate synchronous concurrency. For example, in the back-end generated code above, the `switch` statement selects between cases 0, 1 and 2. Here, case 2 represents termination of the reaction. The case 0 represents the very first transition of the reaction. Finally, case 1 represents all other transitions. In the above example the `switch` statement first enters the case 0 and checks if the signal `S` is present. If the condition evaluates as true then signal `A` is emitted. Else, no signal is emitted and the reaction pauses (shown by setting the termination code, the `ends` array to 1). Emitting signal `A` involves setting its status to true (`A0.setPresent()`) and setting its value using the associative and commutative `hook` function, which is provided by the designer. The program then communicates with the environment by emitting the output signals to the environment and reading the next set of input signals. The Java program then proceeds to carry out the next tick. In this tick the signal `B` is emitted and the program terminates. It should be pointed out that different instances of the same signal (example `A0` and `A1`) are used to avoid schizophrenic behaviour.

Single threaded back-end implementation is much more efficient in terms of memory consumption and execution time compared to using Java threads like in TReK [17]. The generated single threaded Java code is OS independent: Java threads are mapped to native threads and thus the implementation depends upon the underlying OS scheduler. Besides, verification techniques like model-checking are much more amenable to a single threaded Java program as compared to a multi-threaded one.

The translation into Java from SystemJ for the kernel statements is shown in Table 2. At the back-end, a Java package consisting of one Java class per

Table 2: Compiler Translations from SystemJ to low level Java statements

SystemJ-statements	**Java-translations**
`abort/suspend/present (S){term(p)}`	`if(S.getStatus()) term(p) else;`
`exit(t)`	`exit[thread-num] = 2;`
`reaction p(:){term(q)}`	`public static void p(){}`
`system {}`	`public class system {}`
`signal S;`	`Signal S = new Signal();`
`channel C;`	`Channel C = new Channel();`

SystemJ "system" is generated with reactions/clock-domains being mapped to Java methods (Table 2). We also generate helper signal and channel classes that support generic (Object type) data transfers and emissions. For scheduling asynchronous clock-domains we provide multiple solutions:

1. The default solution is to schedule the clock-domains in a round robin fashion.
2. The second scheduling scheme involves allowing the designer to specify the speed of the clock-domains as tick ratios with respect to a master clock-domain chosen by the designer.
3. Finally, the designer is also allowed to emulate unrelated clock-speeds for multiple clock-domains. This involves letting the user run clock-domains multiple number of times, manually.

For the last solution we have implemented a step-by-step debugger having the ability to display output signals and channels, and to read in test input vectors similar to [5]. The SystemJ compiler provides a switch at compilation stage to choose amongst these scheduling mechanisms.

9. Results

This section presents the results of comparing the AGRC compiler implementation of SystemJ with the TReK-library based approach.

Esterel [19] can only model synchronous systems and hence, comparing SystemJ GALS models with Esterel would be unfair. Rewriting GALS systems into synchronous systems is one way of comparing the two languages but, this rewriting changes the behaviour of the designed systems and hence is unsuitable for comparison purposes. Comparison with MC-Esterel has currently been avoided because MC-Esterel targets purely hardware generation [13], while we are currently targeting efficient software back-end code compilation. The software compilation of MC-Esterel compiles away the asynchrony, producing a single synchronous program using clock gating [14]. Comparison with SHIM would give an unfair advantage to SystemJ as SHIM does not directly support reactive class of programs, thereby leading to large design and runtime overhead. Finally, both CRSM and ECRSM provide very inefficient data-driven support, thereby making comparison with SystemJ infeasible. Please also note that Esterel is practically the only language with available compilers [27], [5], [28].

Table 3: Attributes of the GALS benchmark suite

Example	**Lines of source code**		**Number of synchronous parallel reactions**	**Number of clock-domains**
	SystemJ	Java		
`abcd`	85	0	4	1
`ww`	615	504	32	1
`bhs`	1211	0	31	1
`aps`	154	31	6	2
`frelay`	663	219	13	2
`camera`	377	1139	13	3
`snake`	185	5118	8	3

Our benchmark suite consists of a number of models ranging from simple ones (a few lines of code) to large real-life models. The GALS benchmark suite is available from [29]. Table 3 gives the source code length and other attributes, like number of synchronous parallel reactions and number of clock-domains. This table is provided to illustrate the complexity of the chosen benchmarks.

The synchronous examples `abcd` (combination lock) and `ww` (wrist watch) are due to Berry and are chosen from the Esterel test-bench suite [30]. The `bhs` (baggage handling system) example is designed by us and is based on a real-life baggage control mechanism for airports. The GALS examples include the asynchronous protocol-stack (`aps`), frequency-relay (`frelay`) [31], `camera` and `snake`. The `frelay` example is a real-life case study used to control the power relay switch, while the `aps` example is a GALS system obtained from ECL [32] and purposely chosen to test the language with heavy data-driven computations. Finally, the `camera` and `snake` examples were developed by us. The `camera` system is a real-life system employed for secure surveillance, while the `snake` example is a two-player snake game.

Table 4 gives a runtime comparison between the AGRC and the TReK implementations of SystemJ. As expected, AGRC is, on average, 67% faster than TReK. This is mainly due to TReK's reliance on Java threads and templates. The anomaly whereby the `aps` example takes longer to run as compared to `frelay` can be attributed to the data computation needed. Indeed in the `aps` example, each bit of a given byte needs to be updated so the runtime is greater.

These results outline the strengths and weaknesses of SystemJ as a system

Table 4: Runtime performance of TReK vs AGRC compiler speed/tick

Example	**TReK runtime(ms)**	**AGRC runtime(ms)**
`abcd`	0.0187	**0.00107**
`ww`	0.627	**0.0427**
`bhs`	0.96202	**0.654**
`aps`	20	**0.3**
`frelay`	0.343	**0.221**
`camera`	0.35	**0.26**
`snake`	1.38	**0.097**

development language. SystemJ's tight integration of control and data statements provide a highly abstracted design environment. In addition to reducing the amount of source code required to design GALS MoC, the new AGRC compiler significantly reduces the execution time for the GALS examples.

10. Comparison with other models of computation and reactive Java packages

CRSM/CRP Ramesh's CRSM [6] and CRP [9] are the models closest to SystemJ's MoC. CRSM's nodes can be considered as clock-domains, which are connected using asynchronous operator. Even so, differences lie in the asynchronous rendezvous implementation and data computations. While SystemJ uses less resource hungry single buffer based rendezvous, CRSM and CRP both use coordinator based rendezvous. SystemJ's data computation capabilities also far exceed CRSM's. Unlike CRSM and CRP we are able to embed complex data-driven computations within the control flow, thereby easing the programming burden. Also, compared to CRP which allows strong preemption on rendezvous like SystemJ, we are able to maintain asynchronous autonomy and thus able to target distributed implementation. CRP, on the other hand, blocks clock-domains in the rendezvous state, and thus distributed implementation becomes less efficient.

SHIM Edward's SHIM [7] is a fairly new language adopting a hierarchical asynchronous MoC targeted at modelling mixed software and hardware systems. Like SystemJ, SHIM also uses a single buffer underlying layer for modelling rendezvous, but SystemJ's MoC is much more flexible and heterogeneous than SHIM's.

SystemJ encompasses the synchronous hypothesis like Esterel, i.e., it has dedicated statements to deal with reactivity, including signals, various preemption mechanisms (immediate, strong, and weak) on signals etc. SHIM, on the other hand, uses C-like statements to emulate reactivity, thereby loosing system-level abstraction. This leads us to believe that programming synchronous reactive systems is much more intuitive in SystemJ. SystemJ further eases the programming of asynchronous systems by providing explicit syntactic statements (`send`/`receive` built on formal semantics), while SHIM implements the asynchronous concepts by binding rendezvous with C's pass-by-value and reference paradigm. With regards to data-driven processing, SHIM lags behind SystemJ as it does not yet provide support for more complex data types. Also, in our opinion, further extension of SHIM with complex data-types borrowed from C would make data-driven computations as error-prone as C itself.

Kahn Networks SystemJ's MoC is much more restrictive than Kahn networks [16]. Kahn's networks communicate using unbounded buffers, while we utilize 1-place buffer rendezvous. The unbounded buffers can be simulated in SystemJ by using a chain of buffers just like in SHIM. Adding unbounded buffers makes Kahn networks Turing-complete and thus difficult to schedule statically. Scheduling KPN dynamically is resource intensive and might not be suited for some embedded applications. The scheduling algorithm for Kahn networks [33] is complicated to implement and gives no a priori information about buffer bounds, which is a liability for resource constrained embedded systems.

Karp and Millers [34] and Lee and Messerschmitt's [35] all utilize bounded Kahn networks. Once the buffer bounds are determined, these models can be implemented using SystemJ without loss of behaviour or generality.

Purely synchronous MoC Synchronous MoCs like those represented by Esterel are completely deterministic and highly attractive since they can be directly translated into circuit design (thus easily verifiable). While these MoCs are good for representing synchronous behaviour they cannot be utilized to described GALS systems. Many practical systems like video games and server-client communications are asynchronous. While Lustre [36] and Signal [37] are able to model multi-rate data-flow, they are again unable to model GALS systems. Thus SystemJ differs from all these languages as it provides an easy way to model complex real-life synchronous and GALS systems.

Comparison with other reactive Java packages Other reactive Java packages like SugarCubes [38] provides a deterministic synchronous model of computations based on the SL synchronous language [39]. Besides providing an easier development environment which includes reactive statements as first class primitives within Java, SystemJ also differs from SugarCubes in its support for the GALS model of computation. There are three main differences between SystemJ and SugarCubes semantics and scheduling policies. SugarCubes, like the SL programming language, relaxes the synchronous hypothesis, while, signal emissions are reacted to in the same instant of time, reactions to signal absence are postponed to the next instant of time. SystemJ, like Esterel, reacts to signal presence and absence in the same instant of time. SugarCubes, being a reactive Java library/interpreter, allows synchronous parallel reactions to be added dynamically at runtime using the || operator. SystemJ currently does not support this feature, all the synchronous parallel reactions are initialised only once at compile time. SystemJ's approach, although less flexible, achieves the same program behaviour but with increased execution speed. In SystemJ logical clock-domains can be composed asynchronously a feature which is not supported by SugarCubes or any other reactive Java package [40, 41, 42, 43, 44].

11. Conclusions and Future Work

SystemJ is based on the *Globally Asynchronous Locally Synchronous* model of computation that combines the synchronous features of Esterel with rendezvous communication of CSP and data computation capabilities of Java. We provided a detailed description of the model of computation of SystemJ, as well as the micro-step semantics of the language, suited for compiler construction. SystemJ utilizes a single buffer underlying asynchronous layer for rendezvous between asynchronously coupled clock-domains. This leads to easier scheduling mechanism as compared to CRSM/CRP, and without loss of generality. Based on the provided micro-step semantics, we have developed a SystemJ compiler called Asynchronous Graph Code (AGRC). This compiler is better suited for SystemJ than the previous TReK compiler since it is based on a set of formal semantics, and thus is potentially easier to verify. It also allows us to rewrite the asynchronous *send/receive* into synchronous statements, based on the rendezvous model described in Section 5. The AGRC compiler also outperforms the TReK compiler for SystemJ and, on average it provides 67% better performance in our benchmark tests. Finally, the new AGRC compiler allows us to explore target different execution platforms, which would provide direct support for the reactive and concurrency statements within SystemJ.

Currently, SystemJ's non-deterministic model of computation is difficult to verify. In particular, the preemption on rendezvous can lead to non-deterministic programs. The micro-step semantics allow us to generate hardware circuit models directly from SystemJ. This path is very attractive since it opens a wide variety of verification and optimization techniques available in the hardware modelling domain. We plan to pursue this path in the future.

References

[1] M. Keating, P. Bricaud, Reuse Methodology Manual: for System-On-A-Chip Designs, Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[2] C. Tranoris, K. Thramboulidis, A tool supported engineering process for developing control applications, Comput. Ind. 57 (5) (2006) 462–472. http://dx.doi.org/http://dx.doi.org/10.1016/j.compind.2006.02.006 doi:http://dx.doi.org/10.1016/j.compind.2006.02.006.

[3] A. Girault, A survey of automatic distribution method for synchronous programs, in: F. Maraninchi, M. Pouzet, V. Roy (Eds.), International Workshop on Synchronous Languages, Applications and Programs, SLAP'05, ENTCS, Elsevier Science, Edinburgh, UK, 2005.

[4] M. Shapiro, Une méthode de conception progressive des systèmes parallèles utilisant le langage C.S.P, Ph.D. thesis, Institut National Polytechnique de Toulouse, E.N.S.E.E.I.H.T., Toulouse, France (September 1980).

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

[6] S. Ramesh, Communicating Reactive State Machines: Design, Model and Implementation, in: IFAC Workshop on Distributed Computer Control Systems, Pergamon Press, September, 1998.

[7] O. Tardieu, S. A. Edwards, Scheduling-independent threads and exceptions in SHIM, in: EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software, ACM Press, New York, NY, USA, 2006, pp. 142–151. `doi:http://doi.acm.org/10.1145/1176887.1176908`.

[8] T. Grotker, System Design with SystemC, Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[9] G. Berry, S. Ramesh, R. K. Shyamasundar, Communicating reactive processes, in: POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, ACM Press, New York, NY, USA, 1993, pp. 85–98. `doi:http://doi.acm.org/10.1145/158511.158526`.

[10] C. A. R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[11] N. Chandra, Verification of Communicating Reactive State Machines, Master's thesis, Indian Institute of Technology, Mumbai (January 2002).

[12] B. Rajan, R. K. Shyamasundar, Multiclock Esterel: A Reactive Framework for Asynchronous Design, in: 13th Intl. Conf. on VLSI Design, 2000, pp. 76–83.
URL `citeseer.ist.psu.edu/rajan00multiclock.html`

[13] G. Berry, Esterel v7: From Verified Formal Specification to Efficient Industrial Designs, in: Fundamental Approaches to Software Design (FASE), Vol. 3442/2005 of Lecture Notes in Computer Science, Springer-Verlag, 2005, p. 1.

[14] K. Schneider, J. Brandt, T. Schuele, A verified compiler for synchronous programs with local declarations, in: Synchronous Languages, Applications, and Programming (SLAP), Barcelona, Spain, 2004.
URL `citeseer.ist.psu.edu/article/schneider04verified.html`

[15] S. Edwards, O. Tardieu, SHIM: A Deterministic Model for Heterogeneous Embedded Systems, IEEE Transactions on Very Large Scale Integration Systems 14 (8) (2006) 854–867.

[16] G. Kahn, The Semantics of a Simple Language for Parallel Programming, in: J. L. Rosenfeld (Ed.), Information Processing '74: Proceedings of the IFIP Congress, North-Holland, New York, NY, 1974, pp. 471–475.

[17] 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, pp. 149–158.

[18] E. Lee, The problem with threads, IEEE Computer 39 (2006) 33–42.

[19] G. Berry, The semantics of pure Esterel (1993).
URL `citeseer.ist.psu.edu/berry93semantics.html`

[20] D. E. Knuth, The Art of Computer Programming, Vol. 2, Addison-Wesley, 1981.

[21] L. Yoong, P. Roop, Z. Salcic, Compiling Esterel for Distributed Execution, in: Proceedings of Workshop on Synchronous Languages, Applications and Programming (SLAP), Vienna, Austria, 2006.

[22] A. Malik, Z. Salcic, P. S. Roop, SystemJ compilation using the Tandem Virtual Machine Approach, ACM Transaction on Design Automation of Electronic Systems 14 (3) (2009) 1–37.

[23] G. D. Plotkin, A Structural Approach to Operational Semantics, Tech. Rep. DAIMI FN-19, University of Aarhus (1981).
URL `citeseer.ist.psu.edu/plotkin81structural.html`

[24] D. Potop-Butucaru, Optimisations for Faster Execution of Esterel Programs, Ph.D. thesis, Ecole des Mines de Paris (2002).

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

[26] 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, pp. 1–8.

[27] E. des Mines de Paris, ARMINES, INRIA, Esterel Compiler v5_92 (Last Access: 15/01/2009).
URL `http:www.sop-inria.fr/esterel.org/`

[28] S. Edwards, An Esterel compiler for large control-dominated systems., IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21 (2) (2002) 169–183.

[29] A. Malik, SystemJ Benchmark Suite (Last Access: 22/6/2009).
URL `http://www.ece.auckland.ac.nz/~amal029`

[30] S. Edwards, Estbench Esterel Benchmark Suite (Last Accesses: 22/09/2006).
URL `http://ww1.cs.columbia.edu/~sedwards/software.html`

[31] I. Radojevic, Z. Salcic, P. Roop, Modelling heterogeneous embedded systems using SystemC and Esterel: A comparitive study, IEEE Design and Test of Computers 23(5) (2006) 348–358.

[32] L. Lavagno, E. Sentovich, Ecl: a specification environment for system-level design, in: DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation, ACM, New York, NY, USA, 1999, pp. 511–516. `doi:http://doi.acm.org/10.1145/309847.309989`.

[33] T. Parks, Bounded scheduling of process networks, in: Ph.D. dissertation, University of California, Berkeley, 1995.

[34] R. Karp, R. Miller, Properties of a model for parallel computations: Determinancy, termination, and queueing, SIAM Journal of Applied Mathematics 14 (6) (1966) 1390–1411.

[35] E. A. Lee, D. G. Messerschmitt, Synchronous data flow, Proceedings of the IEEE 75 (9) (1987) 1235–1245.

[36] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous dataflow Programming Language LUSTRE, Proceedings of the IEEE 79 (9) (1991) 1305–1320.
URL `citeseer.ist.psu.edu/halbwachs91synchronous.html`

[37] P. LeGuernic, T. Gautier, M. Le Borgne, C. Le Marie, Programming real-time applications with SIGNAL, Proceedings of the IEEE 79 (9) (1991) 1321–1336.

[38] F. Boussinot, J.-F. Susini, The sugarCubes tool box: a reactive Java framework, Software Practices Experience 28 (14) (1998) 1531–1550.

[39] F. Boussinot, R. de Simone, The SL Synchronous Language, IEEE Transactions on Software Engineering 22 (4) (1996) 256–266. `doi:http://dx.doi.org/10.1109/32.491649`.

[40] L. Hazard, J.-F. Susini, F. Boussinot, The Junior Reactive Kernel, Tech. rep., INRIA, Research Report 3732 (July 1999).

[41] L. Mandel, M. Pouzet, ReactiveML: a reactive extension to ML, in: PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming, ACM, New York, NY, USA, 2005, pp. 82–93. `doi:http://doi.acm.org/10.1145/1069774.1069782`.

[42] S. Edwards, L. Lavagno, E. A. Lee, A. Sangiovanni-Vincentelli, Design of Embedded Systems: Formal Models, Validation, and Synthesis, Proceedings of the IEEE 85 (1997) 366–390.
URL `citeseer.ist.psu.edu/article/edwards97design.html`

[43] M. Antonotti, A. Ferrari, A. Flesca, A. Sangiovanni-Vincentelli, JESTER: An Esterel-based Reactive Java Extension for Reactive Embedded Systems, in: Forum on specification & Design Languages, 2000.

[44] C. Passerone, C. Sansoe, L. Lavagno, R. McGeer, J. Martin, R. Passerone, A. Sangiovanni-Vincentelli, Modelling reactive systems in Java, ACM Trans. Des. Autom. Electron. Syst. 3 (4) (1998) 515–523.

APPENDIX

A. Asynchronous reactive semantics

This appendix presents the rewrite rules for the asynchronous reactive statements. The *data* and *data*$'$ decorations are omitted when dealing with purely reactive statements.

A.1. Channel start rule

$$\bullet\,\texttt{input channel}\ C\ \bar{p}, data \xrightarrow[E,E_c\cup C[S_{rc}]]{\perp,\perp} \texttt{input channel}\ C[S_{rc} \leftarrow 0], data' \bullet \bar{p} \tag{2a}$$

$$\bullet\texttt{output channel}\ C\ \bar{p}, data \xrightarrow[E,E_c\cup C[S_{wc}]]{\perp,\perp} \texttt{output channel}\ C[S_{rc} \leftarrow 0], data' \bullet \bar{p} \tag{2b}$$

A.2. Channel resumption rule

$$\frac{E_c[Cw_s] \neq E_c[Cr_r], E_c[Cp_s] = E_c[Cp_r]}{\bullet\texttt{input channel}\ C\ \bar{p} \xrightarrow[E,E_c]{\perp,\perp} \texttt{input channel}\ Cr_r \leftarrow Cw_s \bullet \bar{p}} \tag{3a}$$

$$\frac{E_c[Cw_r] \neq E_c[Cr_s], E_c[Cp_s] = E_c[Cp_r]}{\bullet\texttt{output channel}\ C\ \bar{p} \xrightarrow[E,E_c]{\perp,\perp} \texttt{output channel}\ Cw_r \leftarrow Cr_s \bullet \bar{p}} \tag{3b}$$

A.3. Asynchronous operator start rule

$$\bullet\,(p >< q), data \xrightarrow[E,E_c]{\perp,\perp} (\bullet p^{\perp_p}) >< ({}^{\perp_q}\bullet q), data \tag{4}$$

where p and q are two clock-domains being forked.

A.4. Clock-domain completion and resumption rules

$$\bar{p}^k, data \xrightarrow[E,E_c]{k,S_{set}\ k\in\{0,1\}} \bar{p}, data \tag{5a}$$

$$\bullet\,(\hat{p}), data \xrightarrow[E,E_c]{\perp,\perp} \bullet\bar{p}^{\perp}, data \tag{5b}$$

A.5. Rendezvous macro-step rule

$$\frac{E_c[Cp_s] = E_c[Cp_r]\ \ ,\ E_c[Cr_r] > E_c[Cr_s]}{\{\bullet\hat{p}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} p, data'\}, \{\bullet\hat{q}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} q, data'\}} \tag{6}$$

where p and q are two rendezvousing partner clock-domains.

A.6. Send start rule

$$\bullet\,\texttt{send}\ C, data \xrightarrow[E,E_c]{1,\perp} \widehat{send}\ Cw_s \leftarrow Cw_s + 1, data' \tag{7}$$

A.7. Send resumption rules

$$\frac{E_c[Cw_s] = E_c[Cw_r]}{\bullet\widehat{\texttt{send}\ C}, data \xrightarrow[E,E_c]{0,\perp} \texttt{send}\ C, data} \tag{8a}$$

$$\frac{E_c[Cw_s] \neq E_c[Cw_r]}{\bullet\widehat{\texttt{send}\ C}, data \xrightarrow[E,E_c]{1,\perp} \widehat{\texttt{send}}\ C, data} \tag{8b}$$

A.8. Receive start rule

$$\frac{E_c[Cr_r] > E_c[Cr_s]}{\bullet\texttt{receive}\ C \xrightarrow[E,E_c]{1,\bot} \widehat{\texttt{receive}}\ Cr_s \leftarrow Cr_s + 1, data} \tag{9a}$$

$$\frac{E_c[Cr_r] = E_c[Cr_s]}{\bullet\texttt{receive}\ C, data \xrightarrow[E,E_c]{1,\bot} \widehat{\texttt{receive}}\ C, data} \tag{9b}$$

A.9. Receive resumption rules

$$\frac{E_c[Cr_r] > E_c[Cr_s]}{\bullet\widehat{\texttt{receive}}\ C \xrightarrow[E,E_c]{1,\bot} \widehat{\texttt{receive}}\ Cr_s \leftarrow Cr_s + 1, data} \tag{10a}$$

$$\frac{E_c[Cr_r] = E_c[Cr_s]}{\bullet\widehat{\texttt{receive}}\ C, data \xrightarrow[E,E_c]{1,\bot} \widehat{\texttt{receive}}\ C, data} \tag{10b}$$

$$\bullet\ \widehat{\texttt{receive}}\ C, data \xrightarrow[E,E_c]{0,\bot} \texttt{receive}\ C, data' \tag{10c}$$

A.10. Rendezvous preemption rule

$$\bullet\ \bar{p}, data \xrightarrow[E,E_c]{0,\bot} \bar{p}, data', reset(E_{pc} \subseteq E_c) \tag{11}$$

A.11. Rendezvous preemption response rules

$$\frac{E_c[Cp_r] > E_c[Cp_s]}{\bullet\overline{\texttt{send}}, data \xrightarrow[E,E_c]{0,\bot} \texttt{send}, data', reset(E_{pc} \subseteq E_c)} \tag{12a}$$

$$\frac{E_c[Cp_s] > E_c[Cp_r]}{\bullet\overline{\texttt{receive}}, data \xrightarrow[E,E_c]{0,\bot} \texttt{receive}, data', reset(E_{pc} \subseteq E_c)} \tag{12b}$$

A.12. The reset *function*

$$\frac{E_c[Cp_r] > E_c[Cp_s]}{Cw_s \leftarrow 0, Cw_r \leftarrow 0, Cp_s \leftarrow Cp_s + 1, Cv \leftarrow 0} \tag{13a}$$

$$\frac{E_c[Cp_s] > E_c[Cp_r]}{Cr_s \leftarrow 0, Cr_r \leftarrow 0, Cp_r \leftarrow Cp_r + 1, Cv \leftarrow 0} \tag{13b}$$