The DSystemJ programming language for dynamic GALS systems: it's semantics, compilation, implementation, and run-time system

Avinash Malik, Alain Girault, Zoran Salcic

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

The DSystemJ programming language for dynamic GALS systems: it's semantics, compilation, implementation, and run-time system

Avinash Malik — Alain Girault — Zoran Salcic

N° 7346

July 22, 2010

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--7346--FR+ENG

The DSystemJ programming language for dynamic GALS systems: it's semantics, compilation, implementation, and run-time system

Avinash Malik * , Alain Girault † , Zoran Salcic ‡

Thème COM — Systèmes communicants
Projets POP ART

Rapport de recherche n° 7346 — July 22, 2010 — 32 pages

Abstract:

The paper presents a programming language called DSystemJ, for dynamic distributed Globally Asynchronous Locally Synchronous systems (GALS), its formal model, formal syntax and semantics, its compilation and implementation. The language is aimed at dynamic distributed systems, which use socket based communication protocols for communicating between components. DSystemJ allows the creation and control at runtime of asynchronous processes called clock-domains, their mobility on a distributed execution platform, as well as the runtime reconfiguration of the system's functionality and topology. As DSystemJ is based on a GALS model of computation and has formal semantics, it offers very safe mechanisms for implementation of dynamic distributed systems and potential for their formal verification. The principles and details of DSystemJ's compilation, as well as its required runtime support are described. The runtime support is itself implemented in the SystemJ GALS language, which can be considered as a static subset of DSystemJ.

Key-words: GALS systems, distributed programming, dynamic reconfiguration, formal model of computation, semantics, CSP, π-calculus, SystemJ, DSystemJ.

* INRIA Grenoble Rhône-Alpes, POP ART project team, France, `Avinash.Malik@inria.fr`
† INRIA Grenoble Rhône-Alpes, POP ART project team, France, `Alain.Girault@inrialpes.fr`
‡ University of Auckland, New Zealand, `z.salcic@auckland.ac.nz`

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Le langage de programmation DSystemJ pour les systèmes GALS dynamiques: sémantique, compilation, mise en œuvre et exécution

Résumé :

Cet article présente un nouveau langage de programmation appelé DSystemJ, destiné aux systèmes répartis dynamiques Globalement Asynchrones Localement Synchrones (GALS), ainsi que son modèle formel de calcul, sa syntaxe et sa sémantique formelle, sa compilation et sa mise en œuvre. Le langage est destiné à la conception des systèmes répartis dynamiques, qui utilisent des protocoles de communication basés sur les sockets. DSystemJ permet la création et le contrôle durant l'exécution de processus asynchrones appelés clock-domains, leur mobilité sur des plateformes d'exécution répartie, ainsi que la reconfiguration à l'exécution des fonctionnalités du système et de sa topologie. Puisque le modèle formel de calcul de DSystemJ est basé sur le modèle GALS et possède une sémantique formelle, il offre des mécanismes très sûrs pour la mise en œuvre de systèmes dynamiques répartis et le potentiel pour leur vérification formelle. Nous donnons les principes et les détails de la compilation de DSystemJ ainsi que son environnement de support à l'exécution. Cet environnement de support est lui-même mis en œuvre dans le langage GALS SystemJ, qui peut être considéré comme un sous-ensemble statique de DSystemJ.

Mots-clés : systèmes GALS, programmation répartie, reconfiguration dynamique, modèle formel de calcul, sémantique, CSP, π-calcul, SystemJ, DSystemJ.

1 Introduction

An increasing number of computing applications are concurrent in nature and the only way to describe them efficiently is to use new concurrent programming languages that allow explicit use of concurrency. Sometimes, concurrency is a natural way to describe system operation, while in other cases, it is natural due to the nature of the execution platform (e.g., distributed networked systems). Some typical examples of such applications are sensor networks capable of dealing with nodes being attached or detached *at runtime*, and ad-hoc collaborative systems in which participants *dynamically* enter and exit from a joint activity (such as multi-player gaming, or document editing environments).

We define a superset of distributed systems that we target, called *dynamic distributed systems* (DDS), capable of creating, terminating, migrating, and managing processes at runtime in a distributed environment. The programming languages have not kept pace with this increase in demand of concurrent applications. In fact, concurrent programming with standard languages is still considered difficult [12]. The main reason for this difficulty arises from the fact that the concurrent programming advocated by standard languages requires programmers to deal with synchronization and communication between concurrent processes at a very low level of abstraction, thus diverting them from the actual system design.

Over the years a number of programming techniques and languages have been proposed to make concurrent and especially distributed programming more productive. The *de facto* standard for distributed computing is the *Message Passing Interface* (MPI) specification [21]. MPI is usually implemented as a library providing an *Application Programming Interface*, which can be used from different languages. Other approaches include mobile agent systems, like JADE [4] based on Java, which are specifically designed to take advantage of Java's portability. Yet these runtime libraries are often very heavy in terms of memory footprint and resource requirements, thereby making them unsuitable for systems with less powerful computing nodes, like those used in sensor networks.

All the above mentioned approaches model systems with asynchronous concurrency without being based on a formal *Model of Computation* (MoC) and formal semantics. We believe that formal semantics is essential for faithful compilation and reasoning about the program. Formal semantics is the cornerstone for state space exploration techniques [6], which can be used for formal verification – an important step in building trustworthy, highly reliable systems. More generally, we advocate that formal semantics is essential in languages that are used to describe DDSs, as it offers the potential to reason about the correctness of such complex systems.

A number of formal languages, equipped with a formal semantics, have been introduced, like *Communicating Sequential Processes* [11] (e.g., Occam [9]), π-calculus [17] (e.g., Occam-π [24]), Join-calculus [8] (e.g. JoCaml [15]), and Actor Models [7] (e.g., ActorFoundry [1]). All these approaches have some merits and some disadvantages. Occam and Occam-π are able to model static and dynamic distributed systems, respectively, but they are unable to express complex data transformations and cannot abstractly express data fusion from multiple sources.

JoCaml is able to model DDSs with complex data transformations, thanks to the ML programming language as its base. However, it is unable to express data fusion abstractly. Also, unlike Occam-π it does not allow mobility of processes at runtime.

Finally, actor based languages and libraries, like ActorFoundry, Scala [19], and Erlang [23], are not designed to execute efficiently on distributed systems or lack a number of the above mentioned capabilities.

In this paper, we introduce a new programming language aimed at DDSs communicating via socket based networks, called DSystemJ. It is a conservative extension of the *Globally Asynchronous Locally Synchronous* (GALS) language SystemJ [14]. DSystemJ extends SystemJ with new features to deal with dynamics of asynchronous processes, called clock domains. Like SystemJ, it also allows each asynchronous process to be expressed as a composition of multiple synchronous concurrent processes, called reactions. Thanks to its GALS MoC, DSystemJ is able to model a larger class of systems than any of the above mentioned approaches. DSystemJ allows the designer to create new clock-domain (CD) at runtime (dynamic creation), it provides convenient means to describe weak CD mobility, it provides an abstract means to describe data fusion, reactive programs, and complex control situations, while at the same time mixing them with complex data computations in standard Java. Finally, DSystemJ's implementation of communication between reactions in different CDs is based on CSP-style rendezvous. This is designed to work in a fully distributed memory environment, without a single entity having the complete knowledge of the system. Thus, systems implemented using DSystemJ adhere to the principle of no single point of failure.

After a brief presentation of the language itself, we focus on the key aspects of language, compilation, and implementation, including the description of the runtime environment necessary to support the dynamic nature of the language.

The rest of the paper is organized as follows: Section 2 gives an example DSystemJ program, highlighting the main features of the language. Section 3 presents the DSystemJ MoC, syntax, and intuitive semantics. The formal semantics and MoC are presented next in Section 4. Section 5 explains the compilation procedure. Section 6 provides an overview of the DSystemJ runtime system and associated libraries. A detailed comparison between DSystemJ and currently available languages and libraries is provided in Section 7. Section 8 gives the quantitative comparison between DSystemJ and JADE. Finally, we end the paper in Section 9 with the conclusions and future work directions.

2 Language Features and Example

In this section, we present a DSystemJ example that highlights all the important features of the language and familiarizes the reader with the syntax and semantics of DSystemJ.

Listing 1: A dynamic security surveillance system

```
1  system{
2   interface{
3   //The signals and channels that are used for
```

```
4     //communication with the environment and between the various CDs.
5      input Object channel askA,askB,receiveA,receiveB;
6      output Object channel askA,askB,receiveA,receiveB;
7      input String signal attach,ctrlA,ctrlB,killBDone,killB;
8      output String signal attachMessage,ctrlMessageA,ctrlMessageB;
9     }
10    {
11    //The ruby GUI listener CD
12     GUIListener->GUIListener(askA,askB,receiveA,receiveB,attach,
13     killBDone)
14     ><
15     //CD server listener
16     serverListener->{
17      {
18      //Wait to get a message on channel askA,
19      //send the CD camAController on channel receiveA.
20       while(true){
21        receive askA; send receiveA(camAController);
22       }
23      }||
24      {
25      //Wait to get a message on channel askB,
26      //send the CD camBController on channel receiveB.
27       while(true){receive askB; send receiveB(camBController);}
28      }}
29     ><
30     //The camera A controller
31     camAController->{
32     //Alternatively move camera A left and right
33     int counterA=1;
34      while(true){
35       await(ctrlA);
36          if(counterA%2 == 0) emit ctrlMessageA(``move A left'');
37          else emit ctrlMessageA(``move A right'');
38          ++counterA; pause;}}
39     ><
40     //The camera B controller
41     camBController->{
42     //Alternatively move camera B right and left
43      int counterB=1;
44      abort(killB){
45       while(true){
46        await(ctrlB);
47        if(counterB%2 == 0) emit ctrlMessageB(``move B right'');
48        else emit ctrlMessageB(``move B left'');
49        ++counterB; pause;}}
50      emit ctrlMessageB(``B killed'');}}
51    }
```

Listing 2: The GUI listener CD

```
1    reaction GUIListener(output Object channel askA, output Object
2     channel askB, input Object channel receiveA, input Object channel
3     receiveB, input String signal attach, input String signal killBDone){
4     boolean aDone=false,bDone=false;
5     {
6     while(true){
7      await(attach);
8      String name = ((String)#attach);
9      //Asked to attach controller for camera A?
10     if(name.equals("ATTACH_A")){
11      //Is camera A controller already attached in runtime?
12      if(Helper.exists("camControl.camAController") && aDone)
```

```
13        emit attachMessage(``A exists'');
14       //Get the camera A controller from the server.
15       else{
16        send askA("camControl.camAController"); //The fully qualified name
17        pause;
18        receive receiveA; //Received the camera controller A CD
19        run #receiveA();
20        aDone=true;
21        emit attachMessage(``A is now controllable'');}}
22      //Same as for A
23      else if(name.equals(``ATTACH_B''))){
24       if(Helper.exists("camControl.camBController") && bDone)
25        emit attachMessage(``B exists'');
26       else{
27        send askB(``camControl.camAController``);
28        pause;
29        receive receiveB;
30        run #receiveB();
31        bDone=true;
32        emit attachMessage(''B is now controllable``);
33       }}}
34     }||
35     {
36     while(true){
37       //Get the Kill B signal from GUI
38       await(killBDone);
39       bDone=false; //Set the B killed boolean to false
40       emit attachMessage(``B killed'');
41       pause;}}
42   }
```

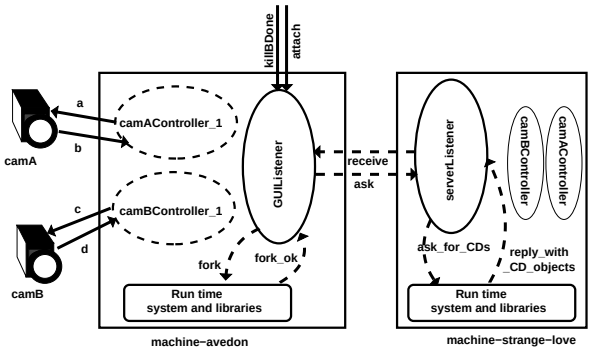

Figure 1: Pictorial representation of a security surveillance system. The thick solid ellipses are CDs initialized at the start of the system. The thin solid ellipses are CDs present in the system as code but not yet running. The dashed ellipses are instantiations of the CD code forked at runtime.

Figure 1 gives a graphical illustration of a security surveillance system. Listings 1 and 2 show the DSystemJ code implementing this system. The system consists of two Internet enabled PTZ cameras, camA and camB, being controlled by two CDs, `camAController` and `camBController`. The cameras are attached to a physical machine called `avedon`, while

all the camera controller code is present on another machine called `strange-love`, which also acts as the code repository. The `GUIListener` CD receives commands input by the user via a GUI written in the Ruby programming language, and launches/kills the camera controller CDs as requested. It also communicates with the `serverListener` CD running on `strange-love`, to obtain the new code for the camera controllers if they are not already present on `avedon`.

To keep the example relatively simple, the camera controller CDs move the cameras left and right alternately (Listing 1 lines 31-38 and 41-50). The DSystemJ code shown in the listings is well commented. The readers should note the level of abstraction – the ease of communication between reactions of asynchronous CDs using channels (Listing 1 line 21 and Listing 2 line 16), the data fusion constructs (`await`, `abort`, `signal`, etc) for communicating with the environment, including programs in various other programming languages (e.g., communicating with a Ruby program, Listing 2 lines 7, 38), the intermixing of complex data transformations in Java with DSystemJ's control flow, the ease of dynamic CD creation (Listing 2 lines 19, 30), the ease of describing weak mobility of CDs via channels (Listing 1 line 21, 27 and Listing 2 lines 18, 29), and finally, describing asynchronous (`><`) and synchronous (`||`) concurrency (Listing 1 lines 14, 23, etc). Our main goal in designing DSystemJ was to provide a mix of high-level features to build GALS systems, coupled with all the data-handling features of Java, and at the same time offering a formal semantics.

3 DSystemJ: Model of computation, syntax and intuitive semantics

DSystemJ is a conservative extension of SystemJ [14] and hence it follows the GALS MoC of the SystemJ language.

A SystemJ program consists of a set of CDs composed using the asynchronous parallel operator (><) and executing at unrelated logical clock ticks (from here on referred to as tick). CDs synchronize and communicate with each other using channels, with CSP [11] style rendezvous for synchronization and data transfer. Each CD itself consists of one or more processes, called reactions, executing in lockstep, i.e., at the CD's tick. The reactions are combined and controlled using the synchronous parallel operator (||). Reactions within the same CD communicate using the synchronous broadcast mechanism over signals. Finally, a SystemJ program interacts with its environment through a set of input and output signals. The synchronous statements, reactions, and operations on signals, and asynchronous statements, CDs, and channels, are together responsible for the control-flow of SystemJ programs. The data-driven computations and transformations are written in Java.

A DSystemJ program extends this with the ability to fork new CDs at runtime (dynamic process creation) and pass CDs over channels (process mobility).

3.1 DSystemJ syntax

DSystemJ combines features from Esterel [5], CSP [11], and π-calculus [17] with the Java programming language. Table 1 and 2 show the SystemJ and DSystemJ kernel statements and their meaning, respectively.

Table 1: The SystemJ kernel statements and their meaning

Kernel Statements	Meaning
[input] [output] [type] signal S	declare signal S
emit S [(value)]	broadcast signal S
present (S) {p} else {q}	do p if S is present, else do q
abort (S) {p}	preempt program p if S is present
suspend (S){p}	suspend p if S is present
trap (T){p. . . exit T. . . }	preempt p if exit is executed
p||q	run p and q in lock-step
p><q	run p and q asynchronously
send C([value])	send a value through C
receive C()	receive a value through C
pause	finish a logical time instant

Table 2: The kernel statements introduced in DSystemJ and their meaning

Kernel Statements	Meaning
unique-name → CD([args])	declare a named CD closure
unique-name → {}	declare an unnamed CD closure
run unique-name([args])	run CD unique-name
run #channel-name([args])	run CD received via channel channel-name
[input] [output] [type] channel C	declare channel C

Signals are the most basic means of communication in a DSystemJ program; they have a status and possibly a value. Signals can be either local or interface signals, interface signals are qualified with either the `input` or the `output` keywords and are used for communication with the environment, while local signals are used for communication between concurrent reactions within a single CD. A signal emission broadcasts the signal throughout its CD, making it instantaneously visible to all the reactions running in lock-step within that CD. The emission of an `output` signal makes it visible to the environment, too. A signal emission can be pure or include a value, which can be of any Java data type. The `present` instruction is used to check the presence of a signal, while `abort` and `suspend` instructions are used for preemption. The `trap` and `exit` statements together implement user defined preemptions, as opposed to environment based ones through signals (`abort`, `suspend`) similar to Esterel [5].

There are multiple ways to represent concurrency in DSystemJ. The || and >< operators initialize synchronous parallel reactions and CDs, respectively at program startup. The `run` statement allows designers to instantiate new CDs at runtime. The channels are used to communicate between reactions in different CDs. The `send` and `receive` statements together implement a rendezvous (blocking sending and receiving) style communication through channels. Finally, the `pause` statement shows the completion of a tick of a

single CD or a reaction. At the start of the tick of each CD, the CD's input signals are sampled from the environment by the program; then the required transitions are computed, and, finally, the CD's output signals are emitted to the environment at the end of the CD tick, thereby implementing a state machine. All the syntactic constructs presented in Table 1 and 2 can be freely intermixed with most of the Java constructs.

3.2 Intuitive Semantics of Kernel Statements

In this section we describe the intuitive semantics of the kernel constructs introduced in the DSystemJ language (Table 2). The reader is referred to [14] for a detailed explanation about the rest of the kernel statements (Table 1).

3.2.1 The unique-name -> CD([args]) and unique-name -> {} constructs

The `->` construct builds a pointer to the named or unnamed CDs, as shown in Listing 1 (lines 16-28). The `->` delimited name ("unique-name") can then be used to fork or send the CD code via channels. These unique-names have a *global scope* in the system, i.e., they are visible to all the reactions and other CDs, including themselves. The `->` operator can also create a closure, similar to functional programming languages. For example, in Listing 1 lines 31-38, the `camAController` CD forms a *closure process*. Every closure keeps a separate copy of the enclosed variables, which can also include reactive constructs like signals. The enclosed variables are not shared amongst the closures. The enclosed variables are replaced with new values when forking CDs via the `run` statement. Finally, the `->` operator can be applied several times of the same named CD but with different arguments.

3.2.2 The `run` unique-name ([args]) and `run` #channel-name([args]) constructs

The `run` construct is used to dynamically fork CDs. The version "`run` unique-name ([args])" forks the CD already registered with the runtime system, while the version "`run` #channel-name ([args])" forks a CD received via the channel "channel-name". The `run` statement performs a *rendezvous* with the runtime system, asking it to fork the required CD. The `run` statement takes a finite number of ticks to succeed, but the number of ticks cannot be statically determined (in advance). The "tick" here refers to the logical tick of the CD that invokes the `run` statement. Any CD forked via the `run` statement starts from its initial state, i.e., DSystemJ does not allow one to save the current state of a forked CD and hence, only weak mobility is possible.

3.2.3 The `send` C(unique-name) and `receive` C constructs

The `send` statement in DSystemJ is similar to the `send` statement in SystemJ. It performs a rendezvous with the `receive` statement on the same channel-name. In SystemJ, the `send`

and `receive` statements can pass any Java object. DSystemJ provides the syntactic sugar of being able to pass the CD's unique-name itself to implement CD mobility, rather than manually constructing a Java object containing the marshalled CD code, to implement the weak mobility of CD.

The major difference between DSystemJ and SystemJ rendezvous communication stems from the fact that DSystemJ rendezvous communication is not point-to-point (linear). Indeed, DSystemJ allows one to many (single sender-multiple receivers), many to one (multiple senders-single receiver), and many to many (multiple senders and receivers) rendezvous between multiple participants. Listing 3 further illustrates this point.

Listing 3: An example of non-linear channel communication in DSystemJ

```
1  //Example of Many to One
2  //non−linear channel communication on channel ''M''.
3  //Recall that >< is the asynchronous operator
4
5
6  //CD P running on machine 1 sends itself via channel C
7  //In parallel it also sends values via channel M
8  P −> {{send C(P);} || { while(true) send M(4);}}
9  ><
10 //CD Q running on machine 2 gets the value via channel M
11 Q −> {while(true) receive M;}
12 ><
13 // CD R running on machine−3
14 //forks CD P obtained via channel C
15 //and finishes itself. But, now CD P
16 //runs on machine−3 as well, blocking
17 //on channel C due to lack of a receiver and also sending values on
18 //channel M
19 R −>{receive C; run #C();}
```

In the multi-participant case, the DSystemJ runtime *non-deterministically* chooses a parter to rendezvous with, similar to the `select` statement in ADA [10]. The non deterministic selection of a rendezvous partner in a multi-participant scenario raises fairness issues. Indeed, multi-participant rendezvous can introduce starvation in a system. For example, in Listing 3 above, the `receive` statement might always choose to rendezvous with the `send` on machine 1, thereby starving the CD on machine 3.

The DSystemJ compiler is able to statically detect only some starvation situations in multi-participant rendezvous [13]. As DSystemJ is targeted towards DDSs with the goal of *no single point of failure*, there is no single entity in the system having the complete knowledge of the system at runtime. As a result, DSystemJ does not guarantee process fairness. Instead, the developer is advised to use separate channels by creating them at runtime thanks to the `channel` construct. To avoid such problems, DSystemJ might allow, in the future, multiple communication alternatives, such as: (1) rendezvous in a distributed environment and (2) join calculus based communication, which allows one to combine multiple sent and received values using combinator functions [8], in a small subnet, or single machine implementation where data delivery time between senders and receivers can be bounded.

4 Formal semantics

This section presents the formal semantics and the MoC of DSystemJ. Both are described in terms of the SystemJ MoC and micro-step semantics. We first summarise the important micro-step semantic rules of SystemJ on which DSystemJ is based.

4.1 Semantics of SystemJ

All of SystemJ's constructs utilize a structural translation scheme. This translation scheme allows us to obtain a direct intermediate representation of the program from which back-end code can be efficiently generated. The semantical rewrite rules are very fine grained, being targeted towards compiler construction, so, we call them micro-step kernel semantics.

Let p be a SystemJ kernel statement, we write

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

where $term(p)$ 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 the transition, and if none are emitted then it takes the value $\perp$. Term $data$ represents the value stores attached to the statement p before the transition, and $data'$ after the transition. Term k represents the termination code. It has a value of $\perp$, (i.e., unknown), if p does not generate a termination code after this transition, otherwise, an integer value within $[0, \infty]$. A termination code of 0 represents the instantaneous termination of a reaction; 1 represents the fact that the reaction has reached the end of its tick; a termination code in the interval $[2, \infty)$ is reserved for preemptions based on `trap/exit` constructs; and finally, a termination code of ∞ shows an unresolved signal dependency. Such termination codes are inspired by Esterel and the reader is referred to [5] and [25] for complete details of the termination codes.

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 $2 * n$ number of input/output ports, corresponding to the receiving and sending ends, respectively. Thus, the cardinality of set E_c is $3 * 2 * n$. For a channel C, we write, $E_c = \{\{Cw_s, Cw_r, Cp_s\}, \{Cr_s, Cr_r, Cp_r\}\}$, where the set's elements represent the output and input channel port statuses. In the transition rules, for brevity, we use the array indexing notation to refer to the channel and signal statuses: $E[Cw_s]$ represents the output channel port's write-sent status w_s, i.e., $Cw_s \in E_c$, while w_r and p_s are the write-received and preemption statuses, respectively. Similarly for the input channel port, r_s, r_r, and p_r, represent the read-sent, read-received, and preemption statuses, respectively. These channel statuses are used to carry out a full handshake when communicating via channels.

A statement p is also said to be *selected* iff a pause is hit during its execution. A *selected* statement is further decorated with a hat, e.g., $\hat{p}$. We use the notation $\bar{p}$ to indicate that the selected state for term p is currently unknown. The $\bullet$ represents the control point movement

in the SystemJ program and a $\ddot{p}$ indicates that the position of the current control point over p is unknown. We refer the reader to [20] for the full definitions of $\hat{p}$, $\bar{p}$, and $\ddot{p}$.

The rules below show the micro-step semantics of `pause` execution. When a `pause` is hit for the first time (*also called the start rule*) the statement gets selected and the program ends with a termination code of 1. In the next instant (*also called the resumption rule*), the selected statement continues further and completes its execution (termination 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 control statement:

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

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

There are a number of other rewrite rules associated with reactive constructs of SystemJ, which are out of the scope of this article (see [14] for the complete set of rewrite rules).

The macro-step transition rule for a SystemJ system is expressed in terms of the macro-step transition of individual CDs, which in turn is formed by combining the micro-step rules for all the SystemJ constructs. The macro-step transition for a SystemJ system is:

$$\bullet\bar{s_1} \overset{e_{s_1},k_{s_1}}{\underset{E_{s_1},E_{cs_1}}{\hookrightarrow}} \ddot{\bar{s}}_1, \quad \bullet\bar{s_2} \overset{e_{s_2},k_{s_2}}{\underset{E_{s_2},E_{cs_2}}{\hookrightarrow}} \ddot{\bar{s}}_2 \quad \ldots \quad \bullet\bar{s_m} \overset{e_{s_m},k_{s_m}}{\underset{E_{s_m},E_{cs_m}}{\hookrightarrow}} \ddot{\bar{s}}_m,$$

where s_m is some CD and E_{s_m}, E_{cs_m}, e_{s_m}, and k_{s_m} are the signal sensitivity set, channel status sensitivity set, output signal set, and termination code for CD s_m, respectively.

4.2 Semantics of DSystemJ

Before describing the micro-step rewrite rules for the DSystemJ syntactic constructs, we first present the equivalence between the DSystemJ and SystemJ MoCs, i.e., we define a dynamic DSystemJ program in terms of a static SystemJ program.

4.2.1 DSystemJ: Formal MoC

A DSystemJ program composed of CDs $d_1 \ldots d_m$ and a SystemJ system composed of CD $s_1 \ldots s_m$ are considered equivalent iff:

$$\bullet\bar{d_1} \overset{e_{d_1},k_{d_1}}{\underset{E_{d_1},E_{cd_1}}{\hookrightarrow}} \ddot{\bar{d}}_1, \quad \bullet\bar{d_2} \overset{e_{d_2},k_{d_2}}{\underset{E_{d_2},E_{cd_2}}{\hookrightarrow}} \ddot{\bar{d}}_2 \quad \ldots \quad \bullet\bar{d_m} \overset{e_{d_m},k_{d_m}}{\underset{E_{d_m},E_{cd_m}}{\hookrightarrow}} \ddot{\bar{d}}_m,$$

$$\bullet\bar{s_1} \overset{e_{s_1},k_{s_1}}{\underset{E_{s_1},E_{cs_1}}{\hookrightarrow}} \ddot{\bar{s}}_1, \quad \bullet\bar{s_2} \overset{e_{s_2},k_{s_2}}{\underset{E_{s_2},E_{cs_2}}{\hookrightarrow}} \ddot{\bar{s}}_2 \quad \ldots \quad \bullet\bar{s_n} \overset{e_{s_n},k_{s_n}}{\underset{E_{s_n},E_{cs_n}}{\hookrightarrow}} \ddot{\bar{s}}_n,$$

$$and\ \bar{d_1} = \bar{s_1}, \bar{d_2} = \bar{s_2}, \ldots \bar{d_m} = \bar{s_n}$$

$$E_{d_1} = E_{s_1}, E_{d_2} = E_{s_2}, \ldots E_{d_m} = E_{s_n}$$
$$E_{cd_1} = E_{cs_1}, E_{cd_2} = E_{cs_2}, \ldots E_{cd_m} = E_{cs_n}$$
$$e_{d_1} = e_{s_1}, e_{d_2} = e_{s_2}, \ldots e_{d_m} = e_{s_n}$$
$$k_{d_1} = k_{s_1}, k_{d_2} = k_{s_2}, \ldots k_{d_m} = k_{s_n}$$
$$\ddot{d}_1 = \ddot{s}_1, \ddot{d}_2 = \ddot{s}_2, \ldots \ddot{d}_m = \ddot{s}_n$$

The above equations show the macro-step transitions (single tick reaction) for CDs $d_1, \ldots, d_m$ and $s_1, \ldots, s_n$, respectively. Thus, a DSystemJ program is equivalent to a SystemJ program if it has the same number of running CDs, $(m = n)$ and if, for every CD d_m in the DSystemJ program, there exists a CD s_n in the SystemJ program, which when given an input signal set results in an equivalent macro-step transition and produces the same output signal set as d_m.

We define equivalence over a tick only. This is because a DSystemJ program may diverge in its behaviour over an execution trace due to its ability to fork CDs at runtime.

4.2.2 Rewrite rules for DSystemJ syntactic constructs

We now describe the rewrite rules of the DSystemJ constructs presented in Table 2.

◇ The `->` construct: The `->` construct completes instantaneously with an exit code of 0 like any other instantaneous statement in SystemJ.

$$\bullet\texttt{unique-name -> \{\}} \xrightarrow[E,E_c]{0,\perp} \texttt{unique-name -> \{\}} \tag{2a}$$

$$\bullet\texttt{unique-name -> cd} \xrightarrow[E,E_c]{0,\perp} \texttt{unique-name -> cd} \tag{2b}$$

◇ The `run` construct: This statement does not have a single micro-step rule. Instead, every `run` statement is re-written into `send` and `receive` statements to perform a rendezvous with the runtime system.

Consider an executor CD p running concurrently and asynchronously with the CD q, where p is:

```
receive C; m
```

program code of some other clock-domain `CD`, and `C` being a unique named point-to-point channel between p and q respectively. As a result, a program q:

```
run CD(args); emit S
```

can be rewritten as:

```
send C(args); emit S;
```

The CDs p and q take a transition τ, which is the macro-step rendezvous transition on channel C and the state change results in p transforming into `m`, while q transforms into `emit S`. The result is a program where the clock-domain `CD` (`m`) runs in asynchronous parallel with the forking CD q after the extra transition τ.

Intuitively, the semantics of the `run` statement assumes that the above translation of every possible CD in the DSystemJ program is running but blocked on a `receive channel-name` statement, waiting for a successful rendezvous on the unique name "channel-name" before proceeding further with its program code. The `run` statement, in turn, performs a rendezvous with one of these CDs. Note that every `run` statement requires a fresh channel name `C`.

◇ The `send` and `receive` constructs: DSystemJ's `send` and `receive` constructs implement CSP [11] style message passing.

The difference with SystemJ is that we introduce the non-deterministic choice operator $\square$, which chooses one rendezvous partner in case of a non-linear rendezvous with multiple participants (see Section 3.2.3).

$$\frac{E_{c_p}[Cp_s] = E_{c_q}[Cp_r], E_{c_q}[Cr_r] > E_{c_q}[Cr_s], E_{c_r}[Cp_s] = E_{c_q}[Cp_r]}{\{\bullet\hat{p}, data\square \bullet \hat{r}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} p, data' \bullet \hat{r}, data\}, \{\bullet\hat{q}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} q, data'\}} \tag{3a}$$

$$\frac{E_{c_p}[Cp_s] = E_{c_q}[Cp_r], E_{c_q}[Cr_r] > E_{c_q}[Cr_s], E_{c_r}[Cp_s] = E_{c_q}[Cp_r]}{\{\bullet\hat{p}, data\square \bullet \hat{r}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} r, data' \bullet \hat{p}, data\}, \{\bullet\hat{q}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} q, data'\}} \tag{3b}$$

Rules (3a) and (3b) show the macro-step rendezvous transition, when the rendezvous conditions are fulfilled, for two senders and a single receiver. The $\square$ operator internally and non-deterministically chooses one of the sending CDs p, (rule (3a)) or r, (rule (3b)) to rendezvous with the receiving CD q. The other CD blocks waiting for an acknowledgement from the receiver. Further rendezvous rules are provided in [].

The rendezvous transition Rules (3a)-(3b) are valid only in the absence of strong preemptions, possible due to constructs such as an `abort`. DSystemJ' strong preemption in presence of rendezvous is similar to that of SystemJ', except, a preemption can occur even before choosing a partner in case of multi-participant rendezvous.

$$\frac{E_{c_p}[Cp_r] \neq E_{c_q}[Cp_s], E_{c_r}[Cp_r] \neq E_{c_q}[Cp_s]}{\{\ddot{\bar{p}} \overset{\perp,\perp}{\underset{E,E_c}{\rightarrow}} \ddot{\bar{p}}\}, \{\ddot{\bar{r}} \overset{\perp,\perp}{\underset{E,E_c}{\rightarrow}} \ddot{\bar{r}}\}, \{\bullet\bar{q} \overset{1,\perp}{\underset{E,E_c}{\rightarrow}} \dot{\bar{q}}\}} \tag{4a}$$

$$\frac{E_{c_p}[Cp_r] = E_{c_q}[Cp_s], E_{c_r}[Cp_r] \neq E_{c_q}[Cp_s]}{\{\bullet\bar{p} \overset{0,\perp}{\underset{E,E_c}{\rightarrow}} p\}, \{\ddot{\bar{r}} \overset{\perp,\perp}{\underset{E,E_c}{\rightarrow}} \ddot{\bar{r}}\}, \{\bullet\bar{q} \overset{0,\perp}{\underset{E,E_c}{\rightarrow}} q\}} \tag{4b}$$

$$\frac{E_{c_p}[Cp_r] \neq E_{c_q}[Cp_s], E_{c_r}[Cp_r] = E_{c_q}[Cp_s]}{\{\ddot{\bar{p}} \overset{\perp,\perp}{\underset{E,E_c}{\rightarrow}} \ddot{\bar{p}}\}, \{\bullet\bar{r} \overset{0,\perp}{\underset{E,E_c}{\rightarrow}} r\}, \{\bullet\bar{q} \overset{0,\perp}{\underset{E,E_c}{\rightarrow}} q\}} \tag{4c}$$

$$\frac{E_{c_p}[Cp_r] = E_{c_q}[Cp_s], E_{c_r}[Cp_r] = E_{c_q}[Cp_s]}{\{\bullet\bar{p} \underset{E,E_c}{\overset{0,\perp}{\rightarrow}} p\}, \{\bullet\bar{r} \underset{E,E_c}{\overset{0,\perp}{\rightarrow}} r\}, \{\bullet\bar{q} \underset{E,E_c}{\overset{0,\perp}{\rightarrow}} q\}} \quad (4d)$$

Rules (4a)-(4d) give the preemption rules in case of a single sender and multiple receivers. If the sender is preempted then it keeps on broadcasting this request until it receives at least a single reply (Rules (4a)-(4c)). If all the receivers get the message and send a reply, then no choice is made and all the receivers are preempted (Rule (4d)). Similar preemption rules apply for multiple senders and single receivers. If the preemption occurs after selection of a partner then the preemption rules from SystemJ apply [14]. These rules can be further aggregated to derive the rules for communication between multiple senders and multiple receivers. This is left as an exercise for the reader.

◇ The channel declaration construct: This construct has the same semantical rewrite rules as the SystemJ channel declaration statement [14]. The differences between the two are purely syntactic: in DSystemJ, the `input` and `output` keywords, which define the input and output ports of the channel, are optional; the DSystemJ compiler infers the type of ports. Also, unlike SystemJ, DSystemJ allows new channel declarations at runtime.

5 Compiling DSystemJ programs

The compilation of DSystemJ programs, presented in this Section along with the supporting runtime and library description in Section 6, are the major contributions of this article.

5.1 The *Asynchronous GRaph Code* (AGRC)

DSystemJ compilation follows a structural translation scheme based on the *Structural Operational Semantics* (Section 4). DSystemJ programs are compiled into a semantics preserving intermediate format, called the *Asynchronous GRaph Code* (AGRC).

An abstract AGRC for the example in Listings 1 and 2 is shown in Figure 2. AGRC nodes are used to represent the reactivity and concurrency in the DSystemJ language. The asynch-fork node (vertex joint triangles) is used to fork CDs. It is complemented with the asynch-join nodes (base joint triangles), which represent the completion of a tick of a CD. The fork node (triangles) and join-node (inverted triangles) together delineate synchronous concurrency. The switch nodes (double diamonds) and enter nodes (ellipses) are responsible for decoding and encoding the state transitions, respectively. The test-nodes (diamonds) implement signal and Java tests (e.g., `present` statements). The action-nodes (horizontal) rectangles represent the instantaneous computations, like emission and Java data transformations. Finally, the exit nodes (hexagons) encode the exit value of synchronous parallel

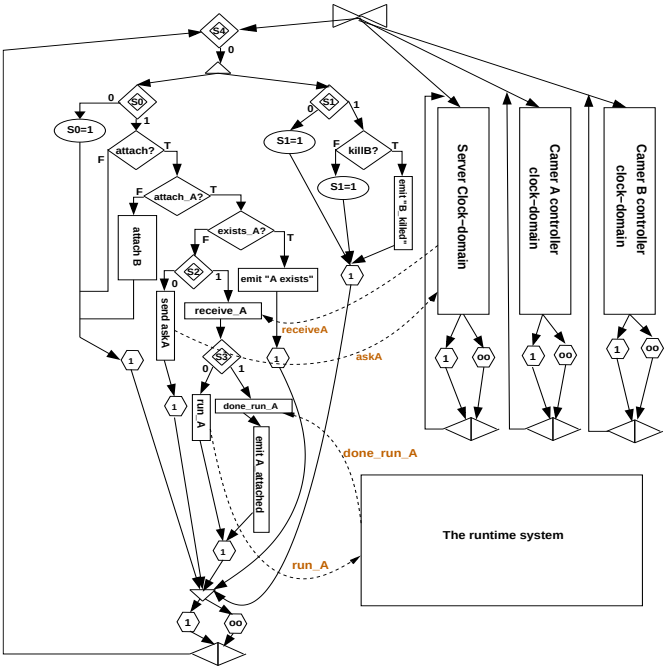

Figure 2: The AGRC for example in listings 1 and 2

reactions or CDs. Each exit node can take an integer value in the interval $[0, \infty]$ (see Section 4.1).

We now show how the AGRC captures the semantics of the DSystemJ program by traversing through the AGRC of Figure 2 as control point movements. Figure 2 is an abstract representation of Listings 1 and 2. The `GUIListener` CD is represented in some detail, while the other CDs from Listing 1 are completely abstracted out. Every CD starts with a switch node. The `GUIListener` CD decodes the switch node `S4` with a value 0, thereby entering its only child branch. Next, the fork node forks out two synchronous parallel reactions. The scheduling order of these two synchronous parallel reactions is inconsequential. Supposing that the left synchronous parallel reaction gets scheduled first, the `S0` switch node in the first synchronous parallel reaction again gets decoded to 0 and enters its left child. The enter node encodes the `S0` value as 1 and finishes execution with a exit code of 1 (indicating the completion of a tick). Similarly, the second synchronous parallel reaction completes a tick after encoding `S1` to 1. The join-node makes sure that all the incoming reactions complete with some exit code, thereby implementing the lock-step execution of the synchronous parallel reactions. Finally, the control reaches the asynch-join node, where the output signals, if any, are emitted to the environment and the new input signals are

read from the environment. This represents the end of the tick transition for each CD. After reading the new set of input signals, a new iteration of the CD in a new tick is carried out.

In the second iteration, the first synchronous parallel reaction's `S0` switch node enters its right most child, where it first checks if the `attach` signal is present. If present, a check on the value of this signal is made. If the `attach` signal asks the `GUIListener` to attach the controller for camera A, the `GUIListener` then checks if this controller is already attached or not. If so, a signal with the string "`A exists`" is emitted, otherwise, this CD rendezvous's with the `serverListener` CD via channels `askA` and `receiveA`, obtaining the code for the controller. Once the code is received, this CD rendezvous's with the runtime system via channels `run_A` and `done_run_A`, asking it to fork this recently obtained CD.

This is the expected behaviour of the `GUIListener` CD shown in Listing 2. Listing 2 first waits to obtain an `attach` signal (line 7). Once obtained, the value of this signal is checked to see which camera controller needs to be forked (lines 10, 23). If the controller for camera A needs to be forked, an enquiry is made with the runtime system to see if this controller is already present (line 12). If this check succeeds, a message "`A exists`" is emitted to the environment, otherwise, a rendezvous is carried out with the `serverListener` CD (lines 16-18), and finally the obtained camera controller code is instantiated as a CD (line 19).

In Figure 2, the dotted lines show the channel communication. Most of the DSystemJ syntactic constructs from Table 2 are directly converted into primitive nodes of the AGRC. Only the `send`, `receive`, and `run` statements do not have a direct translation to primitive AGRC nodes. The `send` and `receive` statements are rewritten into algorithms operating on channel statuses using other reactive constructs [14] (namely `await` and `emit`) to implement a rendezvous. The `run` statement is rewritten as a combination of `send` and `receive` constructs rendezvousing with the runtime system (Section 4.2.2). The runtime system itself is a GALS program responsible for CD management (see next section). The rendezvous in DSystemJ takes finite time to finish but the bound on this time cannot be computed statically, and thus, a `run` statement completes, in the process forking a CD, after a finite number of ticks of the CD that calls the `run` statement.

Listing 4: Example compiled camAController

```
1  //GenericClockDomain implements Serializable and Runnable interfaces
2  //Every CD in DSystemJ is compiled into a Java class.
3  //Every CD in DSystemJ should be serializable and extend the
4  //Java thread.
5  public class camAController implements GenericClockDomain {
6   private Signal ctrlA;
7   private int counterA=1;
8   private char S6 = 0;
9   char ends [] = new char [1]; //this vector shows the value of exit nodes.
10  //This is the Surface Flow Graph -- representing the first
11  //tick of the DSystemJ program.
12  private void SFG(){
13   S6 =1; //setting the switch variable.
14   ends[0] = 1; //shows pausing
15  }
16  private void DFG(){
17   switch(S6){
18   case 0:
```

```
19       S6 =1; ends[0] = 1;
20       break;
21      case 1:
22       if(ctrlA.getStatus()){
23       ctrlMessageA.setStatus(true);
24        if(counterA%2)==0 ctrlMessageA.setValue(``move A left'');
25        else ctrlMessageA.setValue(``move A right'');
26       }
27       break;}}
28     public void run(){
29      super.run();
30      while(true){
31       if(ends[0] == 0) break; //break loop when completed
32       boolean firstIn = true;
33       ReadInputs(ctrlA); //Read signals from environment
34       if(firstIn){ SFG(); firstIn=true;}//Call SFG method first time
35       else DFG(); //Call DFG every other time
36       EmitOuts(ctrlMessageA); //Emit signals to environment}}
37   }
```

Listing 4 shows the generated Java code for the `camAController` CD from Listing 1. Every CD is compiled into a Java class. Every compiled CD implements the `GenericClockDomain interface` (line 5). The runtime system loads the CD as a `GenericClockDomain` and launches it using the Java's `Thread.start` method. All signals and channels are compiled into Java objects, which consist of a status and a value. Every synchronous parallel reaction is compiled into a Java method. Thus, all synchronous concurrency is compiled away to create single threaded Java code (similar to other synchronous languages like Esterel); but in our case we have multiple threaded Java code, one such thread for each CD. Extra variables (lines 8-9) are generated that represent the nodes in the AGRC. These variables, especially the switch nodes, are used to implement the state machine as represented in the AGRC of the DSystemJ program.

5.2 Procedure to compile and run DSystemJ programs

Figure 3 shows the procedure to compile and run the DSystemJ security surveillance system described previously. In this example, all the CDs are written in a single file called `controller.sysj` (the DSystemJ compiler also accepts CDs in multiple files for better organization of large projects). Each CD is compiled into a single threaded Java program by the DSystemJ compiler. These Java programs are then compiled with any Java compiler to create the class files. Next, the designer describes the underlying execution system topology and configuration via one or more XML files. This description of the system topology consists of information such as the IP-addresses of the machines involved, binding of all the input/output signals to the underlying socket connections. Finally, the DSystemJ program is launched by parsing this XML description into the DSystemJ runtime system. The runtime system needs to be started beforehand on all the machines that will participate in the system. We explain in more detail all the stages presented in Figure 3 in the next sections.

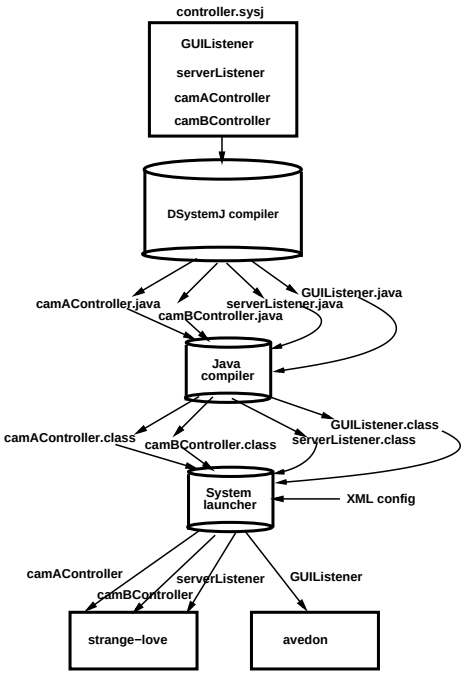

Figure 3: The compilation and runtime procedure for the DSystemJ program in Listings 1 and 2

6 Runtime system and library

The DSystemJ language compiler is accompanied with a runtime system and a library that is responsible for managing CDs. In this section we give an overview of these components, which form a part of the complete DSystemJ runtime hierarchy. This is another major contribution of this article.

6.1 The runtime system

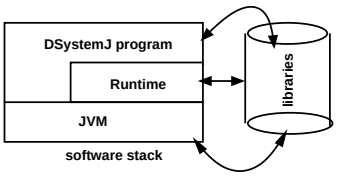

Figure 4: The DSystemJ runtime environment

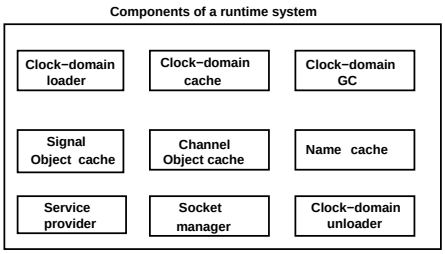

Figure 5: The various CDs implementing the runtime system

Figure 4 shows the interactions of the DSystemJ runtime system with the DSystemJ program. When compiled, the runtime system occupies only 106KB. The runtime system itself is a GALS SystemJ program with a number of dedicated CDs (Figure 5) as described below.

6.1.1 The CD loader

The CD loader is responsible for loading the CD (class file) when requested. The loader listens on a fixed socket for incoming requests to load CDs. Once the request is received, the loader searches the system class path to check if the CD code is present; if the CD is found, the loader registers the name of this CD in the name cache and loads the CD class along with all the classes it references in the CD cache. The loader also registers the signal and channel arguments, which consist of IP addresses, sockets numbers, type of sockets, type of serializers (used for marshalling), which bind the DSystemJ signals and channels to the underlying physical layers.

The designer does not necessarily need to load the CD code into the cache; instead, the designer can just register the name of the CD along with the signal and channel arguments by providing a flag `register`. In this case, the loader does not search for the CD code and just registers the name of the CD along with its associated signal and channel arguments with the name cache for later use (e.g. when forking a clock-domain at runtime).

6.1.2 The CD cache

It is the cache of CD code (classes) and its referenced Java classes that are currently registered with the name cache. The CDs present in this cache need not necessarily be instantiated and running. The latter scenario is possible in the case that the CD is killed but not yet garbage collected.

6.1.3 The signal object cache

This cache holds all the signal objects; it is a hash-map where the CDs are used as indices to access the signal objects. Each signal object itself utilizes the underlying physical layer for

communication with the environment, and hence holds references to all the communication objects like the TCP/IP and UDP servers, clients, etc. The communication objects are generic servers and clients, which are casted into concrete ones using Java's polymorphism.

6.1.4 The channel object cache

This is the cache of all the channel objects. This cache is implemented in a manner similar to the signal object cache.

6.1.5 The name cache

This cache holds the names of all the CDs that the runtime system knows about. It also holds the arguments (IP addresses, socket numbers, etc) for all the signals and channels that are known to the runtime system. Registering a CD with the runtime makes the name of the CD and any associated signal/channel arguments known to the name cache. De-registering a CD does the reverse.

6.1.6 The Service provider

It provides a number of services to the DSystemJ programmer, via the library functions. For example, providing the name of the currently registered CDs, giving a list of the signal and channel arguments, the name of the physical machine, etc.

6.1.7 The socket manager

It creates new socket ports used for communication between machines. It recycles the sockets when a CD is de-registered (killed), thereby freeing the sockets associated with the signals and channels in use by this CD. The implemented recycle algorithm is simple: It maintains a free and used list of socket numbers within the runtime system, from which sockets are allocated.

6.1.8 The CD unloader

It does the opposite of the CD loader; it unloads the CDs from the CD cache and de-registers them from the name cache. Unloading usually happens when the CD is killed or upon explicit designer request. By default the unloader calls the CD garbage collector (GC) after de-registering it. This default behaviour can be changed by the system designer to avoid runtime overhead.

6.1.9 The CD GC

It frees/de-allocates heap memory allocated to CDs and their referenced classes. The runtime uses Java's GC as the CD GC. If required, the designer can replace it with another one.

6.2 The library support

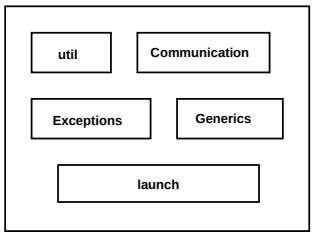

Figure 6: The DSystemJ library support

The DSystemJ library provides a number of general and some specific classes (Figure 6), which can be used by the designers to write DSystemJ programs more easily. The library is designed with the purpose of being easily extensible by designers.

6.2.1 The `util` library

It provides a number of utility functions, For example, signal and channel argument factories, serializers for marshalling purposes, helper classes providing access to the runtime name and CD cache, and so on.

6.2.2 The `Communication` library

It provides access to the TCP/IP, UDP, and multicast servers and clients, which can be used or extended by the designer to implement the communications between the DSystemJ CDs running on different physical machines, by binding them to channels or signals.

6.2.3 The `Exceptions` library

It provides a number of exceptions that are thrown by the runtime system to indicate various errors, like CD code not found, wrong CD name, etc. The exceptions can be extended by the designer to implement his/her own exception types.

6.2.4 The `Generics` library

It provides a number of generic `interfaces`, which are implemented by the classes in: (1) `communication` library, (2) CDs and (3) the signal and channel arguments. The designer must `implement` the interfaces in this library when extending the DSystemJ runtime with his/her own communication protocol or CDs.

6.2.5 The `launch` library

It is responsible for parsing the system description in XML (Figure 3), registering and launching the CDs in the runtime system in a distributed environment with multiple machines. Listing 5 gives a partial XML description of the DSystemJ example presented in Listing 2.

An XML description of the system topology has a number of advantages: firstly, the simplicity of generating and parsing of an XML description. One of our future goals is to have a graphical user interface (GUI), called a *system composer*, that will allow designers to describe the topology of large distributed DSystemJ programs conveniently. This system composer would create the XML description automatically. Secondly, an XML description can be changed and parsed into a running system, thereby changing the system configuration at runtime. Finally, there is a clear separation between the system topology and functionality, allowing the designer to change the functionality or the topology of the system at runtime without affecting each other.

Listing 5: Partial XML description for Listing 2 running on machine avedon

```
1  <Root>
2   <SystemJProgram Name=‘‘camControl”>
3   <ClockDomain Name=‘‘camControl.camAController’’ IPAddress=‘‘avedon’’
4    Register=‘‘yes’’>
5     <Inputs>
6      <Signal Name=‘‘ctrlA’’ Type=‘‘String’’ RemoteIP=‘‘localhost’’
7          RemotePort=‘‘5555’’
8          SocketClass=‘‘org.systemj.dynamic.Communication.TCPServer’’
9          TimeOutValue=‘‘200’’
10         Serializer = ‘‘mytestSerializer.MytestSerialize’’ BufferSize=‘‘300’’/>
11    </Inputs>
12    <Outputs>
13     <Signal Name=‘‘ctrlMessageA’’ Type=‘‘String’’
14         RemoteIP=‘‘localhost’’ RemotePort=‘‘3333’’
15         SocketClass=‘‘org.systemj.dynamic.Communication.TCPClient’’
16         Serializer=‘‘mytestSerializer.MytestSerialize’’ BufferSize=‘‘300’’/>
17    </Outputs>
18   </ClockDomain>
19   <ClockDomain Name=‘‘camControl.GUIListener’’ IPAddress=‘‘avedon’’>
20    <Outputs>
21         <Signal Name=‘‘attachMessage’’ Type=‘‘String’’
22         RemoteIP=‘‘localhost’’ RemotePort=‘‘2222’’
23         SocketClass=‘‘org.systemj.dynamic.Communication.TCPClient’’
24         Serializer=‘‘mytestSerializer.MytestSerialize’’ BufferSize=‘‘300’’/>
25         <Channel Name=‘‘askA_o’’ Type=‘‘Object’’ RemoteIP=‘‘avedon’’
26         RemotePort=‘‘44468’’
27         SocketClass=‘‘org.systemj.dynamic.Communication.UDPClient’’ />
28    </Outputs>
29    <Inputs>
30         <Channel Name=‘‘receiveA_in’’ Type=‘‘Object’’
31         RemoteIP=‘‘0.0.0.0’’ RemotePort=‘‘44472’’
32         SocketClass=‘‘org.systemj.dynamic.Communication.MultiCastServer’’
33         TimeOutValue=‘‘200’’/>
34    </Inputs>
35   </ClockDomain>
36  </SystemJProgram>
37 </Root>
```

The XML description in Listing 5 shows the system topology for two CDs, the `camAController` CD (lines 3-18) and the `GUIListener` CD (lines 19-35), running on machines `strange-love` and `avedon`, respectively.

On line 3, we declare the name of the CD, qualifying it with the attributes `IPAddress` and `Register`. The attribute `IPAddress` indicates the machine this CD runs on. If accompanied by the `Register=yes` attribute, the runtime system just registers the CD name and the signal/channel arguments with the name cache (see Section 6.1.1) without actually instantiating the CD code (the Java class). In the case of the `camAController` CD, this is useful as the CD itself will be forked later on when asked by the user (Listing 2 lines 10-21). Upon the instantiation of the `camAController` CD, these cached arguments are required to bind the signals and channels for communication via the physical layer. It is not necessary to pre-register the arguments like in Listing 5: the arguments can instead be provided in the DSystemJ program via the DSystemJ library functions.

Next, the XML file binds the `camAController` interface signals `ctrlA` (lines 6-10) and `ctrlMessageA` with the underlying physical socket types. In this case since `ctrlA` is an input signal, we bind it to a TCP server socket (line 8), while the `ctrlMessageA` (lines 13-16) signal being an output signal, it is bound to a TCP client socket (line 15).

The `GUIListener` CD, declared on line 19, is instantiated at the launch of the DSystemJ program, and, hence, this CD declaration is not qualified with a `Register` attribute. The `GUIListener` CD then binds the signals to TCP sockets, but unlike the `camAController` CD it uses a multicast server and UDP client for communication via channels (lines 32 and 27).

6.3 The rendezvous protocol

DSystemJ uses rendezvous via channels to enable communication between reactions in different CDs. A channel consists of two ports, the sending port and the receiving port, respectively. Each port is endowed with channel statuses, which are used to implement a handshake, described formally in Section 4.2. The channel statuses are sampled at the start of the tick and emitted to other CDs at the end of the tick. Thus, channel statuses can be considered equivalent to signals in DSystemJ. Indeed, the rendezvous algorithm uses a number of reactive kernel constructs like `await` and `emit` on the channel statuses to implement the rendezvous [14].

Unlike SystemJ, DSystemJ allows multi-participant rendezvous, i.e., a rendezvous with multiple senders or receivers. Figure 7 shows an example rendezvous between two senders (CD1 and CD3) and a single receiver (CD2). We use a combination of UDP and TCP/IP to implement rendezvous in multi-participant scenarios. Usually, the designer specifies in the XML description of the DSystemJ program the type of rendezvous that needs to be implemented (e.g., Listing 5 lines 8, 15, 23, 27, and 32). A UDP based rendezvous should be chosen by the designer if a multi-participant scenario is expected indeed, it is impossible to carry out a multi-participant rendezvous without the UDP support; in contrast, TCP/IP based rendezvous is more appropriate in the case of point-to-point rendezvous as its faster and more scalable.

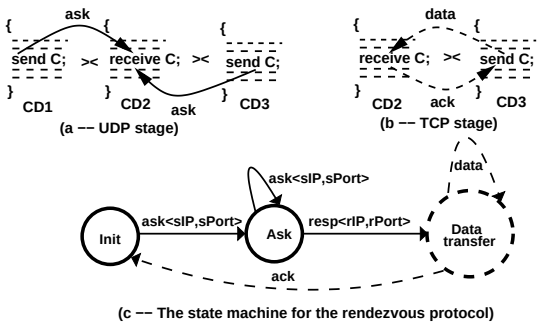

Figure 7: The rendezvous protocol in DSystemJ

Since the UDP based rendezvous encompasses the TCP/IP based rendezvous, we provide a UDP example. In Figure 7 (a) there are two senders trying to synchronize on the channel named C with a single receiver. Both senders broadcast the request to rendezvous (`ask` signal) on channel C. The receiver listening to this broadcast sends a reply back to a single sender, chosen *non-deterministically*. Figure 7 (b) shows the receiver choosing the second sender (CD3). Once a pairing is established, the sender and receiver carry out a two-phase handshake to synchronize and deliver data, if any (Figure 7 (b)).

Figure 7 (c) shows the state machine for the complete rendezvous protocol. The solid arrows and circles show the UDP communication, while the dashed arrows and circles show the TCP/IP based communication.

Let us now consider the rendezvous scenario in Figure 7 (c) in more detail. In the `Init` state, the receiver listens on the multicast IP-address specified in the XML description. The senders, on the other hand, broadcast using the multicast clients, while at the same time also listening on TCP/IP ports (`sPort`) and IP-address (`sIP`) allocated by the socket manager (see Section 6.1.7). The senders broadcast a request to rendezvous (`ask`) along with `sIP` and `sPort` at the end of every tick.

At the start of its tick the receiver samples incoming data on the multicast server. If the receiver receives more than one rendezvous request (`ask`), the receiver chooses a sender *non-deterministically*. The receiver then stops listening on the multicast server and instead initializes a new TCP/IP server to start listening on. The TCP/IP-address (`rIP`) and the port number (`rPort`) are allocated by the socket manager. The receiver sends a TCP/IP reply to the sender using the `sIP` and `sPort` as the destination address and port, respectively. This reply consists of the `rIP` and `rPort` parameters.

Each sender (CD1 and CD3) samples incoming packets on the TCP/IP server at the start of its tick. Once the sender receives the receiver's `rIP` and `rPort` identifiers, everything needed to carry out a rendezvous is now available. Next, the sender and receiver carry out a two-phase handshake using the algorithms described in [14].

After the completion or preemption of this rendezvous, the receiver stops listening on its TCP/IP server (in the process freeing and recycling the sockets) and starts listening on

Table 3: Qualitative comparison between DSystemJ and other similar languages and libraries

Languages and Libraries	**Process Forking**	**SR-constructs**	**Asynch-constructs**	**Mobility**	**Formal-MoC**	**Heterogeneity**
MPI	No	No	Yes	No	No	Yes
RML	Yes	Yes	No	No	Yes	Yes
JoCaml	Yes	No	Yes	No	Yes	Yes
Occam-π	Yes	No	Yes	Yes	Yes	No
Erlang / Salsa / Scala	Yes	No	Yes	partial support	No	Yes
ActorFoundry	No	No	Yes	Yes	Yes	Yes
JADE	Yes	No	Yes	Yes	No	Yes
DSystemJ	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

the multicast UDP server again, ready to carry out another rendezvous when requested. Finally, it should be mentioned that the above mentioned rendezvous protocol is only valid in the absence of strong preemptions, caused by preemptive constructs like `await`. In case of strong preemptions, the rendezvous transition rules from Section 4.2.2 and [14] apply.

7 Comparison with other languages and libraries

Table 3 compares the qualitative properties between the different languages and libraries found in the literature. **Process Forking**, is the ability to provide mechanisms to easily express dynamic process creation. **SR-constructs** is ability of the language to incorporate data-fusion capabilities as first class citizens of the language. **Asynch-constructs** defines the ability to program asynchronous distributed and multi-core platforms. **Mobility** is the ability to describe movement of program code and possibly data on geographically distinct machines. **Formal-MoC**, is the property that the language is based on rigorous mathematical foundations. Finally, **Heterogeneity** is the property that control and data-dominated applications can both be described and combined with ease.

As one can see from Table 3, DSystemJ is the only one to full-fill these criteria. MPI [21] is the de-facto industry standard for programming distributed systems, but being a library rather than a language it lacks both; abstraction and a formal MoC. RML [16] and JoCaml [15], both formal languages, are based on very different concepts. RML provides abstraction and SR data-fusion constructs like DSystemJ, but lacks support for asynchronous processes and mobility. JoCaml based on join-calculus [8] is targeted at design and implementation of distributed programs, but lacks support for reactivity, and mobility.

Occam-π [24] is based on the π-calculus [18] and hence is closest to DSystemJ, but, unlike DSystemJ, it does not support implementation of heterogeneous designs (with significant data-dominated computations); also, it does not provide any reactive constructs as first class citizens of the language. Erlang [23], Salsa [22], Scala [19], ActorFoundry [1], and JADE [4] are all based on the actor model of computation [7]. Erlang, Scala, and Salsa do

Table 4: Examples, *lines of code* (LOC), generated memory footprint, and total memory footprints

Examples	**LOC**		**Generated memory foot-print (KB)**		**Total memory foot-print, including libraries (KB)**	
	DSystemJ	JADE	DSystemJ	JADE	DSystemJ	JADE
send-receive	**39**	118	38	**5.6**	**145**	2616.6
camControl	**125**	238	158	**14.5**	**265**	2625.5
sieve	**163**	267	99	**12**	**216**	2623

Table 5: Runtime comparison between DSystemJ and JADE on a single machine with two cores

Examples	**Runtime (ms/tick)**									
	DSystemJ					JADE				
send-receive	CD1	CD2				CD1	CD2			
	5	**5.57**				74.7	185.9			
sieve	CD1	CD2	CD3	CD4	CD5	CD1	CD2	CD3	CD4	CD5
	0.1	**17**	**16.75**	**23.4**	**17**	1	340	361.8	322.435	514

Table 6: Runtime comparison between DSystemJ and JADE on a distributed platform with two machines and four cores

Examples	**Runtime (ms/tick)**							
	DSystemJ				JADE			
send-receive	CD1	CD2			CD1	CD2		
	20.7	**22.2**			86.88	470		
camControl	CD1	CD2	CD3	CD4	CD1	CD2	CD3	CD4
	202.715	**191.33**	**125.14**	**133.666**	3243.44	1498.1139	1320.5584	1603

not support process mobility as first class citizens of the language, while ActorFoundry and JADE support weak and strong mobility as language programming paradigms, respectively, (Salsa and Scala pass references rather than copies of program code or of messages, which can contain program code and hence, are unable to accomplish mobile distributed processing). None of these approaches also provide the SR-programming paradigm.

8 Experimentation Results

In this section we quantitatively compare DSystemJ with JADE. We chose JADE for comparisons, because the released version of ActorFoundry [1] does not support distributed implementations, and MPI-based Java bindings do not support process forking and mobility. All the benchmarks, DSystemJ compiler, and runtime library are available to download from [2].

Table 4 shows the examples that we have chosen for comparison. We chose three very different programs; (1) `send-receive` is a simple communicator, that sends and receives continuously between two CDs in a very tight loop. Both the CDs in send-receive are static, i.e., forked at the start of the program. This program judges the communication performance. (2) `Sieve`, is the classical sieve of Eratosthenes, which computes primes. The

`Sieve` example involves a large amount of process forking, except for a single CD all the other CDs are forked multiple times dynamically, and communication between the various CDs is also established dynamically. Finally, (3) `camControl` is the example shown previously in Listings 1 and 2; it involves significant amount of code mobility across machines in a network, in conjunction with dynamic process forking. The experimental setup consists of: (1) a two-core 32-bit Linux machine running Sun-jdk-1.6 and (2) a two-core 64-bit Linux machine running open-jdk-1.6. All the Java class files were compiled using Sun javac-1.6 compiler.

As can be seen from Table 4, DSystemJ performs well compared to JADE. DSystemJ's abstract syntactic constructs along with its formal MoC help the designer to write code succinctly. JADE, being a library, lacks these advantages and, hence, requires more *lines of code* (LOC). DSystemJ also performs better than JADE with regards to the total memory foot-print (class files). This advantage can be attributed to the tiny DSystemJ library foot-print (106KB for DSystemJ compared to 2.6MB for JADE) as opposed to the generated code size. DSystemJ compiler produces bigger Java files and consequentially larger class files, unlike the hand written Java files as is the case with JADE.

Tables 5 and 6 show the runtime comparison between DSystemJ and JADE. Table 5 shows the runtime for a single 32-bit machine implementation with two cores. We ran the `send-receive` and `sieve` examples on this platform for a million ticks to get the results. The runtime is in `ms/tick`. DSystemJ has a clear notion of a tick; for JADE, all agents were implemented with `CyclicBehaviour` class, which implements reactivity, to emulate the same behaviour as in DSystemJ. DSystemJ is far superior compared to JADE implementations. The slow JADE runtimes can be attributed to the fact that JADE implements a much more elaborate communication, dynamic forking, and mobility protocol compared to DSystemJ, based on the FIPA [3] standard. Also, while DSystemJ runtime library is optimized for single machine implementations, JADE concentrates more on the transparent locality model, i.e., the same communication mechanism is used for single and distributed platforms and this difference in implementation affects the runtime results.

In a distributed setting (Table 6), DSystemJ again outperforms JADE, although, in this case, the performance difference is smaller. This lack of JADE performance can again be attributed to the elaborate communication and mobility protocols that one has to follow when implementing JADE agents. On average, DSystemJ is 20 times faster compared to JADE on a single machine (multi-core) implementation, and 12 times faster in a distributed setting.

9 Conclusion and future work

In this article, we have described a new programming language called DSystemJ, designed specifically for *dynamic distributed systems*. DSystemJ has rigorous mathematical semantics and hence, is amenable to compilation and formal reasoning. DSystemJ compared to other formal languages in its application domain provides an exhaustive design perspective, taking distributed communication, mobility, dynamic forking, and reactivity into account, thereby

easing the design burden of software programmers. DSystemJ' operational semantics can be used to derive the behavioural semantics and thus, are helpful in abstract modelling and formal verification (out of the scope of this article).

In the future we plan to provide tools for formal verification and real-time analysis of DSystemJ programs. We also plan to use the presented operational semantics to derive distributed controllers for addressing the fairness and non-deterministic behaviour of DSystemJ programs raised in this article.

References

[1] ActorFoundry. `http://osl.cs.uiuc.edu/af/` [Last Access: 23/03/2010].

[2] The DSystemJ website. `http://dsystemj.gforge.inria.fr` [Last Access: 25/06/2010].

[3] The Foundation for Intelligent Physical Agents. `http://www.fipa.org` [Last Access: 25/06/2010].

[4] The Jade website. `http://jade.tilab.com` [Last Access: 16/03/2010].

[5] G. Berry. The semantics of pure esterel, 1993.

[6] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Logic in Computer Science, 1990. LICS '90, Proceedings*, pages 428–439, Jun 1990.

[7] W. Clinger. *Foundations of Actor Semantics*. PhD thesis, Massachusetts Institute of Technology, 1981.

[8] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385, New York, NY, USA, 1996. ACM.

[9] J. Galletly. *Occam-2*. University College London Press, 2nd edition.

[10] L. Henry. *Reference Manual for the ADA Programming Language*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1983.

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

[12] E. Lee. The problem with threads. *IEEE Computer*, 39:33–42, May 2006.

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

[14] A. Malik, Z. Salcic, P. S. Roop, and A. Girault. SystemJ: A GALS language for system level design. *Elsevier Journal of Computer Languages, Systems and Structures*, 36(4):317–344, December 2010.

[15] L. Mandel and L. Maranget. Programming in JoCaml – Extended version. Technical Report 6261, INRIA, 2008.

[16] L. Mandel and M. Pouzet. ReactiveML: a reactive extension to ML. In *PPDP*, pages 82–93, New York, NY, USA, 2005. ACM.

[17] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[18] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992.

[19] M. Odersky, P. Altherr, V. Crement, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[20] D. Potop Butucaru. *Optimisations for Faster Execution of Esterel Programs*. PhD thesis, Ecole des Mines de Paris, 2002.

[21] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Inc.

[22] C. Varela and G. Agha". Programming Dynamically Reconfigurable Open Systems with SALSA. *"ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings"*, "36"("12"):"20–34", December "2001".

[23] R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall PTR, 2 edition, 1996.

[24] P. Welch and F. Barnes. Communicating mobile processes: introducing occam-pi. In A. Abdallah, C. Jones, and J. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, April 2005.

[25] L. Yoong, P. Roop, and Z. Salcic. Compiling Esterel for Distributed Execution. In *SLAP*, March 2006.

Contents

List of Figures

List of Tables

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399