A GALS Language for Dynamic Distributed and Reactive Programs

Avinash Malik, Alain Girault, Zoran Salcic

A GALS Language for Dynamic Distributed and Reactive Programs

Avinash Malik
IBM Research Dublin and Department of Computer Science and Statistics, Trinity College Dublin, Ireland.
Email: avimalik@ie.ibm.com, amalik@scss.tcd.ie
Alain Girault
INRIA Grenoble Rhône-Alpes and LIG laboratory, POP ART team, Grenoble, France.
Email: alain.girault@inria.fr
Zoran Salcic
Department of Electrical and Computer Engineering, University of Auckland, New Zealand.
Email: z.salcic@auckland.ac.nz

***Abstract*—We propose a *Globally Asynchronous Locally Synchronous* language DSystemJ for designing dynamic distributed systems. DSystemJ, an extension of the reactive asynchronous SystemJ language, enhances it with dynamic creation and process mobility, and uses the Java language for programming sequential data computations. Moreover, DSystemJ is equipped with a formal semantics, which allows, formal system specification, reasoning, and automatic code generation. Compared to special purpose languages, DSystemJ is better in terms of implementation, scalability, and features. Compared to general purpose languages, DSystemJ is better because it exposes, at the language level, OS features like communication, concurrency, process creation and migration, therefore making it better suited for system level design of complex distributed systems.**

***Keywords*-GALS, distributed systems, dynamic process creation, weak process mobility, Java, DSystemJ.**

I. Introduction

With the availability of high performing processors connected in distributed configurations, the number of applications with dynamic changes of system structure and configuration is increasing. Some typical examples of such systems are complex sensor networks capable of dealing with nodes being attached or detached at runtime, ad-hoc systems like collaborative gaming and collaborative editing environments, security surveillance systems, and medical robots, the last two also being safety critical systems. Programming languages that can be used to abstractly and easily model, design, and implement such systems are therefore becoming essential. The required features of a programming language to adequately design such dynamic distributed systems are: dynamic process creation and migration, asynchronous concurrency, communication, reactions on the events generated in the environment, fusion of such events along with corresponding data, and, of course, constructs for manipulation of data structures and abstractions. Additionally, we advocate that it is important to have a formal semantics, because the complexity of such systems makes them hard to compile and difficult to reason about.

The most popular languages of today, like C/C++ and Java, are *not* suitable for describing such complex dynamically evolving distributed systems (from here on referred to as dynamic distributed systems) because these languages lack the mechanisms to describe even static concurrency and communication. For example, the Java threading model is based on the shared memory model of communication, which is absent in a distributed memory environment. Thus, a program implemented using Java threads alone would be unable to meet the needs of programming a dynamic distributed system. Even in the presence of shared memory, programming concurrent systems with general purpose languages is still considered difficult [1]. Moreover, Java or C/C++ programs are usually run in conjunction with large runtime libraries [2], [3], [4], which act as the middleware for describing and implementing distributed memory communication and concurrency. These runtime libraries are often larger than the programs themselves and thus can be too large a burden on the resources of a system, especially in cases such as sensor networks. Besides, library based approaches (e.g., [4]) are usually not supported by formal semantics, hence they are unsuitable for designing safety critical systems.

A number of formal languages and libraries, ULM [5], JADE [2], ActorFoundry [6], RML [7], CRP [8], Occam [9], Occam-π [10], and JoCaml [11], have been proposed to model distributed, dynamic, and possibly even mobile systems. ActorFoundry and JADE are based on the actor model of computation, they do not support reactivity, and being libraries they are not amenable to automated formal reasoning. Occam, Occam-π, RML, JoCaml, CRP, and ULM are some of the most promising languages for describing and implementing dynamic distributed systems. All these languages have advantages and disadvantages: Occam and Occam-π both lack the support for abstract data fusion constructs, essential to ease the development of sensor network applications. Besides, they lack support for complex high level data manipulation capabilities. RML supports both high level data manipulation capabilities, being based on OCaml [12], and synchronous reactive data fusion constructs like in Esterel [13], but lacks support for asynchronous process creation and communication. JoCaml, on the other hand, encompasses asynchrony but lacks support for process migration and synchronous reactivity. ULM supports asynchronous communication between synchronous islands, but does not provide a generic communication framework between these islands. Also, ULM cannot express all possible reactive programs, as it is based on the SL [14] synchronous language rather than Esterel [13]. CRP [8] as opposed to ULM extends the Esterel language with asynchrony and thus is more expressive compared to ULM, but CRP lacks support for process mobility and inbuilt support for complex data transformations. Finally, the approach

in [15] introduces higher-order Esterel synchronous processes, which can be instantiated at runtime. This approach is related to DSystemJ, because of its ability to send and receive processes as closures, but, unlike our approach, it neither provides mobility, as the Esterel processes cannot be moved from one computation node to another (like in DSystemJ), nor any form of asynchrony. Overall, none of these languages natively support all the features required for programming dynamic distributed systems.

SystemJ [16] is a recently introduced language that implements the formal *Globally Asynchronous Locally Synchronous* (GALS) *Model of Computation* (MoC). It combines the Esterel style reactivity [13] and the CSP style message passing [17], with the Java language to provide powerful and abstract means of implementing complex concurrent and reactive systems with high level data-manipulation constructs. SystemJ targets two contrasting application directions: (1) multi-core single computer platforms by exploiting parallelism and (2) reactive safety critical systems. However, SystemJ does not provide any means of exploiting concurrency in a distributed setting.

In this article, we propose DSystemJ, a conservative extension of SystemJ, which extends SystemJ with: (1) the ability to create asynchronous processes called clock-domains at runtime (dynamic creation), (2) exploitation of parallelism at both the multi-core and distributed system levels, (3) the ability to move clock-domains around (weak process mobility), and (4) a rendezvous implementation between multiple participants in a distributed setting without a single point of failure.

The main contributions of this article are: (1) the presentation of a new reactive GALS language, DSystemJ, capable of dynamic process creation and mobility in a distributed setting; (2) the formalization of dynamic process creation and mobility in presence of reactivity and the GALS MoC; (3) the implementation and formalization of communication protocol between processes running in a distributed setting and in the presence of dynamicity and mobility; (4) the implementation of an open-source compiler for DSystemJ; (5) and finally, the implementation of a very efficient and easily extensible runtime system and a library for DSystemJ. DSystemJ's library and runtime system are written in SystemJ, a formal language itself, making the complete system amenable to automated formal reasoning and verification.

The rest of the paper is organized as follows. Section II presents the DSystemJ syntax and intuitive semantics. Section III motivates and introduces the DSystemJ language via an example. Section IV explains the formal semantics of the DSystemJ language. Section V shows the compilation and implementation of the language. Section VI gives some quantitative comparisons, and finally, we present the conclusions in Section VII.

II. DSystemJ: Model of computation, syntax and intuitive semantics

DSystemJ is a conservative extension of SystemJ and hence it follows the *Globally Asynchronous Locally Synchronous* (GALS) MoC of the SystemJ language.

A. DSystemJ: Model of Computation

A **SystemJ** program `system` consists of a set of asynchronous concurrent processes called clock-domain(s) (CD(s)) composed using the $><$ operator and executing at unrelated logical clock ticks (from here on referred to as tick), and synchronizing and communicating with each other using channels. SystemJ uses CSP [17] style rendezvous for synchronization and data transfer between CDs. Each CD itself consists of one or more processes, called reactions, executing in lockstep, defined by the CD's **tick**, representing a logical clock at which the CD executes. 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 and operations on these signals. Every CD samples inputs from the environment, reacts to these inputs instantaneously (perfect synchrony hypothesis [13]) and produces the outputs back to the environment, thereby implementing a state machine. 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 dynamically at runtime (dynamic process creation) and passing CDs over channels (weak process mobility).

B. DSystemJ: Syntax

DSystemJ combines features from Esterel [13], CSP [17], and π-calculus [18] with the Java programming language. Tables I and II show the SystemJ and DSystemJ kernel statements and their meanings. A more detailed explanation of these constructs is presented later in the sub-sections.

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

Table I: 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, blocking send
receive C()	receive a value through C, blocking receive
pause	finish a tick and communicate with environment

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 the `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`).

Table II: New syntactic constructs in DSystemJ.

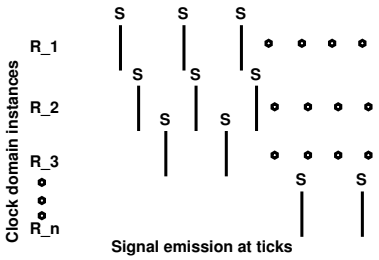
Syntactic constructs	Interpretation
unique-name -> cd()	pointer to a named CD cd
unique-name -> {...}	pointer to an unnamed CD given between {}
`run` unique-name ([args])	run a new instance of a CD
`send` channel-name (unique-name)	send the code of a CD
`receive` channel-name	receive the code of a CD
`run` #channel-name ([args])	run a new instance of the received CD code
`[input] [output] [type] channel channel-name`	declare a new channel

C. DSystemJ: Intuitive Semantics

In this section, we describe the intuitive semantics of the DSystemJ syntactic constructs presented in Table II.

1) The unique-name −> cd() and unique-name −> {} constructs: The −> construct acts as the pointer to named or unnamed CDs, as shown in Listing 4 (lines 7-13). The −> delimited names ("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 4 (lines 7 and 9), `cam1` and `cam2` enclose the signals `a`, `c`, and `b`, `d`, respectively. Every closure keeps a separate copy of the enclosed variables, which can also include reactive objects like signals. These enclosed variables are not shared amongst the closures. Finally, the −> operator can be applied to the same named CD but with different arguments (Listing 4, lines 7 and 9).

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 a CD already present within the 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 but unbounded number of ticks (i.e., the number of ticks cannot be determined statically) to succeed, "tick" being 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., the running state of a CD cannot be saved and, hence, DSystemJ only allows weak mobility. For the same reason, a top-level CD can be migrated but it has to restart from its initial state.




```
output signal S;
R -> {
 //fork itself and move to the next statement
 //once the rendezvous with runtime is complete.
 run R();
 //emit S forever.
 while(true) {emit S; pause;}
}
```

The figure above shows one possible outcome of running the sample code above: CD `R` forks itself. Multiple instances of CD `R` run at the same time, each emitting the signal `S` forever. These are different instances of `S` since `S` is enclosed by `R`. As one can see from this example, DSystemJ unlike SystemJ and other reactive languages, like Esterel, does not guarantee bounded time and memory consumption, which is expected since it is dynamic.

3) The `send` channel-name (unique-name) and `receive` channel-name 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 unique-name itself, rather than manually constructing a Java object containing the marshalled CD code, to implement weak CD mobility.

The major difference between DSystemJ and SystemJ rendezvous communication stems from the fact that DSystemJ rendezvous communication is not point-to-point (linear). 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. See Listing 1.

Listing 1: An example of non-linear channel communication in DSystemJ

```
1  // Example of Many to One non−linear channel communication on channel M.
2  // Note that >< is the asynchronous composition operator from SystemJ.
3
4  // CD P running on machine 1 sends itself via channel C.
5  // In parallel, it also sends values via channel M.
6  P −> {{send C(P);} || { while(true) send M(4);}}
7  ><
8  // CD Q running on machine 2 gets the value via channel M.
9  Q −> {while(true) receive M;}
10 ><
11 // CD R running on machine 3 forks CD P obtained via channel C
12 // and finishes itself. But, now CD P runs on machine 3 as well,
13 // blocking on channel C and also sending values on channel M.
14 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 [19]. This raises fairness issues as it is possible to introduce starvation in a system in presence of multi-participant rendezvous. For example, in Listing 1 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 partially detect deadlocks/starvation in presence of multi-participant rendezvous at compile time. The deadlock detection algorithm is based on the SystemJ deadlock detection algorithm described in [20]. As DSystemJ is targeted towards dynamic distributed systems with the goal of *no single point of failure*, there is no single entity in the system with 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 using the `channel` construct. To avoid such problems, DSystemJ might allow, in the future, multiple communication alternatives, such as rendezvous in a distributed environment and join calculus based communication in a single subnet or a single machine. The join calculus allows the combination of multiple sent and received values by using combinator function [21]. This approach is practical in a small subnet or single machine implementations, where data delivery time between senders and receivers can be bounded. The implementation details of the DSystemJ rendezvous are described in [22].

III. A Complete DSystemJ Example

In this section we show a complex security surveillance system modelled and implemented in DSystemJ. This system is used as a running example throughout the article to familiarise the reader with the language features.

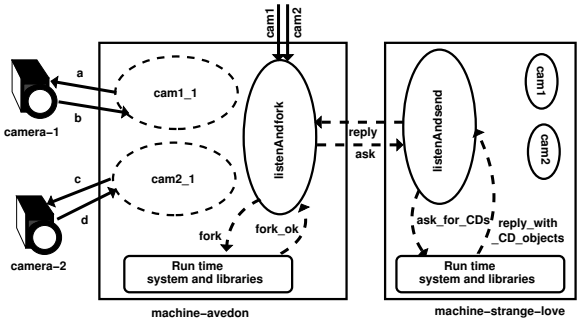


Figure 1: Pictorial representation of Listing 2

Listings 2, 3, and 4 show parts of a security surveillance system, for which Figure 1 shows a pictorial view. This system consists of multiple sensors (different types of cameras) that are used to track people in an indoor environment. This example features asynchronous concurrency, communication, dynamic asynchronous process creation, reactivity, and CD (clock-domain) mobility. The system consists of IP enabled cameras, which act as sensors for tracking. Listings 2, 3, and 4 show an abstract representation of the main components of the DSystemJ program implementing the surveillance system.

Listing 2: The DSystemJ code that runs on machine avedon

```
1  // Import the required classes from the DSystemJ library.
2  import org.systemj.dynamic.util.Helper;
3  import org.systemj.dynamic.interfaces.SignalArgs;
4  // This reaction listens to the environment and forks CD as required.
5  reaction listenAndfork(:output channel boolean ask,
6                         input channel Object reply){
7   // Wait for a signal for two different type of cameras
8   while(true){
9   // If we get a cam1 signal from environment
10    present(cam1){
11     // Check if this CD is registered with the runtime.
12     // ''controller'' is the name of the DSystemJ program.
13     if(Helper.exists(''controller.cam1'')
14       // Run it using the default arguments cached in the runtime system.
15       run cam1();
16     else{
17       // Ask for this CD.
18       send ask(''controller.cam1'');
19       // Receive a reply with the CD in it.
20       receive reply;
21       // Create the new signal args that will bind the signals
22       // used in cam1 CD to the physical layer.
23       SignalArgs
24        a1 = new SignalArgs(''a'',''44569'',''avedon''
25           ''org.systemj.dynamic.Communication.TCPClient'', ''0'',
26           ''null'',''String'',''0''),
27        b1 = new SignalArgs(''b'',''44570'',''avedon'',
28           ''org.systemj.dynamic.Communication.TCPServer'', ''200'',
29           ''null'',''int'',''0'');
30       ArrayList list = new ArrayList();list.add(a1); list.add(b1);
31       // Run the received CD passing it the new args.
32       run #reply(list);
33     }
34    }
35    else present(cam2){//Same for cam2}
36   }
37  }
```

Listing 3: The DSystemJ code that runs on machine strange-love

```
1  // The generic camera controller present on strange−love
2  reaction camControl(:output signal a,
3                      input signal b){//tracking control}
4  // Abstract representation of the CD that replies to the requests
5  // from listenAndfork, running on strange−love.
6    reaction listenAndsend(:output channel Object reply,
7                           input channel boolean ask){
8    while(true){
9      receive ask;
10     send reply(cam1);
11   }
12 }
```

Listing 4: The top-level system

```
1  system{
2   interface{
3    input signal a,b;output signal c,d;
4    // Some other channel declarations.
5   }
6   {
7    cam1 −> camControl(a,c)
8    ><
9    cam2 −> camControl(b,d)
10   ><
11   listener−> listenAndfork(//channel args)
12   ><
13   sender −> listenAndsend(//channel args)
14  }
15 }
```

All the listings are well commented for ease of understanding. The CD `listenAndfork` (Listing 2, lines 4-37) is running on a machine called `avedon`, while the other CDs (Listing 3 and 4) are running or present on another machine called `strange-love`. Although we run Listing 4 on strange-love in this case, it could also have been run on avedon, since it is the top-level system and is not bound to a specific machine. The top-level system instantiates the CDs that are then bound to the machines (as shown in Figure 1) using an XML architecture description (see [22] due to lack of space).

A CD is considered to be present on a machine if it is registered with the runtime system on this machine. Accordingly, a CD is said to be *registered* with the runtime system if the fully qualified name (`name_of_systemj_file.name_of_CD`) is known by the runtime. At start-up, allocation and distribution of the CDs on different machines is done by the DSystemJ runtime system, where the system is described using XML [22]. The `listenAndfork` CD listens to incoming signals and forks camera controllers accordingly. If the camera controller that needs to be forked is already registered with the runtime (the registration is carried out implicitly via the `run` statement) the camera controller is forked internally using a rendezvous with the runtime. If the required CD is not registered with the runtime system then the `listenAndfork` CD asks for the required camera controller CD code via the channels `ask` and `reply` (Listing 2 lines 18, 20). Once the required camera controller CD code is received, it forks this CD, passing the appropriate arguments to the camera controller (Listing 2 line 32). The # operator retrieves the value from the channel.

In Figure 1, the dotted arrows show the channels, while the rigid arrows show the signals. Similarly, the dotted ellipses show the instances of the CDs forked using the `run` statement, while the rigid ellipses show the CDs running at startup or present in the system but not instantiated. Furthermore, the `fork`, `fork_ok`, `ask_for_CDs`, and `reply_with_ CD_objects` channels shown in Figure 1 are the internal rendezvous channels used to instantiate new CDs via the `run` statement and to obtain Java objects encapsulating the marshalled CD code ready to be sent via channels, respectively.

As can be seen from Listings 2, 3, and 4, DSystemJ provides very abstract means for describing code mobility via channels and rendezvous. The system's topology and physical configuration are also represented within the language (e.g., Listing 2, lines 23-29).

Compared to Java alone, asynchronous concurrency, communication between CDs, dynamic creation of CDs, and data encapsulation can be described very easily and abstractly in DSystemJ. For example, the developer does not need to delve into the low level details of mutual exclusion like in Java. Similarly, the locality of each asynchronous CD can be abstracted out when communicating via channels, unlike in Java, where the designer needs to deal with socket or *Remote Method Invocation* (RMI) calls explicitly. Overall, DSystemJ is a very abstract programming language with a runtime support for designing complex distributed systems that evolve in their life time.

IV. Formal semantics

This section presents the formal semantics and the MoC of DSystemJ. Both are described in terms of SystemJ MoC and micro-step semantics, so we describe these two first in Section IV-A. These micro-step semantics can be used to construct the macro-step semantics of compiled DSystemJ programs. The macro-step operational semantics are essential for formal reasoning and *Worst Case Reaction Time* (WCRT) analysis of DSystemJ programs.

A. Semantics of SystemJ

All of SystemJ's constructs 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.

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 of $\perp$. Term $data$ represents the value stores attached to the statement p before 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, else, an integer value within $[0, \infty]$. A termination code of 0 represents the completion of reaction; 1 represents the completion of a single tick; a value in the interval $[2, \infty)$ is reserved for preemptions based on `trap/exit` constructs; and finally ∞ represents an unresolved signal dependency.

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 and 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, $E_c = \{\{Cw_s, Cw_r, Cp_s\}, \{Cr_s, Cr_r, Cp_r\}\}$. Here, the set's elements represent the output and input channel port statuses. 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 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 the execution of statement p. 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. Also, a $\ddot{p}$ indicates that the position of the current control point over p is unknown. We refer the reader to [23] for full definitions of $\hat{p}$, $\bar{p}$, and $\ddot{p}$.

The example below shows the micro-step semantics of `pause` execution. The $\bullet$ represents the control point movement in the SystemJ program code. 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 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:

$$\bullet pause \xrightarrow[E]{1,\perp} \widehat{pause} \qquad \bullet \widehat{pause} \xrightarrow[E]{0,\perp} pause$$

In our operational semantics, the control point (•) is considered as a part of the semantic rules, i.e., a construct is considered enabled for execution `iff` it is decorated with a •, else the construct cannot be executed. Consequently, as opposed to the more traditional way of rewriting a construct into a nothing (or empty) statement upon the completion of its execution, we represent this behavior with the control point movement. Our representation, as opposed to the more traditional approach, is borrowed from the operational semantics of Esterel [23]. There are a number of other rewrite rules associated with reactive constructs of SystemJ, which are out of the scope of this article (see [16], [24] for the complete set of rewrite rules).

A CD's execution is represented with a macro-step $\hookrightarrow$, which is defined as a sequence of micro-steps:

$$\hookrightarrow := \{\rightarrow, \rightarrow, , , \}$$

where $\rightarrow$ are the micro-steps of the DSystemJ constructs encapsulated within the CD. The macro-step transition rule for a SystemJ program is expressed in terms of the macro-step transition of individual CDs. The macro-step transition for a SystemJ program is:

$$\bullet\bar{s_1} \overset{e_{s_1},k_{s_1}}{\underset{E_{s_1},E_{cs_1}}{\hookrightarrow}} \ddot{\bar{s}}_1, \quad \bullet\bar{s_2} \overset{e_{s_2},k_{s_2}}{\underset{E_{s_2},E_{cs_2}}{\hookrightarrow}} \ddot{\bar{s}}_2 \quad \ldots \quad \bullet\bar{s_m} \overset{e_{s_m},k_{s_m}}{\underset{E_{s_m},E_{cs_m}}{\hookrightarrow}} \ddot{\bar{s}}_m, \tag{2}$$

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.

B. Semantics of DSystemJ

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

1) DSystemJ Formal MoC:: DSystemJ program is equivalent to a SystemJ program if it has the same number of executing CDs, ($m = n$) and 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. 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 dynamically instantiate new CDs at runtime.

2) Rewrite rules for DSystemJ syntactic constructs:: We now describe the rewrite rules of the DSystemJ constructs presented in Table II.

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

$$\bullet\texttt{unique-name} \rightarrow \{\} \xrightarrow[E,E_c]{0,\perp} \texttt{unique-name} \rightarrow \{\} \tag{3a}$$

$$\bullet\texttt{unique-name} \rightarrow \texttt{cd} \xrightarrow[E,E_c]{0,\perp} \texttt{unique-name} \rightarrow \texttt{cd} \tag{3b}$$

The `run` construct: The `run` statement does not have a single micro-step rule. Instead, every `run` statement is rewritten 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`
`m` being the program code of some other CD `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 programs 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 system where the CD `CD (m)` runs in asynchronous parallel with the forking CD q after an extra transition τ. This is the required behaviour of the `run` statement.

Informally, the semantics of the `run` statement assumes that 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 CD 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.

The `send` and `receive` constructs: DSystemJ's `send` and `receive` constructs implement CSP [17] style message passing. The difference with SystemJ is that we introduce the non-deterministic choice operator □, which chooses a rendezvous partner in case of a non-linear rendezvous with multiple participants (see Section II-C3). This is similar to `select` statement in ADA [19].

$$\frac{E_{c_p}[Cp_s] = E_{c_q}[Cp_r], E_{c_q}[Cr_r] > E_{c_q}[Cr_s], E_{c_r}[Cp_s] = E_{c_q}[Cp_r]}{\{\bullet\hat{p}, data \Box \bullet \hat{r}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} p, data' \bullet \hat{r}, data\}, \{\bullet\hat{q}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} q, data'\}} \tag{4a}$$

$$\frac{E_{c_p}[Cp_s] = E_{c_q}[Cp_r], E_{c_q}[Cr_r] > E_{c_q}[Cr_s], E_{c_r}[Cp_s] = E_{c_q}[Cp_r]}{\{\bullet\hat{p}, data \Box \bullet \hat{r}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} r, data' \bullet \hat{p}, data\}, \{\bullet\hat{q}, data \overset{0,\perp}{\underset{E,E_c}{\hookrightarrow}} q, data'\}} \tag{4b}$$

Rules (4a) and (4b) show the macro-step rendezvous transition, when the rendezvous conditions are fulfilled, for two senders and a single receiver. The rules are read as follows: provided that no CDs are preempted, that $E_{c_p}[Cp_s]$ is equivalent to $E_{c_q}[Cp_r]$, and $E_{c_r}[Cp_s]$, and the receiving CD q is ready to rendezvous, (shown by $E_{c_q}[Cr_r] > E_{c_q}[Cr_s]$), then the rendezvous takes place by making a non-deterministic choice between either of the sending CDs. The □ operator internally and non-deterministically chooses

one of the sending CDs p (Rule (4a)) or r (Rule (4b)) to rendezvous with the receiving CD q. The other CD blocks waiting for an acknowledgement from the receiver. The r_r port status is updated at the start of every tick by sampling the w_s statuses of the sending CDs.

The rendezvous transition Rules (4a)–(4b) are valid only in the absence of strong preemptions, possible due to constructs such as an `abort`. DSystemJ's strong preemption in presence of rendezvous is similar to that of SystemJ's, except, a preemption can occur even before choosing a partner in case of multi-participant rendezvous.

$$\frac{E_{c_p}[Cp_r] \neq E_{c_q}[Cp_s], E_{c_r}[Cp_r] \neq E_{c_q}[Cp_s]}{\{\ddot{p} \xrightarrow[E,E_c]{\perp,\perp} \ddot{p}\}, \{\ddot{r} \xrightarrow[E,E_c]{\perp,\perp} \ddot{r}\}, \{\bullet\bar{q} \xrightarrow[E,E_c]{1,\perp} \dot{q}\}} \quad (5a)$$

$$\frac{E_{c_p}[Cp_r] = E_{c_q}[Cp_s], E_{c_r}[Cp_r] \neq E_{c_q}[Cp_s]}{\{\bullet\bar{p} \xrightarrow[E,E_c]{0,\perp} p\}, \{\ddot{r} \xrightarrow[E,E_c]{\perp,\perp} \ddot{r}\}, \{\bullet\bar{q} \xrightarrow[E,E_c]{0,\perp} q\}} \quad (5b)$$

$$\frac{E_{c_p}[Cp_r] \neq E_{c_q}[Cp_s], E_{c_r}[Cp_r] = E_{c_q}[Cp_s]}{\{\ddot{p} \xrightarrow[E,E_c]{\perp,\perp} \ddot{p}\}, \{\bullet\bar{r} \xrightarrow[E,E_c]{0,\perp} r\}, \{\bullet\bar{q} \xrightarrow[E,E_c]{0,\perp} q\}} \quad (5c)$$

$$\frac{E_{c_p}[Cp_r] = E_{c_q}[Cp_s], E_{c_r}[Cp_r] = E_{c_q}[Cp_s]}{\{\bullet\bar{p} \xrightarrow[E,E_c]{0,\perp} p\}, \{\bullet\bar{r} \xrightarrow[E,E_c]{0,\perp} r\}, \{\bullet\bar{q} \xrightarrow[E,E_c]{0,\perp} q\}} \quad (5d)$$

Rules (5a)–(5d) 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 (5a)–(5c)). If all the receivers get the message and send a reply, then no choice is made and all the receivers are preempted (Rule (5d)). 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.

The channel declaration construct: The DSystemJ channel declaration construct has the same semantical rewrite rules as the SystemJ channel declaration statement. 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 implicitly. Also, unlike SystemJ, DSystemJ allows new channel declarations at runtime. Rule (6a) declares an input channel, while (6b) declares an output channel.

$$\bullet\texttt{channel } C\ \bar{p}, data \xrightarrow[E,E_c \cup S_{rc}]{\perp,\perp} \texttt{channel } C[S_{rc} \leftarrow 0], data' \bullet \bar{p} \quad (6a)$$

$$\bullet\texttt{channel } C\ \bar{p}, data \xrightarrow[E,E_c \cup S_{wc}]{\perp,\perp} \texttt{channel } C[S_{wc} \leftarrow 0], data' \bullet \bar{p} \quad (6b)$$

C. Reactivity

Informally, a DSystemJ CD is reactive if, for every given input signal and channel sensitivity sets, there is at least a single macro-step transition that results in the production of an output signal set. Note that this output set might be empty.

Formally, given a DSystemJ CD d_m, an input signal sensitivity set E_{d_m}, and a channel sensitivity set E_{cd_m}, d_m always takes a transition $\overset{e_{d_m},k_{d_m}}{\underset{E_{d_m},E_{cd_m}}{\hookrightarrow}}$, where $k_{d_m} \in \{0,1\}$. The definition for the reactivity of a SystemJ CD is identical.

Theorem. *Every DSystemJ CD is reactive.*

Proof sketch. The proof for reactivity of a SystemJ CD is based on the structural induction on the micro-step rules, and requires proving that every micro-step transition $\rightarrow$, contained in the macro-step $\hookrightarrow$, finishes with a termination code of $k \in \{0,1\}$. The simple but lengthy proof that a SystemJ CD is reactive is given in [24]. Then, as the DSystemJ MoC is defined in terms of SystemJ, all DSystemJ kernel constructs are rewritten into SystemJ constructs (Section IV-B2). Thus, by implication, every DSystemJ CD is reactive. ■

Rendezvous semantics (Rules (4a)–(5d)) of DSystemJ in conjunction with those of SystemJ [16] are essential in maintaining reactivity in the presence of multi-participant non-linear rendezvous. As shown in the rendezvous rules, `send` and `receive` constructs when blocked still take a micro-step transition (Rule (5a)), producing a termination code k_{d_m} of 1. Intuitively, in the general case of an automaton, this is equivalent to having an implicit self-transition in each state. In the particular case of DSystemJ, this is equivalent to the reactive `await` statement waiting for an input signal. Indeed, the rendezvous in DSystemJ and SystemJ is implemented using the `await` statements working on the channel sensitivity set E_{cd_m}.

V. Compilation and Implementation

DSystemJ provides a number of abstractions for programming dynamic distributed systems with inherent mobility. Thus, it is essential that a DSystemJ program be compiled correctly to low level code. Formal semantics describes the exact behavior of the DSystemJ programs and hence, makes writing compilers easier and less error prone. Particularly in our case, the micro-step semantics also helps us generate an intermediate format that is amenable to optimizations and verification/real-time analysis using model checking approach.

A. Compiling a DSystemJ program into Java code

In this section we present the operational semantics of the compiled DSystemJ program to Java code, which is amenable to formal verification and real-time analysis using a model checking approach.

Definition. *A Java program is a machine $M_J := (J, \Delta_J, s_0)$ where, J are the states, Δ_J are the transitions, s_0 is the starting state, and state*
$s := (g_1, g_2, \cdots, g_n, T_1, T_2, \cdots, T_l, M_1, M_2, \cdots, M_p)$.
Here we have:

1) *Function $Global := [g_1, g_2, \cdots, g_n]$ is the set of global static variables. Thus, $s.Global[i]$ is the i^{th} static global variable of the Java program in state s.*
2) *Function $ThreadMap := [T_1, T_2, \ldots, T_l]$ is the function that maps the index to the thread local variables including references (all as integers). The first position holds the thread location (-1 for uninitialized or completed thread). Thus, $s.ThreadMap[i](j)$ is the j^{th} local variable of the i^{th} thread in state s.*

3) *Function* $InstanceMap := [M_1, M_2, \cdots, M_p]$ *maps the index to the state of each allocated instance of the class in the program. Thus,* $s.InstanceMap[i](j)(k)$ *is the* k^{th} *field of the* j^{th} *instance of the* i^{th} *class and* $s.ThreadMap[i](j).InstanceMap[\ell](m)(n)$ *is the* n^{th} *field of the* m^{th} *instance of the* ℓ^{th} *class pointed to by the* j^{th} *local of the* i^{th} *thread and so on and so forth.*

Encoding the DSystemJ program into Java code is defined by Rule 7:

$$\begin{cases} \Delta_J \subseteq J\ X \rightsquigarrow X\ J \\ \rightsquigarrow \subseteq \{\hookrightarrow X \hookrightarrow \dots\} \\ \rightsquigarrow := j_1 \xrightarrow[\overrightarrow{E_1}, \overrightarrow{E_{c1}}]{\overrightarrow{O_1}, \overrightarrow{k_1}} j_1' \dots j_i \xrightarrow[\overrightarrow{E_i}, \overrightarrow{E_{ci}}]{\overrightarrow{O_i}, \overrightarrow{k_i}} j_i', \overrightarrow{k_i} \in \{0,1\} \end{cases} \tag{7}$$

where,

- $j_i \in \{s.ThreadMap(i)\}$ represents the DSystemJ syntactic constructs.
- $\rightharpoonup \subseteq \{\rightarrow X \rightarrow \dots\}$ is the interleaved partial evaluation of micro-steps ($\rightarrow$) as defined in Section IV and [16], [24].
- Set $\overrightarrow{E_i} \in s.InstanceMap[i](j)(k)$ is the union of signal sensitivity sets for all CDs in a DSystemJ program as defined in Section IV.
- Set $\overrightarrow{E_{ci}} \in s.InstanceMap[i](j)(k)$ is the union of all channel sensitivity sets for all CDs in a DSystemJ program as described in Section IV.
- Set $\overrightarrow{O_i} \in s.InstanceMap[i](j)(k)$ is the union of all output signals sets for all CDs in a DSystemJ program as described in Section IV.
- Set $\overrightarrow{k_i} \in s.InstanceMap[i](j)(k)$ is the union of all termination codes for all the CDs in the DSystemJ programs as described in Section IV.

As can be seen from Rule (7), every DSystemJ CD is compiled into a Java thread. All synchronous parallelism ($\|$) is compiled away to form single threaded code, like in Esterel. All signal and channel sets are compiled into Java fields. These Java threads do not share any global variables, i.e., all communication is carried out via channels (implemented as sockets) as described in [22]. The generated Java program proceeds by interleaving the micro-steps of all the active CDs. Overall, the complexity of the compiled Java program is a CCS [18] cross product of the CD macro-transitions and hence, exponential.

Lemma. *$\rightsquigarrow$ is reactive when all CDs (macro-transitions $\hookrightarrow$) are reactive, but it need not be deterministic even if all $\hookrightarrow$ are.*

Proof. According to the theorem in Section IV and [24], all CDs ($\hookrightarrow$) are reactive and deterministic. According to Rule (7), $\rightsquigarrow$ transition is the result of a CCS style cross product of $\hookrightarrow$ for all CDs in a DSystemJ program. ■

DSystemJ programs and more importantly its subset the SystemJ programs can be formally verified for correctness (e.g., safety and liveness properties) using a model-checker like *Java Path Finder* (JPF) [25]. We use the aforementioned Java encoding of DSystemJ programs for verification using JPF.

- *Formal reasoning and verification*: Formal verification of a DSystemJ program using a model-checking engine, like any other asynchronous language, suffers from problems of state space explosion (due to interleaving of the CCS product). But, the macro-step transition rule (Rule (7)) helps reduce the state space compared to verifying plain asynchronous languages like Java. A macro-step transition ($\hookrightarrow$) can be used for state demarcation. The macro-step transition hides the partial function evaluations ($\rightharpoonup$) along with all Java variables. It exposes only the interface signals/channels that can then be observed for correct program behavior. Model-checking a Java program creates a larger state space compared to a DSystemJ program, as every execution of a statement amounts to a new state transition. Note that execution of a statement is equivalent to a partial function evaluation ($\rightharpoonup$ transition). To overcome such problems, many model-checkers take the approach of arbitrarily demarcating a state boundary, thereby making the model-checking process heuristic. Granted that formally verifying a program at the partial evaluation level gives much better granularity, such fine granularity is not necessary in all situations. Besides, any required variables and micro-steps can be observed using the macro-step transition by using dummy interface signals, containing the Java variables, emitted along with the partial evaluations.
- *Real time analysis*: Every reactive DSystemJ CD follows the perfect synchrony hypothesis, which states: *CD reactions are instantaneous*. In practice, a CD always completes its reaction before the next input signal set is presented by its environment, which is equivalent to saying that a CD always needs to meet its deadline. The model-checking approach described in the previous point can also be utilized for *Worst Case Reaction Time* (WCRT) analysis. The macro-step transition along with the rendezvous rules presented in this article play a crucial role in simplifying the WCRT analysis. These rendezvous rules allow a CD to proceed with its transition even while synchronizing with other CDs (thanks to the `await` statement). Furthermore, rewriting the `run` statement as a `send`/`receive` construct simplifies the analysis, as dynamic process creation only happens at a macro-step transition. Hence, it is possible to observe CD creations and relate them directly to a tick and consequentially to time.

Finally, we would like to mention that although we allow recursion and CD instantiation, these features need to be bounded when performing model checking. One approach is to extract finite sub-sets of DSystemJ programs, i.e., programs that do not instantiate new CDs infinitely. The DSystemJ compiler can extract such programs, but this would require the compiler to over-approximate the infinite sub-sets of instantiated CDs due to conditional constructs. For example given a program like such as:

```
R->{present(A){ run R(); }}
```

the compiler would *over approximate* this to be an infinite sub-set as it is impossible to determine during compilation the number of times the `present` statement will evaluate

as `true`. Another approach is to bound the CD instantiation at runtime. With JPF, this can be achieved by using the provided *Application Programming Interface*, which disallows or throws an error after a certain bound.

DSystemJ's syntactic constructs are re-written into SystemJ constructs and hence, we follow the same compilation mechanism as in SystemJ. The micro-step transitions presented in Section IV lead to a direct intermediate representation, called the *Asynchronous GRaph Code* (AGRC). AGRC is an asynchronous extension of the *GRaph Code* (GRC) format used to compile Esterel. A complete description of the AGRC compilation format is out of the scope of this article and the reader is referred to [24], [16], [22].

Although DSystemJ uses the same compilation scheme, it differs from SystemJ as it requires runtime and library support to execute. We describe the DSystemJ library and runtime system briefly in the next section.

B. DSystemJ Runtime system and library support

The runtime system and library provide support for programming dynamicity in DSystemJ. The runtime system is responsible for (1) implementing the `run` statement and (2) garbage collecting the CDs, which have terminated their execution. The library essentially implements the rendezvous protocol (`send` and `receive`) as described in Section IV-B2, and provides convenience functions for writing DSystemJ programs.

The DSystemJ runtime system and library are written in the formal programming language SystemJ. This approach makes the DSystemJ language and runtime/library amenable to formal reasoning and verification. For a complete description of the components and implementation of the runtime system and library please, refer to [22].

VI. Experimentation results

Currently, only the JADE multi-agent system platform is capable of competing with DSystemJ, in terms of features and available implementation. ULM, does not have an implementation. The released version of ActorFoundry does not support distributed processes. Occam, and Occam-π do not provide support for complex data-transformations. Finally, JoCaml and RML do not support process mobility. Thus, we quantitatively compared DSystemJ only with JADE.

Table III shows the lines of code (LOC), memory footprint, and runtime comparison of a large security surveillance system. The runtime memory footprint consumption is shown in Figures 2 and 3. The security surveillance system is as presented in Section III and is extended to monitor and control 100 cameras. This example shows abstraction and scalability of our approach. This example was implemented on two physical machines, 64-bit and 32-bit Intel processors with 2 GB of RAM, and with two 2.3 GHz cores each. For the security surveillance system, DSystemJ has 3.7 times smaller compiled memory footprint compared to the JADE implementation (Table III (a)). Similarly, DSystemJ has on average 6.75 times faster runtime compared to JADE (Table III (b)). The exhaustive comparison results of DSystemJ with JADE are presented in [22]. In general, DSystemJ programs have a much smaller memory footprint, being on

Table III: DSystemJ Vs JADE

(a) Lines of Code and Compiled Memory footprint: DSystemJ Vs JADE

Comparison metrics	DSystemJ	JADE	Ratio (JADE/DSystemJ)
LOC	1464	1777	1.213
Memory footprint (KB)	860	3184	3.7

(b) Runtime: DSystemJ Vs JADE

DSystemJ (ms/tick)				**JADE** (ms/tick)			
CD1	CD2	CD3	CD4	CD1	CD2	CD3	CD4
342.7	491.6	165.4	123.2	3201.4	1376.78	1426.6	1578

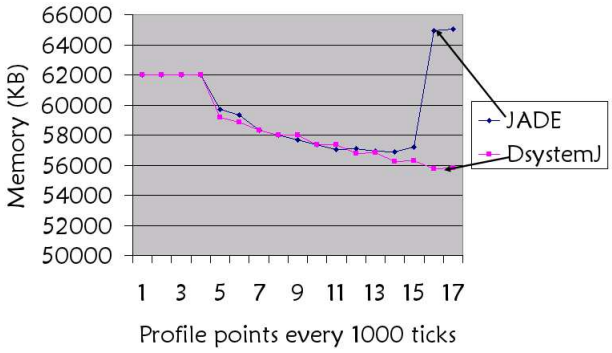


Figure 2: Single machine, multi-core implementation of the security surveillance system with 100 cameras

average 13 times smaller than JADE programs. Similarly, DSystemJ programs have faster runtimes compared to their JADE counterparts. In our benchmarks, DSystemJ programs were 20 times faster on a single (2-core) machine and 12 times faster in a distributed setting with 2 machines and 4 cores. Our benchmarks and case studies along with the DSystemJ compiler and runtime system are available from [26].

DSystemJ provides a much more efficient compilation and runtime environment thanks to the optimized code generated from the AGRC intermediate format, recall that the runtime environment is written in the SystemJ language, which itself is compiled using the AGRC format. DSystemJ communication is also optimized for a single physical machine implementation. JADE on the other hand uses a number of complex Java features along with the very complex FIPA [27] protocol, which substantially increases runtime and the memory footprint. Similarly, DSystemJ's abstract representation of process forking, mobility, and reactive constructs for data fusion help reduce the overall code written by the programmer, which in turn reduces debug and maintenance time.

The DSystemJ implementation also presents a consistent runtime memory footprint, profiled using the `-Xprof` JVM switch, along all the profile points in time (Figures 2 and 3), whereas with JADE the memory consumption keeps on increasing. Overall, DSystemJ program is more scalable compared to the JADE program.

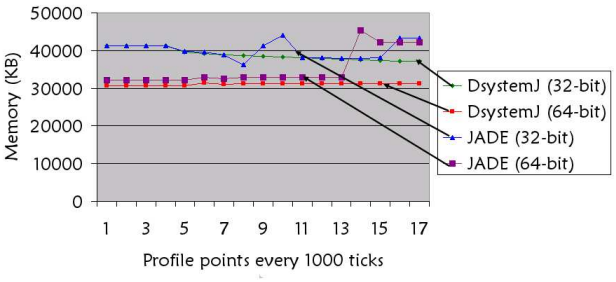


Figure 3: Distributed implementation of the security surveillance system with 100 cameras

VII. Conclusions

We have presented a new language called DSystemJ, which is targeted towards modelling and implementing dynamic distributed systems, such as sensor networks and collaborative environments. DSystemJ is based on the *Globally Asynchronous Locally Synchronous Model of Computation*, with the ability to abstractly model reactivity, data fusion from sensors, dynamic creation of asynchronous processes at runtime, and process mobility via message passing. Combined together, all these features provide a very powerful language and paradigm for implementing complex distributed and dynamic systems. We also provide the micro-step and macro-step semantics of DSystemJ, which are amenable to formal verification and *Worst Case Reaction Time* (WCRT) analysis using the model-checking approach.

DSystemJ implements message passing via channels for communicating between asynchronous clock-domains. In a multi-participant non-linear channel communication scenario, a non-deterministic choice operator is used to choose a rendezvous partner; this method is well suited for communication between distributed components, with the aim of having no single entity with the complete knowledge of the system, and thus no single point of failure.

References

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

[2] F. L. Bellifemine, G. Caire, and D. Greenwood, *Developing Multi-Agent Systems with JADE*. Wiley, 2007.

[3] "The Jade website," http://jade.tilab.com [Last Accesses: 16/03/2010].

[4] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Inc, 2004.

[5] G. Boudol, "ULM: A Core Programming Model for Global Computing," in *European Symposium on Programming, ESOP'04*, ser. LNCS, vol. 2986. Barcelona, Spain: Springer-Verlag, April 2004, pp. 234–248.

[6] "ActorFoundry," http://osl.cs.uiuc.edu/af/ [Last Accesses: 23/03/2010].

[7] L. Mandel and M. Pouzet, "ReactiveML: a reactive extension to ML," in *PPDP*. New York, NY, USA: ACM, 2005, pp. 82–93.

[8] G. Berry, S. Ramesh, and R. K. Shyamasundar, "Communicating reactive processes," in *POPL '93*. New York, NY, USA: ACM Press, 1993, pp. 85–98.

[9] J. Galletly, *Occam-2*, 2nd ed. University College London Press, 1996.

[10] P. Welch and F. Barnes, "Communicating Mobile Processes: introducing Occam-pi," in *25 Years of CSP*, ser. Lecture Notes in Computer Science, A. Abdallah, C. Jones, and J. Sanders, Eds., vol. 3525. Springer Verlag, April 2005, pp. 175–210. [Online]. Available: http://www.cs.kent.ac.uk/pubs/2005/2162

[11] L. Mandel and L. Maranget, "Programming in JoCaml – Extended version," INRIA, Tech. Rep. 6261, 2008.

[12] J. Harrop, *OCaml for Scientists*. Flying Frog Consultancy Ltd, 2005.

[13] G. Berry, "The semantics of pure Esterel," in *G. Berry. The semantics of pure Esterel. In Proc Marktoberdorf Intl. Summer School on Program Design Calculi, LNCS, Springer-Verlag.*, 1993. [Online]. Available: \url{http://citeseer.ist.psu.edu/berry93semantics.html}

[14] F. Boussinot and R. de Simone, "The SL Synchronous Language," *IEEE Transactions. Software. Eng.*, vol. 22, no. 4, pp. 256–266, 1996.

[15] E. Vecchié, J.-P. Talpin, and S. Boisgérault, "A higher-order extension for imperative synchronous languages," in *Proceedings of the 13th International Workshop on Software & Compilers for Embedded Systems*, ser. SCOPES '10. New York, NY, USA: ACM, 2010, pp. 7:1–7:10. [Online]. Available: http://doi.acm.org/10.1145/1811212.1811222

[16] 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*, vol. 36, no. 4, pp. 317–344, December 2010.

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

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

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

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

[21] 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*. New York, NY, USA: ACM, 1996, pp. 372–385.

[22] A. Malik, A. Girault, and Z. Salcic, "The DSystemJ programming language for dynamic GALS systems: it's semantics, compilation, implementation, and run-time system," INRIA, Tech. Rep. 7346, July 2010.

[23] D. Potop-Butucaru, "Optimisations for Faster Execution of Esterel Programs," Ph.D. dissertation, Ecole des Mines de Paris, 2002.

[24] A. Malik, "Principia Lingua SystemJ," Ph.D. dissertation, University of Auckland, 2010.

[25] K. Havelund and T. Pressburger, "Model checking Java programs using Java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.250

[26] "The DSystemJ website," http://dsystemj.gforge.inria.fr [Last Accesses: 02/04/2010].

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