SystemJ Compilation using the Tandem Virtual Machine Approach

AVINASH MALIK, ZORAN SALCIC and PARTHA S ROOP
University of Auckland

SystemJ is a language based on the Globally Asynchronous Locally Synchronous (GALS) paradigm. A SystemJ program is a collection of GALS nodes, also called clock domains and each clock domain is a synchronous program that extends the Java language. Initial compilation of SystemJ has been to standard Java executing on a Java Virtual Machine (JVM), which is both inefficient and bulky for small embedded systems. This paper proposes a new approach for compiling and executing SystemJ using a new type of virtual machine, called a Tandem Virtual Machine (TVM). The TVM approach provides an efficient implementation of SystemJ on both standard processors and resource constrained embedded processors. The new approach is based on separating the control-driven and data-driven operations for execution on two virtual machines. While the JVM executes the data-driven operations, a Control Virtual Machine (CVM) is introduced to execute the control-driven parts of a SystemJ program. The TVM approach is capable of handling all data-driven and control-driven operations required by the GALS model. The benchmark results show that the TVM has code size improvements of over 60% on average and also a substantial improvement in execution speed over standard Java based compilation.

Categories and Subject Descriptors: D.3.4 [Processors]: Compilers
General Terms: Languages
Additional Key Words and Phrases: System level design, SystemJ, Compilation, Esterel, Virtual machines

1. INTRODUCTION

SystemJ is a new system level design language (SLDL) first proposed in [Gruian et al. 2006] based on the Globally Asynchronous Locally Synchronous (GALS) model of computation. SystemJ is unique amongst all currently available SLDLs. It has two main advantages;

(1) Support for very complex data-structures and computations as first class citizens of the language.

(2) The GALS model of computation, which allows designers to describe both purely synchronous and asynchronous programs.

As such it is suitable for programming complex systems that can be described as collections of synchronous nodes which communicate each with the other asynchronously on the top, system level. The language provides mechanisms for specification of synchronous nodes, their interaction with the environment using synchronous mechanisms and description of data transformation using standard Java, as well as the interaction between synchronous nodes using asynchronous communication.
1.1 Need for a new SystemJ execution platform

The first SystemJ implementation (also called original/old implementation) [Gruian et al. 2006] uses TReK, a multi-threaded reactive Java library, to run. Since then, we have developed (also called the current implementation) another approach based on translation to so called *Asynchronous GRaph Code* (AGRC) [Malik et al. 2006a] for SystemJ execution and a new approach, which is presented in this paper, that addresses major shortcomings of both original and current approaches. Figure 1 shows the three, old (TReK based), current (AGRC based) and the new (proposed) approaches for compiling and executing SystemJ.

The old approach (Figure 1a) utilizes TReK, a runtime Java library in conjunction with a translator from SystemJ to Java source. Besides missing support for important reactive constructs like strong signal based preemptions (aborts/suspends), TReK’s reliance on Java’s multi-threading introduces non-determinism not intended by the programmer. SystemJ program behaviour varies depending upon the underlying operating-system (OS) and the scheduling mechanism implemented by this OS.

The current compilation approach for SystemJ, shown in Figure 1b and briefly introduced in this paper, uses an intermediate format called *Asynchronous GRaph Code* (AGRC). The AGRC format is derived directly from the operational semantics of the SystemJ language and the compiler produces single threaded Java code at the backend. This AGRC format and the single threaded nature of the backend generated code guarantees correct program behaviour irrespective of the underlying platform, but suffers from drawbacks such as large generated code size and slow execution speed. In this paper we propose a new approach for SystemJ compilation and execution. This new approach as shown in Figure 1c consists of separating the Java data-processing and the SystemJ control constructs at the AGRC intermediate level and implementing them on a JVM and CVM (Control Virtual Machine) respectively. The new combined execution platform is called the Tandem Virtual...
SystemJ Compilation using the Tandem Virtual Machine Approach

Table 1. Comparison of various SystemJ design flow options

<table>
<thead>
<tr>
<th>Category</th>
<th>Compilation effort</th>
<th>Guarantees</th>
<th>Generated code size</th>
<th>Average tick time (µsec/tick)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old</td>
<td>Small</td>
<td>No</td>
<td>Small</td>
<td>High</td>
</tr>
<tr>
<td>Current</td>
<td>Medium</td>
<td>Yes</td>
<td>Large</td>
<td>Medium</td>
</tr>
<tr>
<td>New</td>
<td>Large</td>
<td>Yes</td>
<td>Medium</td>
<td>Low</td>
</tr>
</tbody>
</table>

Machine (TVM).

In the current compilation scheme (AGRC) the backend generated code emulates synchronous reactive concurrency using switch-case statements (every context switch to a sibling concurrent reaction requires at least two nested switch statements). This in general produces larger code size compared to compilers like Esterel-studio [Esterel Technologies SA 2003], v5 [Berry 1999] and CEC [Edwards 2002], which use goto’s to achieve the same kind of behaviour. Finally, the current compilation approach does not take advantage of the reactive constructs in SystemJ. All the reactive constructs are converted into plain Java statements and this further slows down the execution speed. Consider two concurrent synchronous reactions, first one containing intensive data-driven operations while second one purely reactive (control-driven). Due to the single threaded nature of the generated code, the first reaction needs to be completed before context switching to the second one. If we were able to execute data-driven operations concurrently to the reactive control parts, then the second reaction could be run simultaneously to the first one thereby shortening the logical clock-instant and increasing the execution speed. The main motivation for the new compilation and execution approach is to enhance performance (generated code-size and execution speed) of the language currently limited by the single threaded implementation on the JVM.

Our approach to compiling SystemJ is shown in Figure 1c and briefly introduced in [Malik et al. 2008]. We propose to separate the data-driven operations and control-driven operations at an intermediate level (unlike Esterel which does this at specification level) and utilize JVM with a Control Virtual Machine (CVM) for their execution, respectively. The combined execution environment is called Tandem Virtual Machine (TVM). This approach provides a number of advantages. Firstly, under the assumption that most reactive systems have a significant amount of control-driven operations, we can use custom instructions to provide both direct reactive support and direct execution support for the intermediate format. Since the CVM is built using C, executing instructions customized for control driven part, the execution overhead of JVM is avoided in general (JVM is called by the CVM only when data-driven computations are necessary). Also, with this approach an operating-system (mostly required to run JVM) becomes redundant and converting Java data-driven operations into native code becomes unnecessary. Table I gives a brief comparison between the three SystemJ implementation approaches.

1.2 Contributions

The main contributions presented in this paper are;

1. Novel execution environment for reactive extensions to Java: An execution environment unlike any currently available reactive processor or virtual machine,
providing direct reactive support with complete access to all Java data-driven operations. The proposed virtual machine, for the first time, supports direct execution of reactive constructs (through a control virtual machine) that also coordinates the execution of the standard JVM. This approach thus not only reduces the code size drastically, it also improves the runtime efficiency. Such an approach thus paves the way for Java based design of memory and power constrained embedded systems. The proposed approach is designed to be modular enough to be extended to support not only Java but other programming languages like C.

(2) Tool chain: A complete tool-chain support including the compiler, assembler, simulator and an integrated debugger which ease the programming burden unlike the above mentioned approaches (Section 7)

(3) A new approach to compilation of a GALS language: The paper proposes a novel intermediate format for compiling a GALS language and also proposes a novel approach to generate code from this intermediate format onto the TVM platform.

The rest of the paper is organized as follows. Section 2 shows an example SystemJ system. Section 3 introduces the SystemJ terminology and also provides a detailed description of the SystemJ Model of Computation (MoC). Section 4 gives an introduction to the intermediate format which is utilized for backend code generation. Section 5 introduces the Tandem Virtual Machine (TVM) and also explains its various components. Section 6 presents the benchmark results obtained from comparing the pure Java and the TVM implementations. Section 7 provides a detailed comparison of SystemJ with other models of computations and languages. Lastly, we conclude the paper in Section 8.

2. A SYSTEMJ EXAMPLE

The SystemJ program consists of synchronous reactive processes called reactions, having individual reactive interface with the external environment. A collection of such reactions composed concurrently and synchronously (i.e. all reactions progress in lock-step with the same clock) is called a clock-domain. Clock-domains execute concurrently but asynchronously with other clock-domains. Clock-domains communicate asynchronously using channels which utilize CSP style rendezvous [Hoare 1985].

In this section we use a small SystemJ example to introduce the SystemJ model of computation. Figure 2 shows the access privilege model of an engineering calculator which is designed in SystemJ. The engineering calculator is designed for exams and allows the user to execute commands only if authorized to do so. The calculator is designed to provide varied functionality depending upon the exam situations. As an example, a student is able to execute all the commands during off hours (when not in an exam situation). During exams certain functionality like graphing functions can be restricted. The calculator takes the user-id and the command to execute as inputs from the user and communicates with a server to check if the student at a given point in time is authorized to carry out the requested commands. The actual SystemJ code for the calculator example is shown in listings 1-3.
SystemJ Compilation using the Tandem Virtual Machine Approach

Fig. 2. Abstract view of an Engineering Calculator model in SystemJ

Listing 1. “UserNode” reaction

```java
import perm.*;
import exec.**;

generated reaction UserNode(:
    output char channel TId, 
    output char channel SCmd, 
    input char signal uId, 
    input int channel Auth, 
    input char signal Cmd, 
    input signal escape, 
    output signal success, 
    output signal failed)
{
    while(true){
        abort(escape){
            await(uId);
            char userid = #uId;
            {send TId(userid);} || {await(Cmd);}
            //Obtaining the value and sending through channel SCmd
            char cmd = #Cmd; send SCmd(cmd);
            receive Auth;
            int auth = #Auth;
            if(auth == 1){
                Exec execute = new Exec(cmd);
                execute.execute();
                emit success;
            } else
                emit failed;
        }
        pause;
    }
}
```

Listing 2. “OsNode” reaction

```
reaction OsNode:(input char channel TId,input char channel SCmd,
output int channel Auth){
while(true){
  receive TId; char userid = #TId;
  pause;
  receive SCmd; char cmd = #SCmd;
  pause;
  Permission perm = new Permission();
  int ret = perm.getpermission(userid,cmd);
  send Auth(ret);
}
```

Listing 3. “system” reaction

```
system{
interface{
  output char channel TransferId, SendCmd;
  input char channel TransferId, SendCmd;
  output int channel Authorization; input int channel Authorization;
  output signal success,failed; input char signal UserId,Cmd;
  input signal escape;
}
//Initializing clock-domains UserNode and OsNode
{
  UserNode(TransferId,SendCmd,Authorization,UserId,
  Cmd,escape,success,failed)
  >>
  OsNode(TransferId,SendCmd,Authorization)
}
```

In the engineering calculator example, there are two clock-domains, the “UserNode” (calculator) and the “OsNode” (authentication server). The user can get services of the UserNode but before that the user has to provide authorization information. To begin with, the UserNode waits for a user-ID (listing 1 line 11). Once this is acquired, the clock-domain forks out two synchronous parallel reactions (listing 1 line 13), the first synchronous parallel reaction sends the user-ID to the server (OsNode), the second waits for a command to be input by the user. After receiving a command request from the user it sends this to the OsNode to check if the user is authenticated to carry out this command (listing 1 line 15). If the authentication is successful then the command is carried out, else it is rejected (listing 1 lines 17-25). This process can be reset any time by the user by pressing the escape button (listing 1 line 10).

The OsNode reaction just waits for the user-ID and the command request, once received, it looks up the database and sends the authentication information (listing 2 lines 4-10). The top-level system declares the signals and channels needed for communication and initializes the clock-domains (listing 3 lines 2-13). It should be
noted that the clock-domains are connected together using channels, TransferId, SendCmd and Authorization.

3. SYSTEMJ TERMINOLOGY AND MODEL OF COMPUTATION

In this section we describe the SystemJ terminology and the Model of computation used in SystemJ with the help of the student calculator example described in the previous section.

3.1 SystemJ Terminology

— A reaction: It is a synchronous process consisting of reactive statements over interface objects such as signals. Reactions can be composed together using the synchronous parallel operator (\(|\|\)). Reactions in the resulting composition run synchronously (at the same logical speed). In Figure 2 Reaction11 and Reaction12 are composed using the synchronous parallel operator. In listing 1 line 13 shows one such composition. The ‘UserNode’ (calculator reaction) communicates with the user using signals (User ID, Command). Signals in SystemJ are a combination of a status which can be checked for presence or absence and possibly a value (of any Java data-type).

— A clock domain: A clock-domain consists of at least one reaction or a combination of reactions composed using the synchronous parallel operator, running at its own pace. It has the capability to synchronize with other clock-domains. Two clock-domains which rendezvous at any given instant are said to be partner clock-domains. Every clock-domain has its own set of signals and channels which it uses to communicate with the environment and other clock-domains, respectively. Clock-domains do not share interface signals (communicating with environment) and channels. The only means of communication between synchronous parallel reactions within a clock-domain is using signals. While cross clock-domain communication requires channels. Sharing data structures between synchronous parallel reactions and clock-domains using semaphores and locking is prohibited in SystemJ, since, the signal and channel communication model is much more safer and easier to validate at a later stage. In Figure 2 UserNode (calculator) and OsNode (server) are two clock-domains.

— A system: A SystemJ program is called a “system”. It is a collection of clock-domains which can communicate in pairs using rendezvous on channels. A system is used to declare clock-domains, input/output signals which communicate with the environment and channels for cross clock-domain communication. Figure 2 and listing 3 show an example of a SystemJ system. The clock-domains (UserNode, OsNode) are composed using the asynchronous parallel (\(>\)) operator. There are two parts to a system reaction. The first one is the interface used to declare the signals and channels (listing 2 lines 2-8). The second is the body of the reaction used for clock-domain initialization (listing 2 lines 10-15).

— A channel: Channels are the only means of communication between separate clock-domains. They are used to synchronize and send data between two partner clock-domains. Channels are fixed and point-to-point, for instance, the TransferID channel can only be used once to transfer the user-ID and nothing else between the UserNode and the OsNode clock-domains in the calculator exam-
ple. Every channel is endowed with statues and value buffers which are used to implement CSP style rendezvous communication [Hoare 1985]. Our rendezvous implementation is based on blocking reads and writes. We describe our rendezvous algorithm in Section 3.3.

3.2 The SystemJ asynchronous model and clock-domain scheduling

Clock-domains in SystemJ are reactions which are composed using the \( \triangleright \triangleright \) (asynchronous parallel) operator. These clock-domains, unlike the synchronous parallel reactions, execute at unrelated speeds (i.e., they have unrelated logical instants of time). Each clock-domain can be considered as a synchronous island, running at its own clock “tick”. These clock-domain “ticks” are completely unrelated. All the synchronous parallel reactions within the clock-domain react simultaneously and instantaneously, computing and emitting their outputs in “zero-time”, and then quiescent until the next “tick”. Thus each clock-domain follows the classical Finite State Machine (FSM) model. At the start of the reaction each clock-domain reads in an input signal snapshot from the environment (input signals are not shared amongst clock-domains), the transition depends upon the read input vector. At the end of the transition the clock-domain changes state and emits a set of output vectors to the environment (the output vectors are not shared between clock-domains). For example, UserNode clock-domain (listing 1) reads the signals \( UserID \) and \( Command \) at the start of the reaction and emits the output signals \( Execute/Failed \) at the end of the transition. In SystemJ terminology we call the finishing of the reaction/transitio as the \( \text{End Of Tick} \) (EOT) for a clock-domain.

In the current implementation, clock-domains are scheduled cyclically. In cyclic scheduling each clock-domain completes one EOT communicates with the environment by reading input signals and emitting output signals before the next one is run. The pseudo code in listing 4 shows the scheduling policy for clock-domains as implemented for the calculator example in Section 2.

Listing 4. “Scheduling Clock-domains”

```cpp
UserNode: /* Reactive code for UserNode clock--domain */
1 ..
2 ..
3 /* Check if transition is over, EOT is finished */
4 if(EOT[0] == 1 || EOT[0] == 0){
5    /* Emit the output signal vector to the environment
6       Setting one OutputPort Signal at a time
7 */
8    for(i in EmitedSignals){
9        for (j in OutputSignalPorts){
10           if(i == j){
11              j.setValue(i.getValue());
12              j.setStatus(i.getStatus());
13           }
14        }
15    }
16 /* Just store a snapshot of all the input signal ports for this clock--domain*/
17 goto main; /* goto scheduler */
```

else goto UserNode; /* else rerun */
OsNode: /* Similar to UserNode */

/* The main scheduler code scheduling clock-domains in round-robin */
main:
if(EOT[0] == 0 || EOT[0] == 1){
  EOT[0] = -1 /*reset EOT for UserNode */
goto OsNode; /*Run the next clock-domain */
}
if(EOT[1] == 0 || EOT[1] == 1){
  /*Run next clock-domain or wrap to first if this is the last clock-domain*/
...

The pseudo code, as shown in listing 4 (lines 5-17), for communication between environment and clock-domains is inserted by the compiler at the end of the reactive code of individual clock-domains. The main scheduler runs the clock-domains in a round-robin policy. The scheduling policy of the main scheduler can be easily changed to a weighted cyclic scheduling policy or some other policy where clock-domains are executed for different number of time instants. In the case of weighted cyclic scheduling, the designer provides the speed of clock-domains as a ratio. For example, in the above pseudo code the designer can provide a ratio of 2:1 for the two clock-domains. This would lead the main scheduler to run UserNode clock-domain twice before calling the OsNode clock-domain (line 26), while the second clock-domain is run only once. For clock-domains with unrelated speeds and distributed on multiple processors or control virtual machines the main scheduler becomes unnecessary. Individual clock-domains distributed on separate processors or control virtual machines can execute in a loop without calling the main scheduler (line 18) after the completion of their EOT.

3.3 Rendezvous implementation in SystemJ

Communication between clock-domains is carried out using channels and CSP style rendezvous. Channels are point to point and utilize a two phase handshake protocol to synchronize clock-domains. A rendezvous attempt between partner clock-domains is considered successful only if the two rendezvousing clock-domains synchronize and the data-transfer if any succeeds.

Let us consider the calculator example and sending of the UserID using the TransferID channel from the UserNode to the OsNode. As shown in listing 3 lines 3,4,11 and 14 there are two TransferID channels corresponding to the sending and receiving clock-domains respectively. Since UserNode is the sending clock-domain the output TransferID channel is passed as an input to the UserNode clock-domain, while the input TransferID channel is passed in to the OsNode. Every output/input channel during implementation is decomposed into signals and a value buffer.

Figure 3 shows the decomposed output/input TransferID channel. The output TransferID channel is decomposed into a write-send signal which signifies that the sending clock-domain is ready to synchronize and send the data if any. The input TransferID channel gets decomposed into a read-send signal which gives back an
acknowledgement when the receiving clock-domain reads the data from the buffer. Both the sending and the receiving clock-domains sample the partner clock-domain signals at the start of their ticks with the rest of the environment signals. The sending clock-domain stores the acknowledgement signal as write-receive, while the receiving clock-domain stores the sampled write-send signal as read-receive.

During processing the UserNode reaction, when the control point reaches the “sendTId(userid)” construct (listing 1 line 13), the write-send signal is emitted and the buffer is filled with the “userid”. Next, the UserNode reaction awaits for an acknowledgement from the OsNode. This blocking mechanism utilizes the synchronous await construct on the write-receive signal. On the other hand when OsNode reaches its receiving construct (listing 2 line 4), it blocks (again using await) if the read-receive signal is not set, else, it reads from the buffer and emits the read-send signal. The UserNode samples the incoming read-send signal with the rest of the input signals at the start of its tick and completes the rendezvous if this signal status is set. The blocking mechanism using the await construct differs from KPN [Kahn 1974] as it does not stretch the logical instant, instead, if the signal status is not set it finishes the tick for that reaction. This allows us to proceed with the other synchronous parallel reactions if any. These semantics are similar to channel probing, which makes SystemJ inherently non-deterministic. This non-deterministic behaviour is an essential feature for designers who want to model systems such as communication protocols. Another useful specification mechanism in SystemJ is the ability to preempt an ongoing or uninitialised rendezvous. There are more signals and semantic rules associated which deal with these strong pre- emptions. As the semantics of the language is not the highlight of this paper, the reader is referred to [Malik et al. 2006b] to get the complete description of the rendezvous semantics.
3.4 Deadlock detection algorithm in SystemJ

The biggest disadvantage of rendezvous is the fact that the designer can unknowingly introduce deadlocks. The current SystemJ compiler is deadlock aware. The compiler uses the AGRC for circular deadlock detections. A circular deadlock, as described by [Murata et al. 1989], is a deadlock due to a set of communication statements, each in a separate clock-domain, mutually suspending each other and, thus, also their respective control flows. In this section we describe the deadlock algorithm which is implemented in SystemJ.

We will utilize two relatively complex examples to detail our deadlock detection algorithm. The fundamental requirement for this deadlock detection algorithm is the use of point to point channels. As stated previously, in SystemJ same named input/output channels cannot be used more than once in the whole program.

![Deadlock Detection Diagram](image)

**Fig. 4.** Example SystemJ program with transition diagram for deadlock detection

Figure 4a shows an example SystemJ program. Both the clock-domains have two synchronous parallel reactions and send/receive statements spread throughout trying to perform multiple rendezvous'. We use order relationships on the occurrence of channels in the intermediate format to detect channel deadlocks. For each reaction in the clock-domain we make a strict partial order set while traversing the intermediate format. In the above example for the two clock-domains the partial order sets would be, $CD1R1 = \{C1, C2\}$, $CD1R2 = \{C3, C4\}$, $CD2R1 = \{C4, C2\}$ and $CD2R2 = \{C3, C1\}$ respectively. Here, the first element of the set precedes the second and so on. Once these sets are built we emulate the rendezvous sequence.
of the system using a state transition diagram as shown in Figure 4b. These FSMs are built using the strict order set for each reaction.

The beginning FSM state is built using the first elements from the order relation sets. For example, for the first clock-domain the first element from both the sets (CD1R1, CD1R2) makes up the starting state, same for the second clock-domain. Please note the states formed are same as the cross product of the synchronous parallel reactions (the obvious result of running reactions concurrently and instantaneously). Once we have the two states for the two clock-domains, the common channel is deleted (since a shared channel name would indicate a successful rendezvous). In this example, channel “C3” is deleted from the state transition diagram. Next we make a new state by taking the next element from the order relation set which follows the deleted element. For example, for the first clock-domain the order relation CD1R2 dictates that the next rendezvous attempt after channel C3 would be on channel C4, while for the second clock-domain the next attempt would be on channel C1 (CD2R2). Once the second states are built we again remove the common names. In this example, both channels “C4” and “C1” are deleted (the order of deletion does not matter we can delete C1 and then C4 or vice-versa). Once the common channel names are deleted, we again look at the partial order sets to find the next channel name, in this case it is C2. This process is continued until we have run out of elements in the partial order sets. If we reach a state where clock-domain FSMs hold uncommon channel names, it indicates a possible deadlock, and such systems are rejected by the compiler. An example of such a SystemJ program and its resultant state transition graph is shown in Figure 5.

Fig. 5. Example systemJ program with transition diagram for deadlock detection

The partial order sets for the two clock-domains in example Figure 5a are CD1 =
\{C_1, C_2\} and \(CD_2 = \{C_2, C_1\}\) respectively. It should be noted that while building these sets we abstract out the conditional branches. For example, in the second clock-domain although we have \texttt{abort} (preemptive) construct enclosing the rendezvous construct \texttt{receive C2}, while building the order sets we consider that the control point would always first hit this receiving construct before the \texttt{receive C1} statement. This approach is much more conservative and rejects programs even though they are deadlock free.

The SystemJ program in Figure 5a is potentially a deadlock free program. In the first clock-domain the control point hits the \texttt{send C1} construct. The \texttt{send C1} construct blocks waiting for an acknowledgement from its partner clock-domain. For the second clock-domain the control point hits the \texttt{receive C2} construct. Channel \(C_2\) now blocks any further processing, waiting for the buffer to be full (which can happen only after the rendezvous on channel \(C_1\) succeeds). Thus channels \(C_1\) and \(C_2\) suspend each other. If in any instant of time signal A is present then, the rendezvous attempt on channel \(C_2\) is aborted and the second clock-domain now proceeds to processing \texttt{receive C1}. This leads to \texttt{send C1} getting an acknowledgement. The control point for the first clock-domain then hits \texttt{send C2}. This send construct never initializes a rendezvous following the preemption semantics described in [Malik et al. 2006b]. Finally, the program finishes without any deadlocks. Such conditional deadlocks cannot be detected at compile time and hence, we take the conservative approach of rejecting the program completely.

4. SYSTEMJ INTERMEDIATE FORMAT

We have developed an intermediate format called Asynchronous GRaph Code (AGRC) for compiling SystemJ. The AGRC format captures the semantics of the language. The AGRC intermediate format is a substantial extension of the GRC format [Butucaru 2002] used to compile Esterel. AGRC representation of a SystemJ program is an ideal point from which various backends can be generated and target execution platforms explored. This section briefly describes the GRC format and our contributions to this format to capture the SystemJ’s GALS model of computation. We also describe the algorithm used to separate the Java-data driven computations from the control-flow.

GRC consists of two graphs; a \textit{hierarchical state graph} (HSG), which describes the structural information, and a \textit{control flow graph} (CFG), which describes the operational aspects explicitly. These two graphs handshake with each other to identify the point of control in a given instant. Figure 6 gives the GRC representation of the “UserNode” reaction from Section 2. These graphs are abstracted for clarity.

4.1 Hierarchical state graph (HSG)

The HSG outlines the structure of the synchronous reactions in terms of statements and threads of control. It consists of four different types of nodes. \textit{Thread} nodes (rectangles) represent the sequential composition of a particular thread. \textit{Parallel} nodes (triangles) represent forking of concurrent threads. Meanwhile, loops, aborts and other backtracking constructs are represented as \textit{compound} nodes (circles). Atomic states like pauses are represented with \textit{boundary} nodes (squares). The number of branches extending from a thread node indicates the number of possible states for that thread.
4.2 Control flow graph (CFG)

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

The CFG directly captures the semantics of a SystemJ program. Consider Figure 6 and the corresponding “UserNode” reaction from listing 1. All the state variables (S0, S1, etc.) are initialized to zero. In the first instant of time (surface behaviour) the control flow graph of the UserNode reaction enters the right most branch of the very first switch node encoded by ‘S0’, it sets the value of variable S0 to one, S1 to zero and finishes (pauses) by hitting the “termination” node (hexagon) with a value of one. The termination nodes fulfil two purposes, firstly, they guarantee lock-step processing of the synchronous parallel reactions within a clock-domain [Berry 1993]. Secondly, in our compilation scheme termination nodes are used for scheduling clock-domains. A value of one in the termination node shows that the corresponding reaction has completed the processing for that logical instant (pausing). A value of zero would mean termination of the reaction. Finally, a termination code of ∞ shows, that there are some signal dependencies which are not yet resolved within the clock-domain, this leads to rerunning of the clock-domain until
these signal dependencies are resolved. This scheduling mechanism is called cyclic scheduling and is derived from [Yoong et al. 2006]. The flow of CFG corresponds to the expected program behaviour from listing 1. The reaction UserNode is expected to enter the loop (listing 1 lines 9,10) and then pause on line 11. The abort and await statements do not check the signal status in the first instant of time (surface behaviour), the signal checks are performed only from the proceeding instants of time [Berry 1999].

In the next instant of time (depth behaviour), the switch nodes encoded by S0, S1 and S2 enter the left most branches. In this execution cycle the test-node checks if the escape signal is present; if true, the left branch is taken and the program pauses, else, execution control moves to the right branch. In the right branch again a test-node decides the control-flow; if the signal uId is present, then execution flow proceeds to the left branch else, takes the right branch, and pauses. This behaviour conforms to the expected program behaviour. In listing 1 line 10, if the abort statement is true, then the program jumps to line 21 and pauses. Otherwise, the program waits (await) for the uId signal to be input by the user. This way, the control flow graph explicitly executes the program behaviour. This control flow graph, which is later compiled into backend code executing on the Tandem Virtual Machine, guarantees equivalence between the program specification and execution.

4.3 Asynchronous modifications to the GRC

Fig. 7. AGRC representation of the server-client example

The GALS model of computation in SystemJ requires major extensions to the GRC format. We introduce further nodes to represent channel, send/receive and
“>>” constructs within the GRC. Channels are declared in the interface of the “system” reaction (Listing 3, lines 3,4). Channels have all their statuses and value buffers initialized to zero. Figure 7 shows the complete engineering calculator, a server-client example from Section 2. Channels are initialized with action nodes at the start of the program.

We introduce two new nodes the asynch-fork (vertex joint triangles) and asynch-join (base joint triangles) to represent the asynchronous clock-domains as shown in Figure 7. SystemJ follows a GALS model of computation and thus, semantically clock-domains run (start/resume) independently. In accordance with the clock-domain semantics, >> operator (asynch-fork node) is traversed only once in the first instant of time to start the various clock-domains. The resuming instants are all confined within the clock-domains. Within each clock-domain we use cyclic-executive scheduling for concurrent synchronous reactions [Yoong et al. 2006]. This scheduling mechanism is chosen rather than static scheduling (which involves topologically sorting the digraph in Figure 6 for sequential code generation) because it provides us with an opportunity to explore distributed architectures at a later stage. The asynch-join nodes play the role of tick boundary (EOT) (Section 3.2) for clock-domains. At EOT, the asynch-join node is utilized for scheduling: to make sure that clock-domains do not finish with ∞ (which shows unresolved data/signal dependencies) as their termination codes, to emit interface signals to the environment, and to read new input signals before resuming the next instant of that clock-domain.

4.4 Restructuring AGRC for the Tandem Virtual Machine

The Tandem Virtual Machine (TVM) is designed to take advantage of the control-driven constructs in SystemJ. The current single threaded backend implementation induces an overhead when large Java data-driven computations are intertwined with the reactive control flow. The TVM overcomes this limitation by separating the control-flow on the Control Virtual Machine (CVM) and the Java data-driven operations on the standard JVM, thereby allowing the control parts to proceed faster as data-driven computations are run asynchronously. The control-flow constructs include all the nodes in the CFG which do not represent pure Java computations. The TVM architecture requires us to partition the Java data-driven operations and the control-flow nodes in the CFG representation. We now describe the algorithm which performs this partitioning.

In the most basic form, the partitioning algorithm does the following:

— Does a depth first traversal of the CFG. Marks and groups all the pure Java action nodes thereby building a macro action node until it finds a data-control dependency.
— Assigns a unique id to the just formed macro action node.
— Inserts conditionals called the data-lock and the result test-nodes which guarantee consistency between the data and control flow.
— Moves to the next pure Java action node in the digraph.

We use a part of Figure 6 (encompassed in the dotted box and redrawn here as Figure 8a for convenience) to describe in detail the partitioning algorithm. While
building the AGRC format, all action nodes with pure Java data-computations, such as the test-node auth in Figure 8a, are marked as Java data-driven action nodes. Next, we check the emit statements, and if they are emitting values, the emit statement is decomposed into two action nodes: one setting the value and the other setting the status of the signal. In Figure 8a, none of the emit statements are decomposed or marked because they all emit pure signals. The AGRC is then traversed looking for such marked nodes and all simultaneously (those not separated by control-flow example, switch nodes and test-nodes) occurring marked action nodes are grouped together as one action node called a macro action node. For the current example from Figure 8a, the result of such grouping is shown in Figure 8b. The boxes in Figure 8b show the macro action nodes. There are two macro action nodes separated by the test-node (auth==1). The auth==1 test-node effects the control-flow of the program and thus prevents all the action nodes to be grouped together as one macro action node. During backend code generation stage for each clock-domain a separate Java switch-case statement encapsulated by a Java method is generated where the case number (N ≥ 1) represents the macro action nodes (Figure 8b) to be executed and the method represents the clock-domain being executed. These methods and switch-case statements are implemented on the standard JVM.

We utilize cyclic scheduling policies to schedule the data-computations between CVM and JVM. Additional test-nodes called data-locks are inserted after each
macro action node. Data-locks are implemented on the CVM and have input signal dependencies (i.e., these test-nodes check for signal presence before proceeding further; absence of these signals causes the reaction to complete with a termination code of $\infty$ and suspension of the reaction). In Figure 8b we insert two data-locks, one after each of the two groupings. During cyclic execution of reactions, when the control point hits a macro action node, CVM makes a call to the JVM with the required case-number. Next, the calling reaction suspends execution because of the data-lock and completes with a termination code of $\infty$. Sibling synchronous parallel reactions are executed in a similar manner. Once all the sibling reactions are either suspended (with a termination code of $\infty$) or completed (with termination code of 1 or 0), the current reaction is checked again to see if the data-lock condition has been satisfied. The signals checked in the data-lock test-nodes are emitted by the JVM once it has completed processing the requested case-number. The JVM also sends back the result of the completed case-number processing (called the res signal). The res signal is checked immediately after the data-lock test-node condition is satisfied, the control-flow of the CVM is affected by the outcome of this check. In Figure 8b the res signal is checked after the first data-lock condition is fulfilled. For the second macro action node, there is no check for the res signal, since the macro action node itself does not contain a Java test-conditional (i.e., the result of the case-number evaluation does not affect CVM control-flow).

5. THE TANDEM VIRTUAL MACHINE

In this section we present the TVM, a platform for executing SystemJ, which takes advantage of the reactive control-flow constructs to speed up execution time and reduces the memory requirements. This goal can be refined into a number of sub-goals for the new execution platform:

(1) To provide an environment directly supporting the semantics of SystemJ. This includes direct support for synchronous and asynchronous concurrency semantics.

(2) To be able to implement real-life models/systems. This includes efficient and complete support for the Java-data driven operations in conjunction with the control-flow processing, as well as efficient implementation of the communication between JVM and CVM.

(3) To implement the communication protocol between SystemJ and the environment based on the semantics (Section 3.2 and 4.3).

(4) To be modular and easily extensible. This is necessary for architecture exploration at a later stage (here architecture exploration refers to the various combinations of the virtual machines).

This section describes the important features of TVM which help achieve the above mentioned goals. Throughout the explanations we freely utilize the word “register” for special storage elements of the CVM, which are implemented as variables in the current CVM implementation.
5.1 Overview of the Tandem Virtual Machine

The TVM consists of a Java Virtual Machine (JVM) and a custom Control Virtual Machine (CVM) running in tandem. The JVM, is either a full standard Java virtual machine or a standard Java micro edition (J2ME) running on microprocessor platforms. The main components of the CVM are the program and data memories, the ALU, and the instruction sequencer. The program memory of the CVM holds the compiled and assembled SystemJ control constructs. In the control virtual machine, program memory is implemented as a linked-list with each node capable of holding one 32-bit assembler word. The ALU in the CVM is capable of only the most basic instructions like addition, subtraction, and certain custom instructions providing direct support for AGRC execution, as will be described in Section 5.3.

5.2 The CVM data-structures

The CVM is designed to efficiently implement the AGRC intermediate format. CVM has separate program and data memories. The data-memory contains the data-structures supporting the control constructs and the intermediate representation AGRC. The snapshot of the data-memory at any given EOT for a clock-domain gives the state of the currently executing clock-domain.

The data-memory (DM) for the CVM is arranged in a unique manner so as to allow for fast computations and easy compilation. The memory map for one or more clock domains is shown in Figure 9. The DM is considered to be 16-bits wide, the depth is not fixed and can be specified by the programmer or inferred from the number of clock-domains in the SystemJ program. By default each clock-domain is assigned 64-kbyte of data memory. In the DM first we have the input/output signals declared in the interface of the SystemJ example (Section 2). We use 1-bit for the status of the signal. Signal statuses are word (16-bit) aligned, thus if we have 16 or less signals we use at least one memory space (word) in the DM. The internal signals used within the SystemJ example and their locks (used for cyclic scheduling) are stored next. Like interface signals we use 1-bit word aligned placement for internal signal status. Internal signals can be emitted from multiple synchronous reactions in a given clock-domain and thus we assign 1-bit lock status for each signal per synchronous reaction. Figure 9 shows two lock positions for some arbitrary signal A. Thus there must be two synchronous reactions emitting signal A. Signals B and X have only one lock position since only one synchronous reaction emits them. SystemJ uses cyclic executive scheduling which requires these signal locks to be released by all emitting reactions before the presence check for the signal can be performed and execution can proceed further. Valued signals have their values stored in the standard JVM. Channels are also completely implemented in the standard JVM. The rendezvous algorithm in SystemJ is based on channel statuses and buffers [Malik et al. 2006b], which are implemented as Java variables and thus handled by the standard JVM.

Next we store the data-locks (Section 4.4). Data-locks inform us if the call made for an action computation (Java action node in AGRC) to the JVM has returned. A complete DM word (16-bits) is used for data-locks, unlike the signal locks as the number of data-locks is normally less compared to signal locks (equal to the number of synchronous parallel reactions in the clock-domain). This arrangement
Fig. 9. Data memory layout for the CVM

<table>
<thead>
<tr>
<th>Input signals 16-bits</th>
<th>1bit per signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output signals 16-bits</td>
<td>1bit per signal</td>
</tr>
<tr>
<td>Declared signals n-words</td>
<td>1bit per signal</td>
</tr>
<tr>
<td>Signal locks n-words</td>
<td>1bit per signal per reaction</td>
</tr>
<tr>
<td>Data locks n-words</td>
<td>1-word per reaction</td>
</tr>
<tr>
<td>PC n-words</td>
<td>1-word per reaction</td>
</tr>
<tr>
<td>Terminate codes n-words</td>
<td>4bits per reaction</td>
</tr>
<tr>
<td>Switch node 1</td>
<td>16-bits (1 word)</td>
</tr>
<tr>
<td>Switch child 1/1-word</td>
<td></td>
</tr>
<tr>
<td>Switch child n/1-word</td>
<td></td>
</tr>
<tr>
<td>Switch node N</td>
<td>16-bits (1 word)</td>
</tr>
<tr>
<td>Switch children</td>
<td></td>
</tr>
<tr>
<td>Join node 1</td>
<td>16-bits (1 word)</td>
</tr>
<tr>
<td>Join child 1/1-word</td>
<td></td>
</tr>
<tr>
<td>Join child 16th/1-word</td>
<td></td>
</tr>
<tr>
<td>Join node N</td>
<td>16-bits (1 word)</td>
</tr>
<tr>
<td>Join children</td>
<td></td>
</tr>
<tr>
<td>Repeat DM for CD2, CD3: CDn</td>
<td></td>
</tr>
<tr>
<td>Computation space</td>
<td></td>
</tr>
<tr>
<td>FIFO buffer size</td>
<td>end-address – no of PC</td>
</tr>
<tr>
<td>JVM call queues</td>
<td></td>
</tr>
</tbody>
</table>

also improves the runtime efficiency as pin pointing the memory location while releasing data-locks becomes easier. Next the program counters (PC) for the various synchronous reactions in a clock-domain are stored.

The termination nodes are stored next. Four bits are used for each termination node and up to four termination nodes can be stored in a single DM word. Figure 9 shows an example encoding of five synchronous reactions. In this case, first reaction’s (R1) termination code is stored in the least significant nibble of the first word. The termination code for the second reaction (R2) is stored in the second nibble and so on. The fifth reaction’s (R5) termination code is stored in the least significant nibble of the second word.

Table II. Selected special CVM instructions

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Register transfers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRESENT Rz Rx</td>
<td>if Rz(0) == 1 then PC ← PC+1 else PC ← Rx</td>
<td>If the given bit is high increment program counter, else the program counter jumps to the pointed position</td>
</tr>
<tr>
<td>SENDATA Rx</td>
<td>M[TP] ← Rx</td>
<td>Store the case-number and data lock pointer (from Rx) in queue. Lock the data-lock position</td>
</tr>
<tr>
<td>CHKEND Rz Rx</td>
<td>Rx ← MAX{Rx[15..12], Rx[11..8], Rx[7..4], Rx[3..0]}</td>
<td>Compare four-bit memory blocks from Rx, Rx and store result back into Rx</td>
</tr>
<tr>
<td>SWITCH Rz Rx</td>
<td>Rx ← M[Rz], Rx ← Rx + 1, PC ← M[Rz]</td>
<td>Switch to the child pointed to by the current switch node (Section 5.2)</td>
</tr>
<tr>
<td>SEOT</td>
<td>EOT ← 1</td>
<td>Set the EOT bit to 1</td>
</tr>
<tr>
<td>CLFEOt</td>
<td>EOT ← 0</td>
<td>Clear EOT bit</td>
</tr>
</tbody>
</table>

Switch nodes used for state selection of the currently executing SystemJ program are stored next. Switch nodes can have variable number of children/branches, each indicating the next state of the program (Figure 6). Switch nodes and their children are stored together. The switch node’s children are arranged in an ascending order: child indicating state zero is stored first and so on. Similar to switch nodes, we store the join nodes. But unlike the switch nodes where each switch can have variable number of children, each join node can have a maximum of sixteen possible children determined by the 4-bit termination code mapping ($2^4 = 16$). This arrangement of DM is repeated for each clock-domain. A single FIFO is used for interfacing between all the clock-domains and the shared JVM. Each synchronous parallel reaction, after making a call to the JVM, gets blocked (Section 4.4). Thus the maximum possible number of calls that can be queued at any given instant for a clock-domain is equal to the number of maximum synchronous parallel reactions composed using the $\|$ operator. A single FIFO is sufficient in our current execution model since the clock-domains are run in a round robin schedule, hence, a clock-domain instant (EOT) cannot be started without the completion of the previous clock-domain’s instant (EOT, Section 3.2, 4.3). Further research will be required to find the optimal FIFO configuration if clock-domains are run under different scheduling policies. FIFO starts from the ending address and the offset is dependent upon the largest number of synchronous parallel reactions in any given clock-domain.

5.3 Instruction set architecture (ISA) of the CVM

The CVM has fourteen instructions with immediate (instruction has an immediate operand value), inherent (instruction does not contain any operand or register), registered (instruction has one or more registers as operand), and indirect (instruction contains an operand as pointer into memory) addressing modes. Besides the usual load (LDR), store (STR), add (ADD), subtract (SUB), jump (JMP), and (AND), or (OR) and clear flag (CLF) instructions, we also utilize special instructions supporting SystemJ’s reactive constructs and intermediate representation directly. These include, present, switch, seot/clfeof, sendata and chkend. The register transfers for these instructions are shown in Table II.

—present The present instruction checks if the status of the signal is high. If
true, the execution continues from the next instruction, otherwise the program counter gets reset to the false branch. All presence and preemptive constructs (aborts/suspend) of SystemJ are mapped to this present instruction (test-nodes in AGRC representation). Through the memory mapping, the present instruction also provides direct support for the environment signal checks and thus reduces the code size.

—**sendata** The sendata instruction is used to make Java data calls to the JVM. Complete implementation details of the sendata instruction are explained in Section 5.4.

—**chkend** The chkend instruction is a special instruction which acts as the synchronizer for the \( \parallel \) operator. It is used to compare the termination codes of the synchronous concurrent reactions within a clock-domain and to make the decision regarding the continuation context. In the CVM implementation we have a four bit value which encodes the termination code for synchronous reactions. The chkend instruction takes as input a registered 16-bit value and compares the 4-bit nibbles to find the largest termination code. The current encoding scheme uses 0xF, 0x1 and 0x0 to represent the \( \infty \), pausing and termination of reactions respectively. Depending upon the number of synchronous parallel reactions within a clock-domain the chkend instruction can lead to large reduction in code size.

—**switch** The switch instruction is used to directly decode the switch nodes in AGRC. Like present, this instruction reduces the code size.

—**seot/clfeot** The seot/clfeot are special instructions used to implement the environment communication semantics of clock-domains and setting the boundaries of instants of time. The complete implementation details of communication between SystemJ programs and the environment is described later in Section 5.5.

Table III shows the SystemJ source code (a part of ‘UserNode’ reaction listing 1, lines 13-15), compiled into the equivalent Java and CVM assembler instructions for execution on the TVM. The compiled Java class (Table III(b)) contains a method CD0, which represents the ‘UserNode’ clock-domain. Method CD0, encapsulates a switch statement whose case-numbers represent the various Java instantaneous action-nodes (from the SystemJ source) including channel rendezvous. The CVM machine instructions are responsible for the complete control-flow of the SystemJ program. The CVM instructions (Table III(c)) call the JVM method (in this case CD0) with the required case-number whenever needed.

Table III shows only a part of the complete CVM assembler source code. Assuming that multiple clock-domains are synchronized, lines 1-3 make a call to the JVM. Line 1 fills in the most significant byte with the data-lock position. Line 2 fills in the least significant byte with the case-number to be sent to the JVM. SENDATA instruction fills in this word in the FIFO (Figure 9). Before fetching instructions from the program memory, the CVM checks for pending data calls in the FIFO. If CVM finds a non empty FIFO, it makes a data call to the JVM. When it has completed processing the data call, the JVM interrupts the CVM to return the result. A detailed explanation and algorithms for communication between the CVM and JVM are presented later in Section 5.4.
SystemJ Compilation using the Tandem Virtual Machine Approach

Table III. Example SystemJ compilation for the TVM

<table>
<thead>
<tr>
<th>SystemJ source code</th>
<th>Java source code</th>
<th>CVM assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ //reaction-1</td>
<td>public class UserNode{</td>
<td>1: LDR R0 #$0100 ;lock-position</td>
</tr>
<tr>
<td>{ //reaction-2</td>
<td>} public static void main(){</td>
<td>2: OR R0 RO #$0001 ;case-number</td>
</tr>
<tr>
<td>send TId(userid);</td>
<td>while(true){</td>
<td>3: SENDATA R0 ;data-call 1</td>
</tr>
<tr>
<td>}</td>
<td>/*polling on native C method</td>
<td>4:STR1 LDR R0 $1</td>
</tr>
<tr>
<td></td>
<td>to get the casenumber*/</td>
<td>;checking if data-call is</td>
</tr>
<tr>
<td></td>
<td>int casenum = poll();</td>
<td>;successful</td>
</tr>
<tr>
<td></td>
<td>boolean ret = CD0(casenum);</td>
<td>5: PRESENT R0 ELSE</td>
</tr>
<tr>
<td></td>
<td>/<em>sending result using C</em>/</td>
<td>6: JMP GE</td>
</tr>
<tr>
<td></td>
<td>send_res(ret);</td>
<td>7:ELSE LDR R1 $4</td>
</tr>
<tr>
<td></td>
<td>} public boolean CD0(int cnum){</td>
<td>;locking reaction-2</td>
</tr>
<tr>
<td></td>
<td>boolean ret = false;</td>
<td>8: OR R1 R1 #$00F0</td>
</tr>
<tr>
<td></td>
<td>switch(cnum){</td>
<td>9: STR R1 $4 ;reaction locked</td>
</tr>
<tr>
<td></td>
<td>case 1: TId.setValue(userid);</td>
<td>10: LDR R0 #$2</td>
</tr>
<tr>
<td></td>
<td>break;</td>
<td>11: STR R0 #STR1 ;PC stored</td>
</tr>
<tr>
<td></td>
<td>case 2:</td>
<td>12: JMP OEE ;jump to reaction-3</td>
</tr>
<tr>
<td></td>
<td>cmd = Cmd.getValue();</td>
<td>13:OE LDR R0 #$2</td>
</tr>
<tr>
<td></td>
<td>break;</td>
<td>14: STR R0 #OE</td>
</tr>
<tr>
<td></td>
<td>} return ret;</td>
<td>15:OEE LDR R0 $0 ;input signal check</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>16: PRESENT R0 ELSE2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17: JMP GE2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18:ELSE2 LDR R0 $4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19: OR R0 R0 #$0100 ;pause reaction-3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20: STR R0 $4 ;reaction paused</td>
</tr>
<tr>
<td></td>
<td></td>
<td>21: JMP GE3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>22:OE2 LDR R0 $4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23: OR R0 R0 #$0000 ;terminate reaction-3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24: STR R0 $4 ;reaction completed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;calculating termination codes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>25:OE3 LDR R0 #$3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>26: STR R0 #OE3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>27: CHKEND R0 R1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>28:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40: SENDATA R0 ;data-call 2</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lines 4-12 of Table III(c) check if the JVM has completed processing. If the PRESENT instruction (line 5, data-lock check) returns false, then the termination code for the currently executing thread is locked with value 0x00F0 (lines 7-9) in accordance with the cyclic scheduling algorithm (Section 4.4). 0xF encodes the value of $\infty$. Since this is the second reaction, the second nibble in the exit code position is filled with 0xF. Lines 10, 11, and 12 store the currently executing thread’s program counter and make a jump to the next synchronous parallel reaction/thread. If the PRESENT check (line 5) were true, a different program counter would be stored (line 13-14) and the termination code for this thread/reaction would instead be 0x0000. Lines 15-16, implement the input signal check (await) statement. All the input signals are present in the first memory location $0$. Depending upon the result of the PRESENT statement check, we either finish this reaction by placing a termination code of 0x0001 (pause) (lines 18-21) or a 0x0000 (termination code lines 22-24). Once both the threads have completed/suspended processing we call the CHKEND instruction to find the largest termination context. If PRESENT check (line 5) is unsuccessful, then CHKEND instruction reruns the first reaction (cyclic-scheduling). Once the data-call returns from JVM, PRESENCE check (line 5) is successful, reaction-2 is released from the data-lock, and the next data-call is made by the CVM (line 40).

5.4 Interface and communication between the JVM and the CVM

TVM executes control constructs by using the CVM and Java data-processing constructs using the JVM. In the current implementation, JVM and CVM are run as two operating system processes. Java Native Interface (JNI) is used for communication between the two. CVM starts the processing (since it is responsible for the control flow) and always leads the JVM. JVM on the other hand, once started, just polls for incoming data-requests from the CVM and responds to these by giving the computed result back to the CVM.

Figure 10 shows the communication interface between the CVM and the JVM. As stated before we use a switch-case statement wrapped in a method for representing the various action-nodes (instantaneous data computations) which need to
be processed by the JVM. An 8-bit value is used for communication and to represent the case number (Section 4.4). The instruction SENDATA making the data call provides the case number and a pointer into memory which holds the data lock position (example, Table III CVM assembler source lines 1-3). CVM uses the DPC, DPCR, and the IRQ registers to communicate with the JVM. The DPC register is the control register which is set high to indicate to the JVM that a data-call request has been made. The DPCR register at the same time contains the case number to be sent to the JVM. A currently executing data call is considered complete once the IRQ register is set high by the JVM. Figure 11 shows the algorithms implemented when a call to the JVM is made (SENDATA) and within the interrupt service sequences (ISR). These algorithms perform sequences of atomic micro-operations (register transfers). We utilize the Table III(c) lines 1-3, CVM source code lines to better explain the algorithms and the communication model.

—SENDATA algorithm: When sending data, the case number and the pointer to the data locks position is provided in the (16-bit) \( R_z \) register. The most significant byte (MSB) represents the pointer to the data lock position while the
least significant byte (LSB) represents the case number to be sent. In Table III(c) lines 1,2, register R0 if filled with the address of the data-lock position and then with the case-number to be sent respectively. During processing the sendata instruction the CVM carries out the following steps. First, the memory pointed to by the data lock position is cleared to ensure that the current reaction does not proceed further without the completion of the requested data call. For the current example, the memory at position 0x01 is cleared. We use two pointers into the FIFO buffer called the “tail pointer” (tptr) and the “head pointer” (hptr). Next the tptr is checked to see if it is pointing to the last place in data-memory: if so, tptr is reset to the start of the FIFO, else the value in the R0 register is transferred into the data-memory pointed to by the tptr and the tptr is incremented (effectively implementing a circular FIFO). Figure 10 shows the filled in tptr position with 0x0101 for the current example (here the least significant byte, 0x01 represents the case number which will be sent to the JVM, while the most significant 0x01 represents the data-lock position). Next a check is made to see if any previous data calls are still pending; if a previous data call is still pending then the control returns back, else the DPC register is set high before returning.

—ISR algorithm: The JVM and CVM execute asynchronously as two operating system processes, so CVM needs to manage the return from the JVM through an interrupt, as the time for the return result is unknown. In order to do this, the CVM performs a check on the DPC register before fetching the next instruction. If the DPC register is set, that means a pending data call; otherwise the next instruction is fetched from memory and the program proceeds further. When the DPC is set, the IRQ bit is checked. A set IRQ bit represents a return from the JVM of the current data-call. Thus we read this result from the JVM and transfer the value into the data-memory pointed to by the contents of hptr. For the current example, once the JVM completes processing the first case-number, it indicates this to the CVM by setting the IRQ register. The CVM sees the set IRQ bit and puts in the returned result from the JVM into data memory address 0x01, which is the data-lock position for this data-call. As shown in Figure 10, for the current example the result filled in is 0x01, where the least significant bit is the IRQ bit, while the second least significant bit is the returned result from the JVM. Next the hptr is checked to see if it has reached the last memory position and then either reset or incremented. Then the hptr and the tptr are compared to check if there are any pending calls in the FIFO. If they are equal, all data-calls are considered finished else the next data call is made to the JVM (notice that hptr is always lagging tptr when calls are pending due to the SENDATA algorithm). If the IRQ bit is not set, then the data call is made to the JVM and the procedure returns. For the current example this involves setting the DPC register on which JVM is polling, and setting the DPCR register with the 8-bit case number 0x01. The return from the ISR always sets the IRQ register to 0.
5.5 TVM interface for communication with the environment

Figure 12 shows the environment communication model of the TVM. In SystemJ, each asynchronous clock-domain communicates with the environment only once per instant, at the EOT (Section 4.3). At the EOT, all the environment input signals are read and the required signals including their values are emitted to the environment. In our current implementation, we use a round robin scheduling policy between clock-domains, and thus a clock-domain cannot start/restart execution before the current clock-domain’s logical instant (EOT) is over.

The SIP (Signal Input Port), SOP (Signal Output Port) and SVOP (Signal Value Output Port) are the registers which are used by the TVM to communicate with the environment. We also use the EOT and the EREADY (Environment Ready) registers (1-bit each) as control registers while communicating with the environment. The steps which the CVM undertakes for communicating with the environment are described below.

(1) Before starting execution, the CVM polls on the EREADY register. EREADY is set high by the environment to indicate that it has finished reading output signal values/statuses from the previous instant of time, and also finished writing the new set of input signal statuses/values.

(2) Once the EREADY signal is set, its value has to be reset (EREADY ← 0). The EOT signal is emitted (EOT ← 1). EOT is used by the CVM to communicate to the environment to stop reading/writing signals into the registers (SIP and SOP/SVOP) respectively. The environment is expected to instantaneously respond to this request and maintain the register integrity i.e., no read/write
while the EOT is high. We make two assumptions to validate the abstraction that, environment is willing and able to respond to EOT:

—We assume that the environment is capable of reading and writing the interface signals between clock-domains and itself simultaneously.
—We also assume that the designed system has been validated to adhere to the synchrony hypothesis. This means that each clock-domain is guaranteed to complete processing before its deadline. The deadline is determined by the inter arrival time of the input signals from the environment, a part of the specification provided by the designer.

The synchrony hypothesis guarantees that the clock-domain will complete processing before the arrival of the next input event from the environment. At the same time, the first assumption and (1) guarantee that EOT is not set high before the environment is ready with the next set of input events. This way we guarantee consistent interface between the CVM and the environment.

(3) The Next step is to read the input signal statuses from the SIP to data memory (DM[0] ← SIP).

(4) Next, the SOP is filled with the output signal statuses from the previous instant’s processing, they will be false (0) for the first instant of time (SOP ← DM[1]).

(5) Next, the SVOP register is updated to indicate the type of signals that are being emitted (either pure or valued signals): SVOP ← $R_x$. A high bit represents that the signal is a valued signal and a low bit represents that it is a pure signal. This information is required because the environment needs to know that the values it reads from the data-processor memory are valid.

(6) Next, the case number is sent to the data-processor which informs it to read the input signal values from the memory written to by the environment. The same case number is also used to emit the output signal values in a separate memory slot and to update the channel statuses for the current clock-domain.

(7) Finally, the EOT register is set to low (EOT ← 0) to indicate that environment can start reading and writing again.

These steps are carried out at the start of every clock-domain (EOT) implementing the semantics described in Section 4.3. A single set of registers can be used for environment communication because of the round robin scheduling algorithms. More research regarding the optimal environment communication protocol would be required if multiple clock-domains were to be run simultaneously, on separate virtual machines.

6. BENCHMARK RESULTS

We have carried out a number of experiments to compare the generated code size and the execution speed of the TVM (Figure 1c) versus the JVM (Figure 1b) implementations of SystemJ. The TVM is designed to implement both the GALS, and its subset the synchronous model of computation. TVM is also designed to provide access to complete data-driven computations (including Java method and class calls). The previously available reactive processors and virtual machines lack these features and hence were not considered for comparison purposes. We chose
both synchronous only and GALS examples with heavy data-driven computations to test the TVM.

Currently there is no benchmark suite available suitable to cover both the GALS and synchronous models with heavy data computations. Hence, we decided to develop our own. The synchronous examples include wrist-watch (ww) due to Berry, dace (a very small example), and a combinational lock (cc). These synchronous examples are borrowed from the Esterel test-bench suite [Edwards 2006]. For asynchronous examples we use the asynchronous protocol stack (aps) obtained from [Lavagno and Sentovich 1999], frequency relay (frelay) first introduced in [Gruian et al. 2006], a smart surveillance camera system (camera) which was developed by us and finally, a multi-player game of snake (snake) which is available on cell-phones. The aps example is purposely chosen to test the CVM-JVM communication interface as it has a lot of Java data-driven computations. The frelay example is a real life system capable of controlling switches in a power network. The smart camera system uses real cameras producing jpeg output to follow a designated object. Finally, the multi-player snake game is an enhanced version of the single player snake game available on cell phones. Our test-bench suite is available at [Malik 2008]. Table IV shows the lines of code (LOC) and the number of synchronous parallel reactions used in each example. This table is provided to give an idea of the complexity of the chosen benchmark.

All the experiments were carried out on a Pentium-4 3.2GHz desktop with 1GB RAM. Figure 13 shows the generated backend code size comparisons. The bars for the pure JVM implementation show the code-size of the generated class file after compiling with the javac compiler (version 1.5). The bars for the TVM implementation show the total code-size, which includes, the code-size of the class files for the data-driven processing, and the code-size of assembly code generated for control-driven processing. The generated code size gain over the pure JVM approach is easily visible. Control dominated examples like cc, ww, and dace require substantially smaller amount of program memory as compared to pure Java implementation, mainly due to the ISA used in the CVM. Examples with large amount of Java data processing (aps) show smaller code size gains. It should also be noted that with growing a number of synchronous parallel reactions, the CVM requires much smaller code-size compared to the pure Java implementation.

Table V shows the run-time comparison between the TVM and the pure Java implementation of SystemJ. A pseudo-random input sequence was fed into the examples for 1Million cycles (ticks) to obtain these results. The run-time gain
Fig. 13. Generated backend code size comparison between JVM and TVM

Table V. Run-time comparison, Pure Java Vs TVM

<table>
<thead>
<tr>
<th>Example</th>
<th>JVM ($\mu$sec/tick)</th>
<th>TVM ($\mu$sec/tick)</th>
<th>Ratio (TVM/JVM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc</td>
<td>27.69</td>
<td>2.59</td>
<td>0.09366</td>
</tr>
<tr>
<td>ww</td>
<td>283.2</td>
<td>32.7</td>
<td>0.11554</td>
</tr>
<tr>
<td>dace</td>
<td>5.15</td>
<td>1.76</td>
<td>0.3426</td>
</tr>
<tr>
<td>aps</td>
<td>4.2</td>
<td>15.5</td>
<td>3.6916</td>
</tr>
<tr>
<td>frelay</td>
<td>50.12</td>
<td>26.134</td>
<td>0.5214</td>
</tr>
<tr>
<td>camera</td>
<td>263.2</td>
<td>134.4</td>
<td>0.51</td>
</tr>
<tr>
<td>snake</td>
<td>97.56</td>
<td>42.63</td>
<td>0.436</td>
</tr>
</tbody>
</table>

The run-time for TVM varies from being ten times faster (cc example) to almost four times slower (aps example). The current CVM uses Java native interface (JNI) to communicate with the JVM for data-driven processing. JNI calls put a substantial overhead on run-time, so examples with a lot of such JNI calls (especially aps where calls are made to calculate each bit in a byte) are expected to have slower run-times. Even with the JNI overhead, in general, the TVM performs better than the pure Java implementation.

7. RELATED WORK

In this section we make a comparison between SystemJ and other currently available synchronous and asynchronous languages highlighting the main differences such as, model of computation, rendezvous semantics, compilation etc. We also provide a brief comparison between the TVM and other such reactive virtual machines and processors.

7.1 Comparison between SystemJ and other Models of computations

7.1.1 CRP/CRSM. CRSM [Ramesh 1998] and CRP [Berry et al. 1993] are the models which are closest to our model of computation. Like SystemJ, they use a...
CSP style rendezvous between Esterel [Berry 1993] modules. SystemJ differs in the data-computation capability. Unlike SystemJ, both CRP and CRSM have poor support for data-driven processing. SystemJ allows complex data-computations to be intertwined with control-flow, while CRP and CRSM separate the two programming paradigms, thereby making programming complex systems much harder. Also, unlike CRP, SystemJ’s rendezvous semantics maintain asynchronous autonomy thereby making it suitable for distributed systems.

7.1.2 Synchronous languages. Synchronous models like those based on Esterel are completely deterministic and highly attractive since they can be directly translated into circuit design. While these models are good for representing synchronous systems, they cannot represent multi-clock systems. SystemJ provides a GALS model of computation which encompasses a much broader class of systems. ECL [Lavagno and Sentovich 1999] is another synchronous language which combines Esterel with C to provide a specification environment for designing complex systems. ECL comes closest to our compilation approach for TVM (both separate control-flow and data-flow at the intermediate level although they are intertwined at the specification level). The main differences between SystemJ and ECL compilation are; SystemJ compiles the control-flow into CVM assembler instructions, whereas ECL compiles these control-flow constructs into Esterel, which are further compiled into C code by the Esterel compiler. The SystemJ compiler compiles the data-processing constructs into Java. The two are then implemented onto a processor-simulator (CVM) and JVM respectively. In ECL one can either choose to build a static C binary, or implement separate synchronous parallel reactions with the help of an underlying RTOS [Lavagno and Sentovich 1999]. In our opinion our approach is much more portable than the ECL compilation approach. Also the CVM is specifically built for faster execution and smaller memory footprint, two goals which are not targeted by the ECL compiler. Lastly, SystemJ compiler is able to compile GALS models, while ECL compiler has no support for this model of computation.

7.1.3 Comparison with SHIM. SHIM [Edwards and Tardieu 2006; Tardieu and Edwards 2006] is a fairly new language adopting a hierarchical asynchronous model for software hardware co-design. But unlike SystemJ, SHIM does not provide direct support for reactivity within the language itself. For example, SystemJ has constructs like `abort`, `await` etc, which allow the designer to directly specify reactive models. SHIM on the other hand emulates reactivity with C like constructs thereby loosing system-level abstraction. SystemJ also provides cleaner rendezvous syntax than SHIM. In SystemJ a designer can explicitly express rendezvous between synchronous islands (clock-domains) using `send` and `receive`. SHIM on the other hand binds rendezvous with C’s pass by value and reference paradigm, thereby making it much more obscure from the designers perspective. Lastly, SHIM also lacks the data-driven support provided by SystemJ for specifying complex systems.

7.1.4 Comparison with other reactive Java Packages. Other reactive Java packages like SugarCubes [Boussinot and Susini 1998] provides a deterministic synchronous model of computations based on the SL synchronous language [Boussinot and de Simone 1996]. Besides providing an easier development environment which
includes reactive constructs as first class primitives within Java, SystemJ also differs from SugarCubes in its support for the GALS model of computation. There are three main differences between SystemJ and SugarCubes semantics and scheduling policies. SugarCubes, like the SL programming language, relaxes the synchronous hypothesis, while signal emissions are reacted to in the same instant of time, reactions to signal absence are postponed to the next instant of time. SystemJ, like Esterel, reacts to signal presence and absence in the same instant of time. SugarCubes, being a reactive Java library/interpreter, allows synchronous parallel reactions to be added dynamically at runtime using the $||$ operator. SystemJ currently does not support this feature, all the synchronous parallel reactions are initialised only once at compile time. SystemJ’s approach, although less flexible, achieves the same program behaviour but with increased execution speed. In SystemJ logical clock-domains can be composed asynchronously a feature which is not supported by SugarCubes or any other reactive Java package [Hazard et al. 1999; Mandel and Pouzet 2005].

7.2 Comparison between TVM and other reactive architectures

The idea of introducing CVM is inspired by earlier works on reactive processors such as REFLIX [Salic et al. 2002], REMIC [Salic et al. 2005], STARPPro [Simon et al. 2008], KEP [Li and von Hanxleden 2005] and the Esterel virtual machine (EVM) [Plummer et al. 2006]. Reactive processors such as REFLIX, REMIC and KEP are designed to provide hardware support for reactive programs (all are inspired by purely synchronous language Esterel). While all these approaches provide efficient execution of reactive control parts and modest support for concurrency, they all provide very poor data-driven support and have practically no support for complex data operations.

Plummer et.al. [Plummer et al. 2006] have recently proposed an Esterel virtual machine (EVM) aimed at reduction of the generated code size. EVM utilizes custom byte-codes with variable number of operands to support the GRC format [Butucaru 2002] developed to compile Esterel. The EVM is also unable to support the host C function calls and provides data-driven computations similar to KEP. The main drawback of the EVM is the slow-down in the execution speed compared to the natively generated C code [Plummer et al. 2006]. Finally, none of these proposed approaches is capable of supporting asynchronous execution required in a GALS model and SystemJ.

8. CONCLUSIONS AND FUTURE WORK

SystemJ is a system-level programming language which extends Java to support globally asynchronous locally synchronous programs. Its first compilation strategies used Java virtual machine as the target execution platform. Our current compilation is based on transforming SystemJ program to its intermediate asynchronous graph code (AGRC) representation, and then its mapping on a JVM. In this paper, we have proposed a new approach for compilation and execution of SystemJ that uses advantage of AGRC representation and ability to separate all control (reactive and concurrent) driven processing from data (Java) driven processing. In this approach, two types of processing are mapped on two cooperating virtual machines, control virtual machine (CVM) and JVM, called tandem virtual machine (TVM).
The TVM provides significant reduction of actual total memory requirements to store compiled SystemJ programs, and in most cases much faster execution times. It also removes the need for an operating system. The major limitation of the current approach to achieve even higher gains in execution speed is the use of Java Native Interface as the mechanism through which the CVM and JVM communicate. This clearly leads to another task of making real control processor instead of CVM with further significant gain in execution speed. This approach will allow the use of tandem machine not only on a desktop, but also in embedded applications. Also, this approach allows architecture exploration, where multiple data processors (JVMs) and control processors are used as a target platform.

Another future research direction which we plan to explore is the scheduling scheme for clock-domains on a single CVM. The current round-robin scheduling of clock-domains has the advantage of being simple to implement and reducing resource consumption (single FIFO for communication between CVM and JVM). But, this scheduling policy might slow down the clock-domain execution time. The resultant average tick time (µsec/tick) for each clock-domain when scheduled in round-robin is the sum of the average tick time for all the clock-domains. For example, when two clock-domains designed with average tick times of 5 µsec/tick and 10 µsec/tick are scheduled in round-robin on a single CVM, the average tick time of both the clock-domains gets stretched to 15 µsec/tick, since, the clock-domains are run after another. In the resultant system the clock-domain might be unable to satisfy the synchrony hypothesis i.e., they might miss their deadlines.

For such scenarios a different kind of scheduling policy would be needed. A fixed priority preemptive scheduling policy like those in Real Time Operating System (RTOS) might be able to overcome such scenarios. Another, research direction would be distribution algorithms for clock-domains on multiple control virtual machines or processors.

REFERENCES


Avinash Malik et al.


