SystemJ Technology - Programming Without Borders

Zoran Salcic and Avinash Malik

SystemJ technology is a suite of powerful tools for seamless development of concurrent software systems for large a range of platforms, from high-performance embedded processors, via traditional single and multicore computers to complex distributed systems. It includes SystemJ Developer, a complete Eclipse based development environment, with a range of compilers for system-level language SystemJ and other tools for efficient development, validation and verification of designed systems. The compilers target the range of operating systems and execution platforms, from simple and embedded processors to multicore and distributed systems.

Introduction

SystemJ is a system-level programming language that makes easy; programming and composing software systems, which consist of multiple parts working together concurrently. The language enables design of software systems in a very intuitive way, first by partitioning the software on logical parts and then programming functionalities of each individual part using standard procedural language augmented by a group of new control statements, and finally integrating those parts into the overall system. This way it promotes both top-down and bottom-up software design with a number of other advantages, which will be clear after reading this paper. Concurrent entities of a SystemJ program can be grouped to be synchronous each with the other following strict synchronous model of computation, or asynchronous each with the other communicating through message passing. As such they are amenable for partitioning during execution on multiple processors (tightly or loosely connected) and can execute on a single very powerful or tiny processor, multicore platform or in a distributed environment. A SystemJ program fully complies with Globally Asynchronous Locally Synchronous (GALS) model of computation, thus offering some nice features automatically, which are otherwise usually difficult for most of the software designers, such as dealing with the details of synchronization and communication of concurrent behaviors and avoidance of unwanted effects (e.g. deadlocks and starvation) if not dealt with properly. For programming the sequential parts the language incorporates Java, thus enabling use of legacy code and through Java can include the code written in other languages. SystemJ is compiled to Java first and then processed by Java compilers. This way it maintains very high degree of portability on platforms which can execute Java code.

Motivations for a New Language

SystemJ was contemplated and then designed within a research project as an attempt to make a breakthrough in the current programming technology. All today's major programming languages have

reached a plateau in their power to deal with data types and objects, concurrency control and ability to communicate with their environments. The major characteristic of these languages is that they are designed for sequential programming, where the programmers write the code in the form of algorithms which execute in the order specified in the specification phase. Concurrency is present to some extent through the use of libraries, which in turn use the features of the operating systems. Concurrency is also present at the language level, such as Java itself, but in this case the onus of correct specification and implementation lies with the programmers and they need to be careful and responsible for their specifications. These languages have been designed for use on computing platforms of 1970s or 1980s, but not with today's platforms in mind. They have not been designed for execution on machines with multiple processors or applications, which are inherently running on multiple processors such as huge range of distributed and networked systems. Moreover, the appearance of wearable computers with the access to wireless networks has already made a big shift in the computing paradigm. Software technologies have evolved and developed to deal with new platforms, but they are fragmented and require use of complex interfaces and knowledge of various tools. They are not naturally integrated with a single programming paradigm which would enable easy and straightforward use when developing new applications and shift the onus of programming concurrent systems from the programmer to the language and compilation technology.

What are the typical new application/programming requirements that the current programming technologies lack? In order to illustrate this let us consider an example of a distributed system which is used for video surveillance in a large environment, such as an airport or any other public building, or even a segment of the city with few streets and intersections.

Smart Distributed Surveillance System – Motivating Example

We will describe the system operation through a simplified scenario. The system is illustrated with Figure 1. Sometimes hundreds or thousands of video cameras are used to capture information from the environment. That information has to be processed on different levels. First, image processing algorithms are used at the point of capture to extract some features from the input video stream (e.g. detect the moving objects, recognize and/or identify the objects and then track the objects, for which either static or dynamic cameras can be used). If the object tracking is the goal, it might be necessary to pass the information of the object from one camera to another; as the object moves and comes in the field of view of another camera, the object is handed over. Also, information on the object might be of interest to the system supervisors who monitor system operation. Those supervisors can be static or mobile. Finally, the input video streams have to be stored for archiving purposes and made available for retrieval in the future. Obviously such a system must be based on the use of distributed computing resources. Computers are present in cameras to capture information and enable basic network connectivity, but the processing power of such computers is typically limited for serious and real-time image processing. Therefore, one or more cameras must be connected to a more powerful computer (we call it a control box) to perform the image processing and to extract the necessary information. Control boxes may need to communicate with one another, to coordinate their activities and handover objects. Data storage of events in the system, and archiving of video streams obviously will be done on

some centralized resources. Also, the supervisor computers have to be networked, but they may have different power and connectivity. Even cellular phones can be used as a mobile supervisory computer. In order to make the example system more realistic, we may assume that other types of sensors are also used for surveillance system (infra-red, LEDs, RF) and also for control of the access, so the system has to enable fusion of the sensor information on its lowest hierarchical level. The designers of such complex systems would typically use standard languages to program individual functionalities, with inevitable use

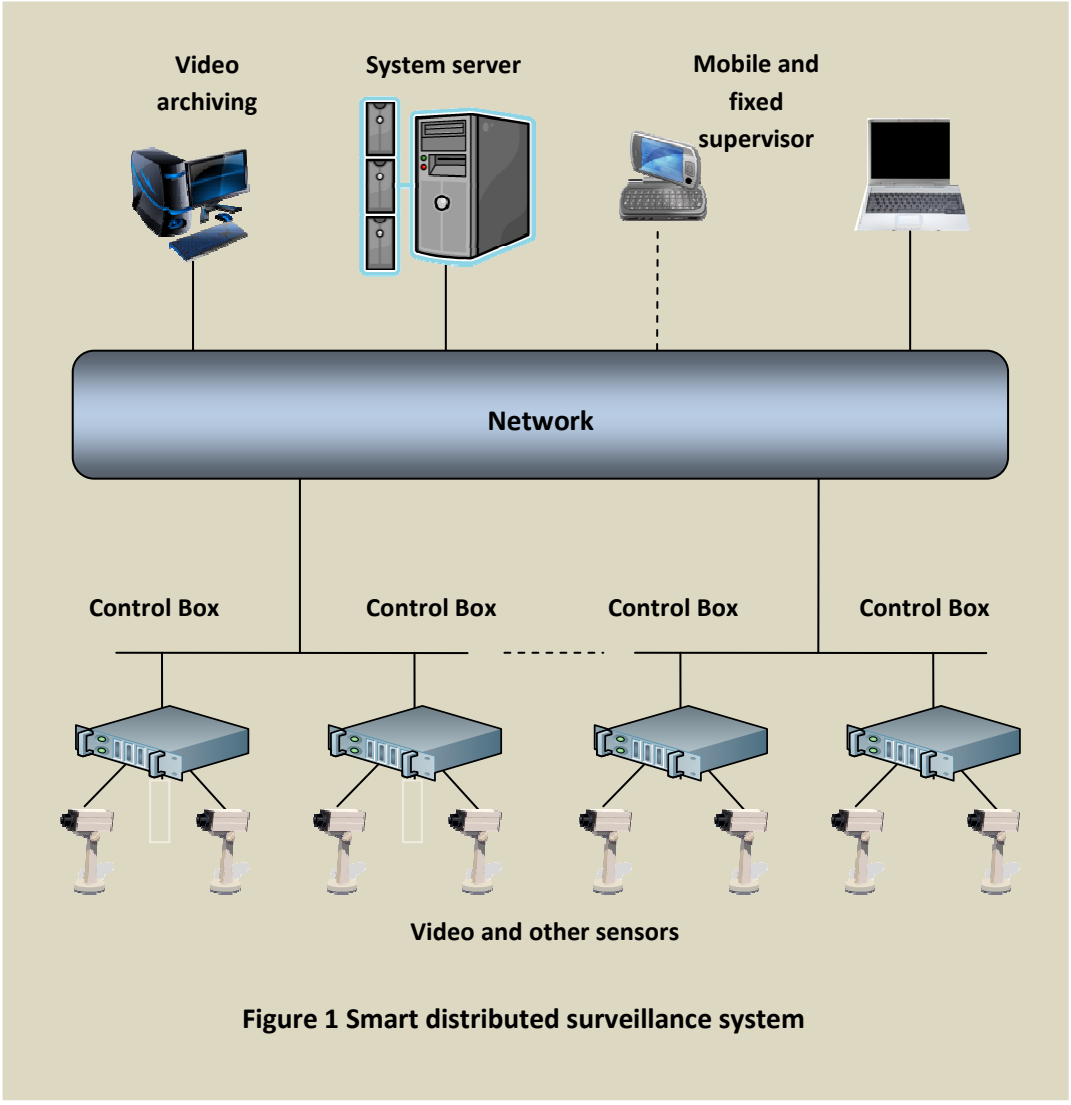

Figure 1 Smart distributed surveillance system

of middleware to enable communication of system parts. This type of approach has many drawbacks on all levels of hierarchy of such a system. As we will see, SystemJ enables abstraction of information, sensors fusion, communication between system parts and integral view of the system through a single specification on the language level in applications such as smart distributed surveillance.

Requirements on Design/Programming Language

What are the requirements of the smart surveillance systems that are very similar to almost any complex distributed system?

We will answer this question by giving a list of desirable features from the system design point of view:

1. **Hierarchical design**. Ability to decompose (partition) system into parts with the natural use of hierarchical relationship. This feature is a must for any system design because it enables splitting system functionality on several parts and then by concentrating on less complex parts and their composition into the bigger design units once the simple parts are designed. When designing software, hierarchy should be of both, structural and behavioral types. It is related to modularity, but modularity itself is not sufficient as those parts may depend each on the other in their operation.
2. **Concurrency**. Ability of parts of the system to operate at the same time (logical or real) and still be able to together make a meaningful system. Software concurrency is a well known concept, but it has been a mystery for most programmers due to complicated semantics and mechanisms to express and use it.
3. **Synchronization and communication.** Concurrent behaviors require to communicate, but also to ensure consistency of computation and data. This feature is obviously required because concurrent entities are not isolated each from the other. They exchange data and sometimes need to wait on information to be passed from another. These features are usually programmers' nightmare as small mistakes in traditional programs can cause fatal consequences during program execution.
4. **Reactivity**. Ability of the software system to react on the events happening in its environment (which may be physical environment or logical, i.e. other software systems running at the same time). It would be preferable that the system react on those events with different level of priority and even completely preempt if something very important event happens.
5. **Distribution**. This is a requirement for both the code and the execution platform, which would enable to run code with the same outcome on single processor computer, multicore computer or a networked platform.
6. **Sequential programming**. It is a feature of most of the currently used languages. It enables description of algorithms in a sequential form. Programmers should be able to use and deal with abstract data types, objects, control of program flow and other advanced features of contemporary programming practice.
7. **Verifiability.** Ability to verify some key properties of code without extensive tests, which is something what many programmers dream of. We are used to validate code through testing, but formal verification is difficult especially for languages which are not based on formal semantics and do not support any formal model of computation. Also, why programmers would not use well-established technique of test-bench used in hardware design?
8. **Code re-use**. It is an important feature, not so often done in programming as in the hardware design world. This also naturally extends to legacy code in which hundreds of billions of dollars have been invested.

We could bullet-point more desirable features of a language, but even those given here are sufficient to analyze the language we are using now and find out how much it satisfies those feature and matches requirements for the powerful software system design. The above features have been the main guide when designing SystemJ, which is our answer and offer to the software designers' world. We intentionally talk about software designers rather than programmers, because the design skills and how to engineer software systems are becoming very important, and they include, at the same time, very good programming skills.

SystemJ enables design of the software systems to be much more reliable and trustworthy, while at the same time shortening the design time and addressing one of the key issues in programming – programmers' and software designers' productivity.

SystemJ in a Nutshell

So, what is SystemJ? It is a language built on top of Java programming language, where Java can be considered its subset. The basic programming unit is a SystemJ program, or in the language called a ***system***, which consist of at least one sequential behavior. We will interchangeably use program and system although; the word ***system*** is a reserved keyword of the language itself. However, as SystemJ is a concurrent language the typical SystemJ program consists of multiple asynchronous concurrent entities on the top level of program specification. Those entities, called ***clock domains***, execute concurrently and asynchronously (each at its own ***logical speed***) and sometimes communicate each with the other using message passing. Communication mechanism in system is an object called a ***channel***.

SystemJ Features

- **Asynchronous and synchronous concurrency**
- **GALS model of computation**
- **Behavioral and structural hierarchy**
- **Built-in communication and synchronization**
- **Reactivity**
- **Distribution**
- **Determinism of synchronous parts**
- **Java for sequential programming**
- **Formal verifiability**
- **Code re-usability**

Each clock domain in SystemJ can contain a number of synchronized concurrent behaviors called ***reactions***. Reactions within a clock domain execute synchronously driven by an internal logical clock, which is in SystemJ called a ***tick***. In each tick each reaction makes one step of its execution and waits until all reactions within the clock domain have done the same. We say the reactions execute in lock-step. Each reaction can be further decomposed into a number of reactions, effectively creating a parent-child hierarchy, and the number of these levels is theoretically unlimited. The only thing to remember is that all reactions (parents and children) within a clock domain are ***synchronized***. Also, as reactions are concurrent, they need to communicate each to the other. The mechanism for communication between reactions within a clock domain is another object called a ***signal***. Signals have an interesting property. If

a reaction emits a signal, it is instantaneously, in the same tick, visible to all other reactions in a clock domain. We say, the signals are ***broadcasted***. If a reaction in a clock domain needs to communicate with a reaction in another clock domain, it uses ***channels***. A Channel, as opposed to a signal, is used for point-to-point communication. Reactions that communicate over a channel are called ***sending*** and ***receiving*** reactions. A reaction which does not have children is referred to as a ***simple reaction***. Simple reaction is nothing but what usually programmers consider a sequential program. In SystemJ a simple reaction is described using only sequential statements. There are two kinds of these statements: (1) new statements of the SystemJ language that belong to the flow control and communication mechanisms and (2) all statements of the Java language. If a programmer is not using SystemJ control constructs, then a simple reaction is effectively a Java program. Reactions which have children also use sequential statements, both from SystemJ and Java repertoire, but they also use parallel statements by which they can create and spawn their children.

A graphical illustration of a SystemJ program is given in Figure 2. Graphical representations are natural, because they reflect the structure (hierarchy and connections) between the parts of the program, which is actually a software system. The program in Figure 2 has three top level asynchronous behaviors, clock domains, called CD1, CD2 and CD3. Clock domain CD1 has two synchronous parallel reactions R11 and R12, clock domain CD2 has a single reaction R21 and clock domain CD3 has two high-level reactions R31 and R32, while R32 also has two children reactions, R321 and R322. The notation for reactions follows this parent-child relationship, which can also be presented as a hierarchical tree shown in the lower part of Figure 2.

Figure 2 also gives an informal graphical notation of the GALS model of computation that all SystemJ programs comply with. On the top level are clock domains and then they may be decomposed into further concurrent behaviors, reactions. We also see that clock domain CD1 communicates with clock domain CD2 using channel C12, clock domains CD1 and CD3 do not communicate directly at all, and clock domains CD2 and CD3 communicate via channels C23 and C32, respectively. Arrows show the direction of communication. Also, Figure 2 graphically shows, which reactions within clock domains communicate each with the other through signals. You have to notice that signals are emitted and broadcasted from a reaction and then become visible to all reactions. Arrows in Figure 2, showing communication between reactions via signals, actually indicate which reaction(s) use the emitted signals. For example, reaction R11 emits signal E, which is used in reaction R12. Conversely, reaction R12 emits signal F, which is used to control computation in reaction R11.

One more thing, which has not been mentioned previously, appears in Figure 2. We have graphically presented the ***environment*** to the SystemJ program with which program exchanges information and that way becomes a system as it gets inputs and outputs. SystemJ program allows communication with the environment only through special kind of signals, called input and output signals. Those are signals used by reactions, but only for communication with the environment. It is also important to note that environment is very abstract for SystemJ programs. It can be a physical environment with which a SystemJ program interacts thorough electronic signals, it can be a communication line through which messages are exchanged between the program and the environment or can be another program written in Java or any other programming language, which understands the signal abstraction.

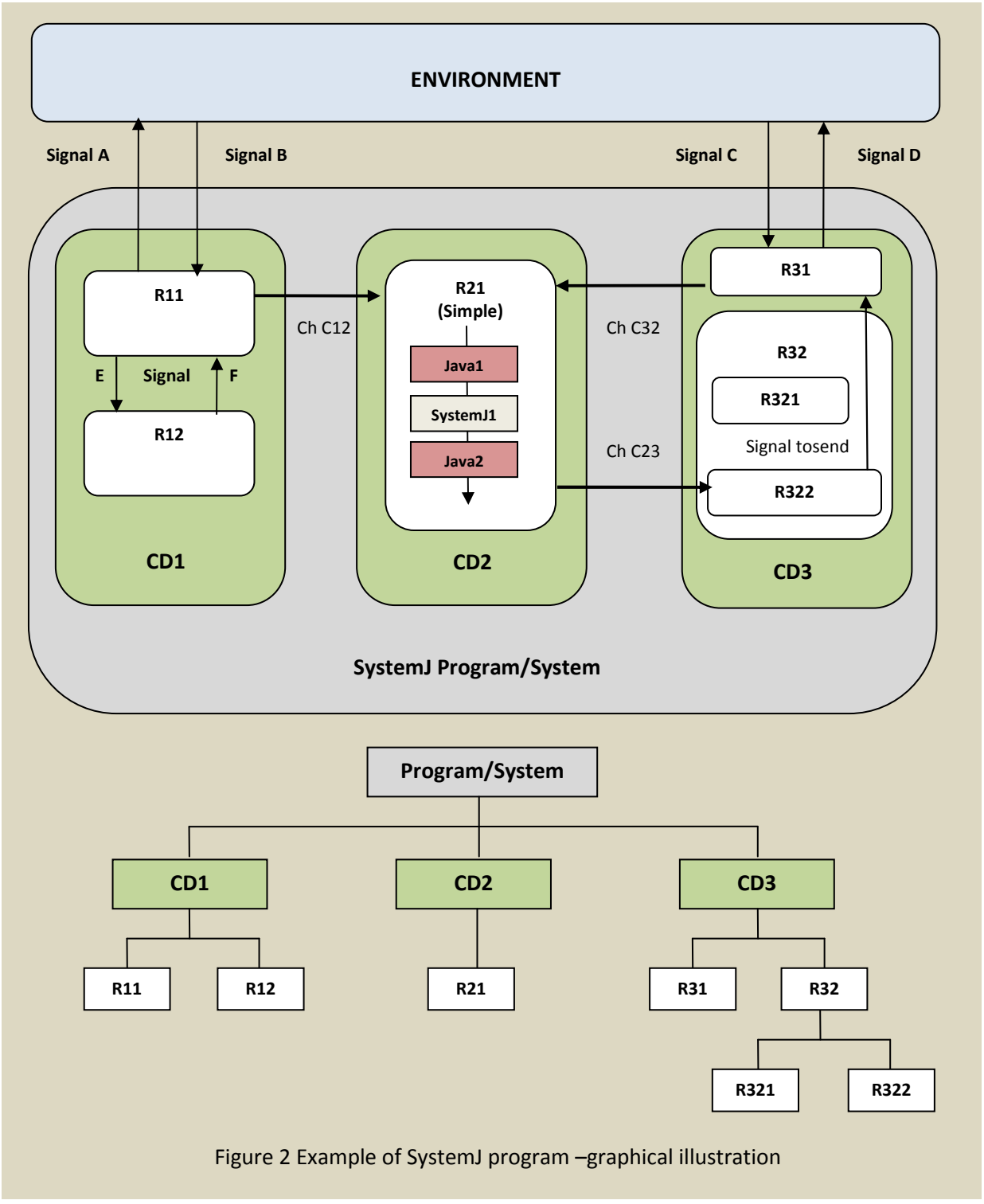

Figure 2 Example of SystemJ program –graphical illustration

Finally, Figure 2 also illustrates the place of Java code. Java statements are freely interleaved with SystemJ statements to make complex computations. They can also be wrapped up into complex control mechanisms through which reactions communicate with each other or the external environment. Java

code is used to implement usual programming techniques and abstractions of variables, data types and objects which are not provided by SystemJ itself.

So, what is SystemJ, then? It is the language that enables structuring of software system as one in Figure 2 and provides all run-time requirements to execute such a system correctly. For Java programmers this means a SystemJ program will be translated into Java code, which is then compiled by a Java compiler to provide the code that executes on the usual Java Virtual Machines. Once we know this, we can delve one step deeper into SystemJ and describe the major elements of its syntax and semantics. As SystemJ is a textual language with its own Java-like syntax, we will first introduce the basic features of the language. A more detailed presentation and treatment of the language itself with numerous examples that illustrate use of those features is given in another white paper called “The SystemJ Primer”.

SystemJ Entities and Objects

In order to make the transition to the notations and terminology of the SystemJ language easier, we present the concepts from the preceding section in a more formal way. Speaking more formally, SystemJ introduces three types of high-level entities, and two types of objects used within those entities on which new SystemJ control flow statements operate. These entities are:

1. **Program or system**. It is the top level entity through which a SystemJ program sees its environment and vice versa. Within the declaration part of this program entity interface signals with, which program communicates with the environment and channels through which clock domains will communicate are declared. Also, clock domains are also declared in the body of this entity. The SystemJ program will start by executing its body and starting clock domain execution in any order because clock domains are asynchronous.
2. **Clock domain**. Clock domain is the entity which groups all reactions that will execute synchronously. Clock domain itself is nothing but another reaction which may have children, grandchildren and so on, but is executed asynchronously with those reactions that are instantiated as clock domains.
3. **Reaction**. It is probably the most important entity because within reaction all computation and communication with its surrounding is defined. Reaction uses sequential and concurrent statements to specify its operation and those statements are from both SystemJ and Java repertoire. Reactions execute synchronously and fully comply with the perfect synchronous reactive execution paradigm. The difference between reactions and programs in imperative programming languages like Java and C/C++ is that reactions perform and use time. The time is in this case is explicitly introduced through new SystemJ statements, although it does not have absolute explicit value. It is logical time expressed in ticks. When any of the time consuming statements is executed by a reaction, it consumes one tick of its time. It stops its own execution and waits until other synchronous reactions in the same clock domain consume their own tick. All other statements, being SystemJ or Java statements, are executed instantaneously from the point of view of logical time (lock-step execution and the zero delay model).

SystemJ reactions use SystemJ flow control statements to operate on two types of SystemJ objects, which are used for communication and synchronization purposes:

1. **Signals**. Signal is an object that has special meaning. They are emitted by reactions and broadcasted to other reactions and the environment. Other reactions may use the signals to change the flow of their own computation. However, the life of signal validity is time limited. Once emitted, we say the signal is **present** only for one tick of time (in which it is emitted) and then it becomes **absent**. The third possible value is **unresolved** and we will leave the in-depth treatment of this topic to 'The system Primer'. Signal as an object consists of two elements: the status and the value. Signal values are abstractions and can be any object of any Java type (primitive or user defined). Signal status is used to indicate whether signal is present, absent or undecided. The value of the signal is the current value assigned to the Java object associated with the signal. Obviously, signal can be very simple and have just a status. In that case we talk of pure signals, which are similar to the usual signals used in digital electronics or description of hardware. As signals are used to exchange information with the environment, the environment itself has to be able to understand what to do with the signal emitted from a reaction and also be able to supply reactions with meaningful signals. Operations on signals are fully supported by the SystemJ compilation technology.
2. **Channels**. Channels are the object used to provide communication between reactions in different clock domains. SystemJ allows using channels to transfer a message of any Java type from a sending to a receiving reaction. However, as reactions in different clock domains are executing at different clock speeds, in order to provide safe transfer of the messages, synchronization between these communicating reactions is required. The synchronization is implemented by using rendezvous mechanism (full handshaking) and is enforced by SystemJ compilation of a SystemJ program. Thus, the system designer does not need to take care of any aspect of synchronization. The sending reaction is blocked until confirmation from the receiving reaction is received that the transfer was successful. This way channels become a safe way of ensuring that information is transferred or will be informed why the transfer was unsuccessful. As channels are very abstract objects, the underlying mechanism of actual data exchange (e.g. via shared memory, operating systems sockets and communication protocol like TCP/IP or operating system IPC) is not visible to the programmer. SystemJ provides many ways to connect the channels with different actual data exchange mechanisms. While those which use shared memory and execute on a single processor are automatically implemented, some of the others (like socket-based TCP/IP) are selectable options; the others can be added as needed for concrete target execution platforms.

SystemJ Kernel Statements and Syntax

SystemJ introduces rich set of statements for description of asynchronous and synchronous concurrency, communication and synchronization of concurrent entities, and reactivity. It is also able to interact with other languages and programming systems to enable interoperability with the code written in those languages or with legacy code. The language has certain number of kernel statements,

which are sufficient for the full use of the language. Some additional statements are derived from kernel statements in order to enable faster expression of designer's ideas and they are often referred to as syntactic sugar. The list of all kernel statements of SystemJ is given in Table 1.

Table 1 SystemJ kernel statements

Statement	Description and meaning
pause	Indicates the consumption of one time tick
[**output**][**input**][type]**signal** S	Signal declaration
emit S[(exp)]	Emitting a signal
p1;p2	Sequential statements
while(true) { p }	Infinite loop; must consume time
present (S) { p1 } **else** { p2}	Conditional statement
await (S)	Waiting on signal (polling); consumes time
[**weak**]**abort**([**immediate**]S){p}	Watchdog
trap (T) { p1..**exit**(T);..p2}	Software exception
p1 || p2	Synchronous parallel composition and || operator
output [type] **channel** C	Output channel declaration
input [type] **channel** C	Input channel declaration
p1 >< p2	Asynchronous parallel composition and >< operator
send C ([exp])	Sending on channel
receive C	Receiving on channel

It is important to know some of the conventions used in this table. For example, S symbol is used to denote signal identifier, and C symbol for channel identifier. Letter p is used to denote a single or a sequence of SystemJ statements. Finally, letter T is a symbol for the trap used in software exceptions. Curly brackets "{ "and " }" are used to enclose a segment of code, denote the beginning and the end of segment, respectively, on which specific SystemJ statement applies, but are also used indicate beginning

and end of reactions and clock domains. Square brackets "[" and "]" are used to indicate optional features in the statements. Simplest form of the statements is if there are no optional features. For example signal declaration statement can be used to declare input, output or internal (local) signals. Input and output signals are used for communication between reactions and the external environment, while local signals are used within a clock domain for broadcasting information to the reactions. Obviously, further they can be used for synchronization between reactions. The meaning of individual statements in here is described rather intuitively with further explanation in the next section where the statements are used in a concrete SystemJ program. The full meaning of the statements and the actual SystemJ semantics are described and can be found in other documents given in the literature section at the end.

SystemJ in Action – An Example Program

The purpose of the following example is to illustrate the SystemJ statements in action on the example of a simple program which follows our graphical example in Figure 2. The example shows the use of major SystemJ statements, but is slightly abstracted in order to allow us to focus on key elements; it should be considered as a backbone of a more complex program and part of a scenario that can be envisaged. Also, note that not all features shown in Figure 2 are used in this example code (signals A, C and D), but can be easily incorporated in a slightly different scenario, as well as some other features could be introduced as we will explain in the text.

The program is presented in Listing 1. So, let's start with the program explanation. It begins with system declaration (line 1), which is followed by the interface declaration (lines 2 to 8). Notice that all input and output signals into the system must be declared including their type, as well as all channels which will be used in the system. As channel as an object that has two ends (input and output), we have to declare both ends at this stage (line 6 and 7). What follows is the program/system body which is delimited with the curly brackets (lines 9 and 66). Within the program body we have three clock domains, CD1 (between lines 10 and 32), CD2 (between lines 34 and 41) and CD3 (between the lines 43 and 65). Clock domains are asynchronous each to the other and we indicate it by the asynchronous parallel operator "><" (lines 33 and 42).

In the implemented scenario, reaction R11 (lines 12 to 19) in clock domain CD1 waits on the presence of signal B that is of an integer type, extracts its value and after some calculation emits the result in local signal E and then waits on feedback from Reaction R12 (lines 21 to 31) that will be received on signal F. Reaction12 does some computation on the value of signal E, and eventually (not shown in the code) returns feedback in signal F. Notice that these two reactions operate concurrently and are connected by synchronous parallel operator "||" (line 20). If we go back to Reaction11, once it receives signal F it wraps it into a message which will be then sent to another clock domain, CD2. From figure we see that CD2 can receive messages from both CD1 and CD3, so it receives the messages from these clock domains (or more precisely from the reactions within these clock domains) concurrently. It is the reason why we created two child reactions (line 36) in CD2 to monitor concurrently for incoming messages from two different sources (on channels C12 and C32). Once Reacton21 receives a message on any of the channels, it calculates the result which is then sent to clock domain CD3 via channel C23 (line 40). Clock

domain CD3 has two top level reactions and their code is only partly populated. Reaction31 waits on the result of computation of Reaction32 (actually its child reaction R322) which is emitted in the local signal called *tosend* (shown in Figure 2) and once the signal is present, reaction R31 sends the message to CD3 via channel C32.

Listing 1 Example of SystemJ program that corresponds to the system in Figure 2

Line	Program
1	**system**{
2	**interface**{
3	**input** int **signal** B;
4	**input signal** C;
5	**output signal** A,D;
6	**input** int **channel** C12,C23,C32;
7	**output** int **channel** C12,C23,C32;
8	}
9	{
10	{//CD1
11	**signal** E,F; //local signals
12	{//R11
13	**await**(B); //waiting for input
14	int value = #B; //getting the value
15	//Computation on value
16	**emit** E(Math.abs(value)); //emitting the absolute value
17	**await** (F);
18	**send** C12(#F); //send the value on F to CD2
19	}
20	||
21	{//R12
22	int t=0;
23	**present**(E){
24	//computation
25	t = #E;
26	}
27	**else**{
28	t = 1;
29	}
30	//some computation on t
31	}
32	}
33	><
34	{//CD2
35	//R21
36	{**receive** C12;} || {**receive** C32;}
37	int e = Math.abs(#C12);
38	//Java computation on e

```
39                          Calculate.sine(e);
40                          send C23(e);
41                  }
42                  ><
43                  {//CD3
44                          int tosend;
45                          int signal tosend;
46                          {//R31
47                                  await (tosend);
48                                  send C32(#tosend);
49                          }
50                          ||
51                          {//R32
52                                  int random;
53                                  {//R321
54                                          while(true){
55                                                  random = Math.ceil(Random.rand());
56                                                  pause;
57                                          }
58                                  }
59                                  ||
60                                  {//R322
61                                          receive C23;
62                                          emit tosend(Calculate.combine(#C23,random));
63                                  }
64                          }
65                  }
66          }
67  }
```

While the emphasis in this example is on SystemJ features, Java programmers may have noticed a number of Java statements used in the program. First, it should be noticed that SystemJ statements only work on signals and channels, and Java does not understand what those objects are. Therefore, some translation in the exchange of information between two types of statements must take place. It is done through the # operator which allows us to extract the value of a signal (e.g. line 14) or channel (e.g. line 37) and assign it to the normal Java objects for processing using Java statements.

The connection and non-existence of strict boundaries between Java and SystemJ statements also illustrates very intimate linkages between the two languages, which are at the end the foundation of SystemJ power: ability to deal with complex control-type situations and data abstractions at the same time, which has been the ideal for many programmers and even languages.

SystemJ Compilation and Execution

Opposite to the majority of currently used programming languages, SystemJ has strong mathematical foundation and formally defined semantics. The key feature of the language is the GALS model of computation. SystemJ draws from two important existing concepts that have been applied into working programming languages with mixed success. First, it is globally asynchronous model of computation based on ideas from cooperating sequential processes (CSP) [Hoare] that enables safe system decomposition on parts which communicate each with the other using channels and rendezvous. Second, it is synchronous reactive model of computation which is used to allow decomposition of asynchronous entities into concurrent parts that are mutually synchronized by the model itself and do not need any programmers and systems designers involvement into the synchronization aspects. Third element which is incorporated into SystemJ is the Java programming language itself. By integrating Java and enabling programmers to use it freely within the language, SystemJ is giving new life to Java itself, but also benefiting from the huge number of programs and applications developed in Java. Finally, Java is used as the SystemJ common denominator because all SystemJ programs are first translated into Java and then taken by the Java compiler for further compilation. Hence, the only condition to obtain executable SystemJ code is to have Java run-time support, a Java Virtual Machine. Current implementations of SystemJ which are for public release all use the JVM as a target. The first versions of the compiler specifically target J2SE (standard edition) and are available for usual operating systems (Windows, MacOS, Linux and Unix). Versions of the compiler for J2ME and Dalvik JVMs are in coming, as well as some less standard versions of the JVM. The research has been done on alternative compilation strategies and targets, but it is out of scope of this document.

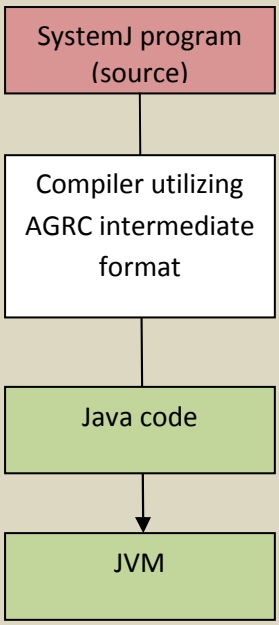

Figure 3 SystemJ compilation

A SystemJ program is first transformed into an abstract representation called the Asynchronous GRaph Code (AGRC), which enables clear separation of control (concurrency and reactivity) part from data transformations of Java and then can be targeted for different execution platforms. The current version targets JVM and produces optimized code for execution on a JVM as illustrated in Figure 3. One of the immediate questions can be, how SystemJ compiler deals with concurrency? and how is it implemented? First, dealing with synchronous concurrency is completely compiler's task. Each clock domain is transformed into very safe single thread of Java. All concurrency on the level of reaction within a clock domain is compiled away, resulting in a single thread with resolved mutual dependencies between signal emissions from different reactions. This strategy is adopted from compilation styles of synchronous languages and is a direct consequence of representing SystemJ program as an AGRC graph.

There are two strategies for compilation and execution of asynchronous entities, clock domains and the designer may choose one which better suits the needs of the application. The first one schedules all threads resulting from the clock domain compilation cyclically, thus effectively serializing the whole SystemJ program. While this strategy may result in slower execution, it has the advantage of essentially producing single threaded program, which is safe and even suitable for formal verification, as is each clock domain program. The second strategy transforms SystemJ clock domains into truly asynchronous Java threads which may execute as the threads or processes of the underlying operating system, depending on how Java threads are implemented. This type of implementation potentially may use as advantage execution on multiprocessor and multicore platforms. Also, compilation approach has another universality dimension in not requiring target system to be with the operating system. For resource constrained systems, or systems in which there is no big advantage of having an operating system, SystemJ can be tightly coupled with the targeted JVM and run without an operating system at all.

The tools provided for SystemJ program development extend capabilities of compilation and code generation to multiple SystemJ programs that have to work together and thus make bigger systems on centralized, tightly-coupled, or distributed computing platforms.

SystemJ programs co-exist with the other programs written in Java or other programming languages. The easiest way of interfacing such a program with a SystemJ program is to use signals as the interfacing mechanism. The idea is illustrated in Figure 4. All other programs are considered environment from SystemJ‘s point of view. So, interface of such programs has to be adapted to the requirements of SystemJ.

Figure 4 illustrates how a SystemJ program (system) communicates with its software and physical environment. Notice that each SystemJ program considers other SystemJ programs as its environment. So everything outside dashed lines surrounding SystemJ Program 1, is considered by Program 1 as its environment. As the primary communication abstraction with the environment is through signals, then other software components have to “understand” and be able to communicate through signals. Adaption of other software is sometimes necessary, as shown by yellow connectors in Figure 4. This adaption can be made by library support for the language in which software is developed (e.g. Java or C/C++). However, SystemJ itself provides implementation of signals and their mapping to some of the standard mechanisms found in common operating systems, such as sockets. In that case operating system performs adaptation, which is illustrated by dashed lines in Figure 4. Communication between SystemJ programs can be direct (if they operate on signals of the same type), or through the operating system if it is a complex type or it requires services of the operating system.

It should be noted that Figure 4 presents “software” as an abstraction running on some execution platform. That platform can be single processor system or anything else (multicore, multiprocessor or distributed computing system). Signals used to realize communication between SystemJ programs and their environment in that case will have different implementations.

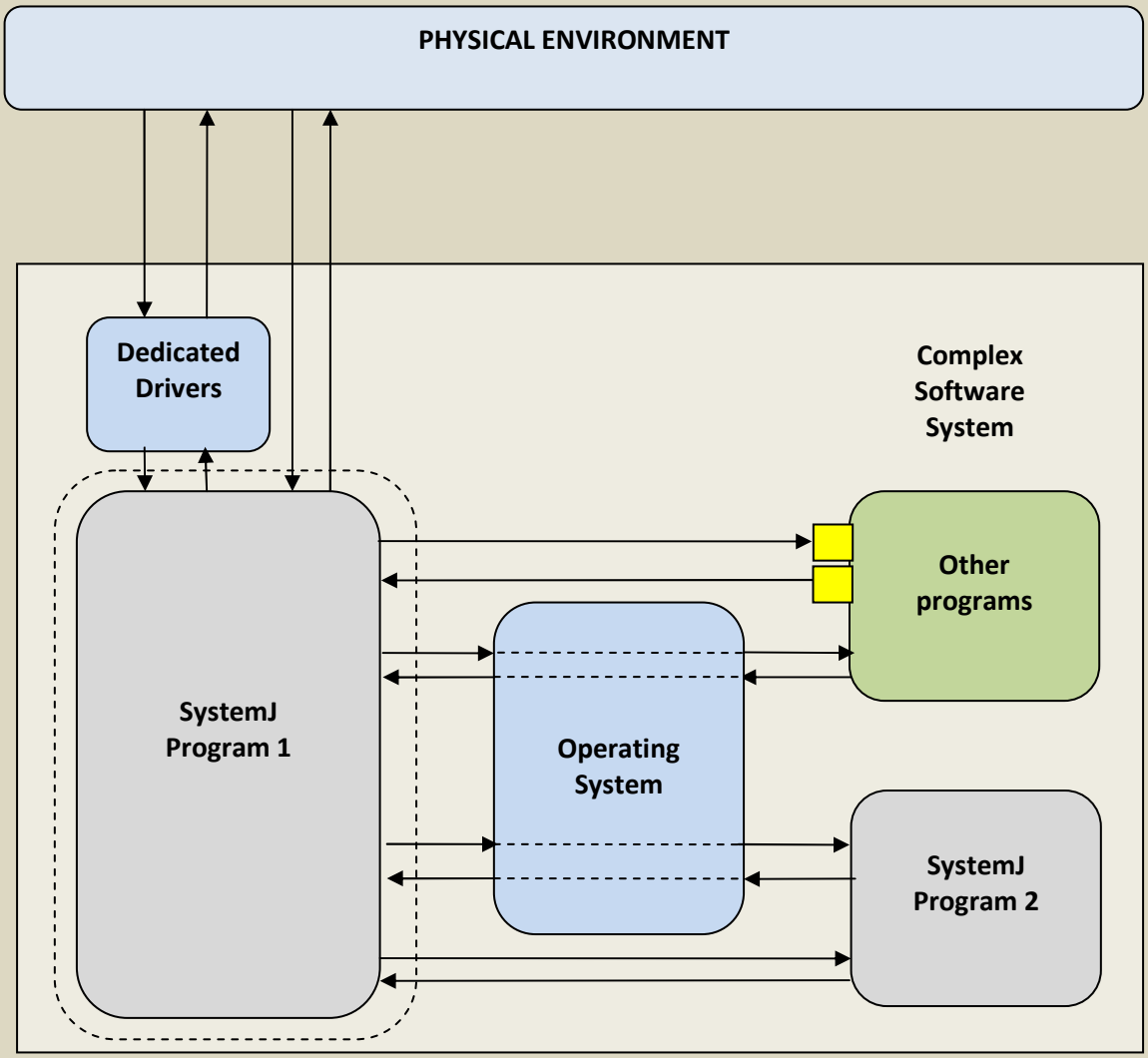

Figure 4 Relationship of SystemJ programs and their environment

It is also essential to indicate that multiple SystemJ programs (in Figure 4 just two) can communicate each to the other in two main ways and this fact is the basis for one of the tools in SystemJ development environment, called the Graphical System Composer, which enables one to integrate multiple SystemJ programs into a system by using a special GUI. The composer will be briefly described in the next section. SystemJ programs communicate directly if they are using simple signals with the values described with standard Java data types. However, the signals are very general abstraction and can have the type corresponding to any Java object. In order to enable such signals to be used for communication between SystemJ programs, or with other software, SystemJ provides universal method of emitting and checking for presence and receiving the values of signals through operating system sockets. Underlying mechanism to “send” and “receive” the status and the value of the signal must follow this universality and has to provide some form of serialization of data. Serialization can be based on standard Java serialization of objects or has to be “hard coded” into interfacing mechanisms. What should be noticed

here is that the SystemJ programs remain unchanged and therefore need not to be recompiled if the underlying mechanism for exchange of signal values is well defined and hidden from the programmers. It is an important feature used when designing Graphical System Composer, enabling very clear separation of computations in SystemJ programs from their communication.

SystemJ Test-Bench

As a concurrent programming language and with the ability to compose multiple SystemJ programs into a bigger system, SystemJ offers unique feature for development and debugging, i.e. validation of designed system by employing technique used by digital hardware designers in designing hardware systems. A test-bench consists of the original design (SystemJ program), which is considered as the design under test (DUT), and the entity which emulates the environment. The test-bench can be made in two ways depending on which SystemJ entity is used to emulate the environment:

1. Using a clock domain to emulate the environment by extending the SystemJ program\design with additional clock domain(s) that generate test vectors and observe the behavior of the design. This requires a minimal modification of the design itself (additional channels have to be declared and used to enable communication between environment emulation clock domains and original design). The easiest way is to add to the original design a reaction which uses channels to communicate with the environment emulation clock domain and signals to adapt the requirements for communication with external environment of the original design. This situation is illustrated in Figure 5 (a). The test-bench appears as new SystemJ program which can normally be compiled and executed by using the standard tool set.
2. Using SystemJ program to emulate environment. This approach is depicted in Figure 5 (b) and essentially uses the approach of composing bigger systems which consist of multiple SystemJ programs as described in previous section. The original design\program becomes the DUT, it communicates with the environment emulation system using signals. Depending on what are the types of the signals used for this communication, support from the operating system may be required. For complex types, typically, a universal mechanism is to use operating system sockets, which is a very nice feature not only for simulation, but also for real system implementations. The use of sockets is an immediate enabling factor for distribution of execution on multiple machines, if necessary. In some situation it can be desirable, especially if the environment emulation program is computationally demanding.

In both the above mentioned scenarios, environment emulator and observer can be made at different levels of details, from just providing sequences of events, which test basic reactivity and sequences of reaction of the designed system, to complex reactive programs which interact with the design. This just has an aim of indicating capabilities of the language and its underlying model of computation, which allow extending designed system with the model of its environment and thorough testing before actual deployment. Our aim is also to add capabilities for formal verification of parts of the language which comply fully with the GALS model of computation.

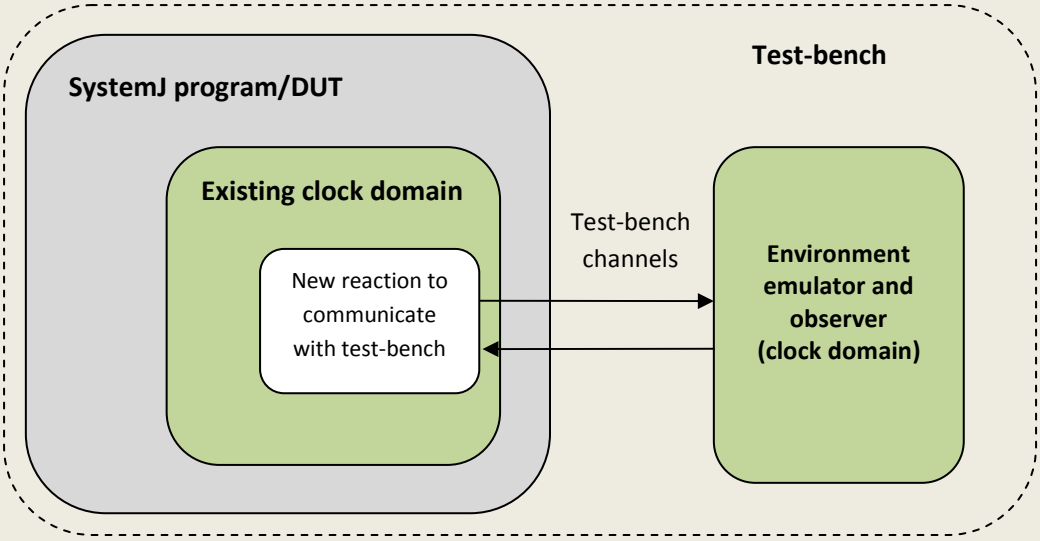

Figure 5 (a) Extending design with environment emulation clock domain

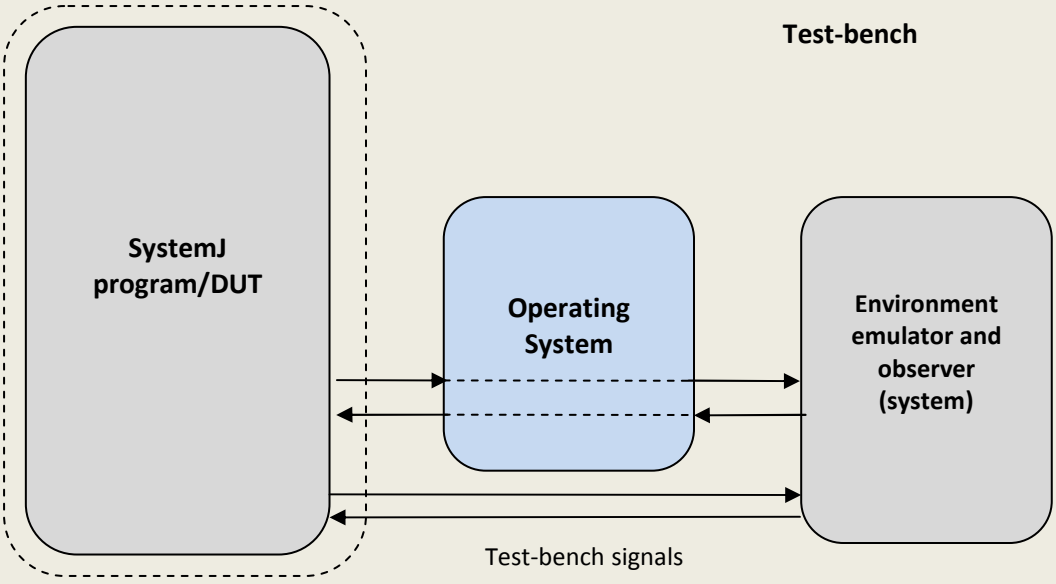

Figure 5 (b) Extending design with environment emulation system

Figure 5 Test-bench as validation mechanism in SystemJ

SystemJ Programs as Components – Building Complex and Distributed Software Systems

One of the key features provided by SystemJ technology is the ability to abstract the whole SystemJ program as a component and then use that component as a building block in bigger software systems. Moreover, the approach enables distribution of those components on different, possibly distributed (networked) computers, and to a big extent alleviates the problems software designers of big systems face. The core of the approach is already shown in preceding sections where we showed how SystemJ programs treat each other or how they treat other software components. The abstraction of communication via signals naturally extends to multiple SystemJ systems working concurrently. Those programs comply with GALS MoC, but the system which consists of multiple such programs, which communicate via signals. But, regardless of this fact, composition of GALS programs, which guarantee features for their local functionality, into non-GALS asynchronous systems makes sense. Because signals as an abstraction can be mapped on different kind of objects, we can assume, for example, that they are mapped on objects which are messages transferred using standard communication protocols. Such mapping is supported within the current SystemJ technology for TCP/IP and UDP protocols, and the support for some others is in development. Access to the protocols is provided on standard non-real-time operating systems through their services, e.g. sockets. In that case, SystemJ signals are mapped on the operations on sockets even though the system designer\programmer does not see it. By doing this, we enable any SystemJ program to communicate with another SystemJ program using signals and sockets and in that way introduce a universal communication method. Obviously, this method is extensible on non-SystemJ components, too, making SystemJ not only for application implementation but also powerful middleware when connecting other software components into bigger systems. Real communication between these components is now supported directly by the underlying operating system and standard communication protocols. This means that communicating components can reside on the same machine, on different cores and on distributed machines in a computer network. In the networked case, obviously, those components are on top of the local operating system. This further means that they do not depend on a single operating system.

The scenario in which concurrent software components are allocated to different computing platforms is illustrated in Figure 6. Allocation of components on the execution platforms is at the time being static and specified in a specially structured XML description, which can be automatically generated even in graphical form using Graphical System Composer (GSC), or can be made manually. This description is analyzed by a parsing program at SystemJ program start and real connections between signal abstractions and the used underlying mechanisms are established. In this way, computation within SystemJ program (component) is completely separated from communication issues, and SystemJ program does not need to be recompiled in order to be able to communicate with the other programs or software components.

This approach has been successfully demonstrated on multiple examples of multi-system programs, one of them being smart distributed surveillance system.

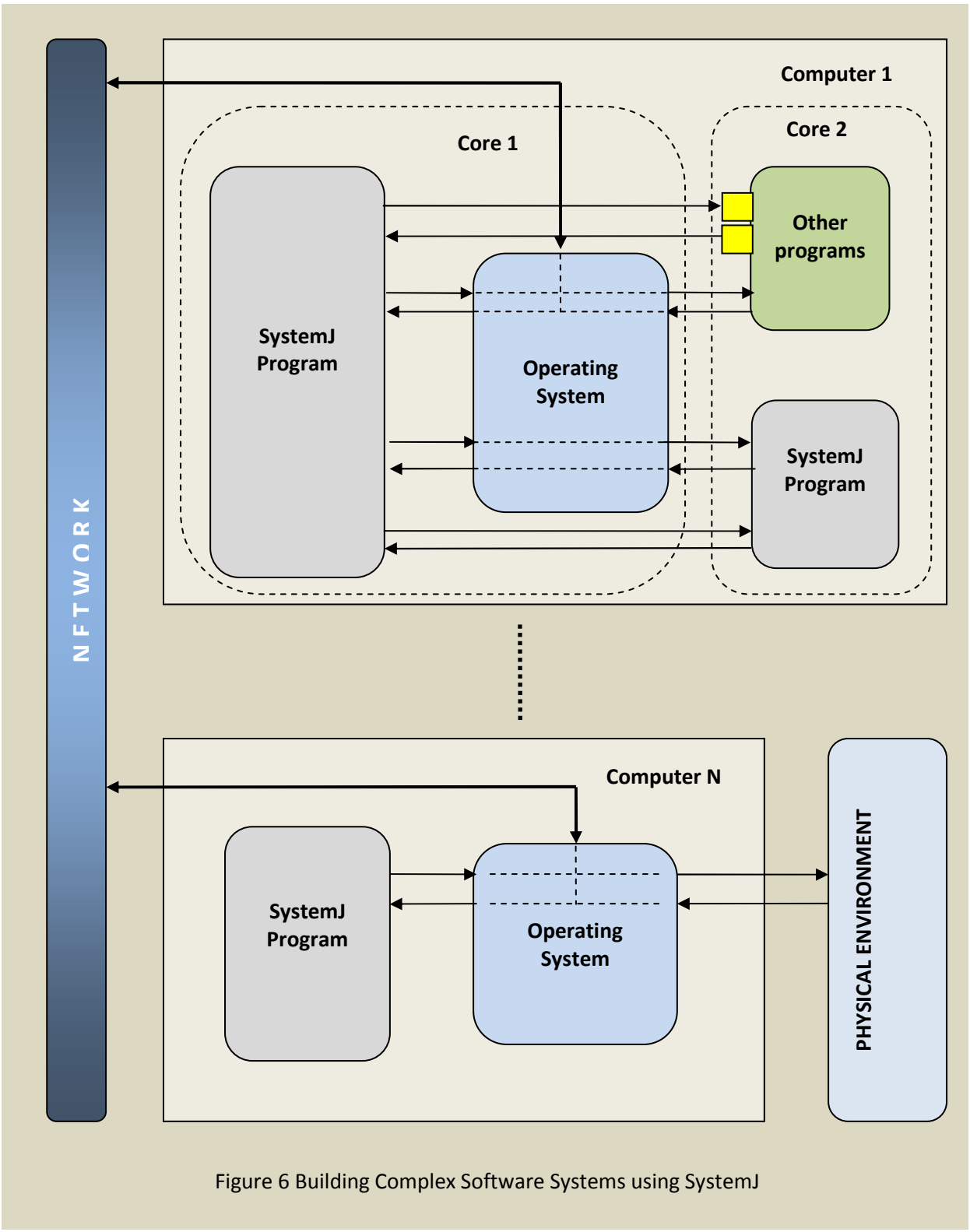

Figure 6 Building Complex Software Systems using SystemJ

The SystemJ Developer

SystemJ technology tools are from the very beginning based on Eclipse as a platform that enables users to run all the current and future tools, as well as the tools from different vendors, from the same familiar environment. Individual tools are implemented as Eclipse plug-ins and can easily be incorporated into already existing environment without the need for full reconfiguration. Tools are automatically up-dated through the up-date site on the Internet.

The core parts of SystemJ Developer are project management support, language editor, SystemJ compiler and SystemJ concurrent debugger. Also, a Graphical System Composer will be released soon. The user interface is organized through Eclipse views, editors and perspectives and easily accommodates future additions in a uniform way. It is made for speed of development as the major goal through the use of incremental project builder, templates, wizards, shortcuts and refactoring. Language editor with templates, which guarantee language completeness and tool tips. SystemJ concurrent debugger enables running programs in debug mode with features such as stop at breakpoints, advancing of clock domain execution to the end of the logical tick, watching signal and channel operation (values and statuses), modifying the statuses and values of signals, manual switching of execution from one to another clock domain, highlighting of executing reactions and other operations. Target for compiler code generation is selectable.

Functionally, the *project manager* offers several key features; the workspace, custom buildpath, shortcuts, delegates, templates, wizards, perspective and informative dialogs. The most important is the workspace as it displays all the aspects of the development that the programmer would need, such as the SystemJ code editor, the package explorer and the console view. To organize the workspace in this manner the user just needs to start and populate a SystemJ project using the supplied wizards and templates or simply select the SystemJPerspective. Once a project has been created and fleshed there are several courses of action. The user may choose to simply push the 'Run/debug SystemJ application' shortcut, which will run/debug the selected SystemJ program with a programmatically defined configuration (typically the most common one). Should the users wish to customize this configuration further, they simply need to alter the contents of the delegates under run/debug configurations, respectively. Lastly, the informative dialogs serve to keep the user informed of what has happened behind the scenes. Of note is the dialog which detects and offers the user the choice to recompile a SystemJ program depending on the last edit and previous compile time. Also, there is a dialog which helps with the assigning of xml files to the respective SystemJ program executions.

As with any concurrent system validation, there are features that make the process much less arduous and more informative. In the case of the SystemJ debugger these are: line breakpoints, suspend and resume asynchronous events, debug targets, source lookup, logical structure and run-to-line execution. The source lookup feature shows the current point of the execution and allows the user to place line breakpoints or run-to-line breakpoints. Depending on where execution has been halted, by breakpoint or suspend event, the logical structure view presents values of the objects in the correct hierarchical format. Finally, the debug targets can be changed from local to remote. In future this will be extended with a timing diagram view in addition to other features.

Two examples of SystemJ Developer workspace and perspectives are given in Figure 7 and 8, respectively. As shown in Figure 7, the development suite offers several useful tools and visual aids which assist in the development of SystemJ projects. Of particular note is the SystemJPerspective which organizes the layout into the most conducive manner for a SystemJ programmer. As well as this, the code editor features such items as textual highlighting that aids in making the structure of the system clearer. Lastly, the package explorer and console views present in a clear manner the contents of the project and the status of the current action.

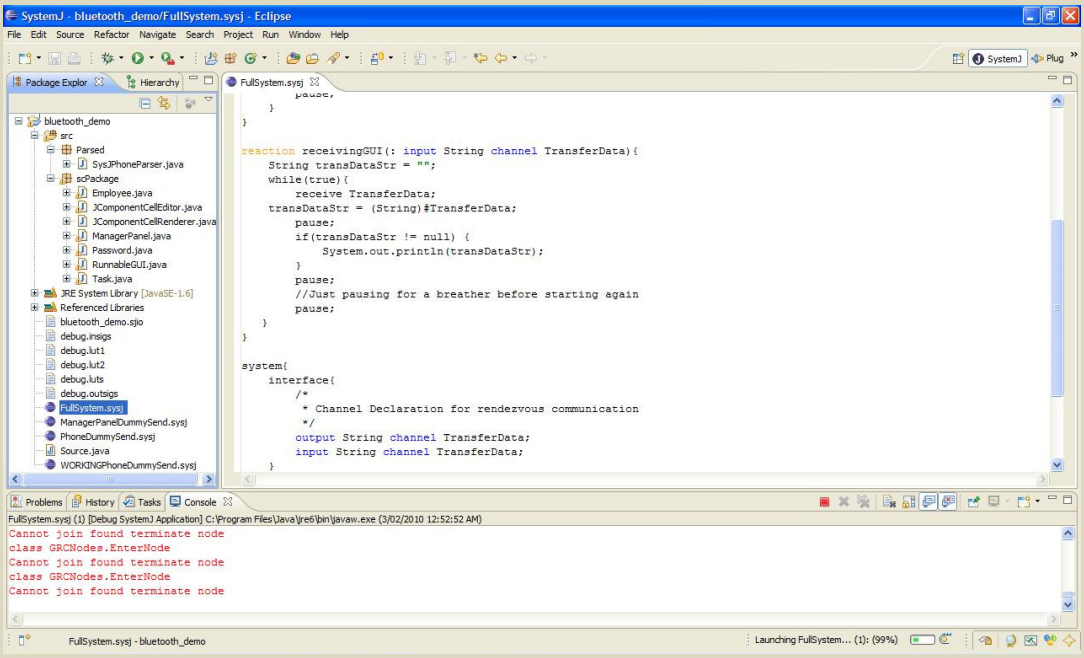

Figure 7 Eclipse SystemJ Developer – Compilation Perspective

As with any system, the functionality must be properly and fully validated. The debugger screenshot shown in Figure 8 presents the essential components that aids in this development procedure. The details of each particular thread of execution (reaction or clock domain) are shown in the debug tab view including actions such as suspend and resume. In the variables view tab the values of signals in individual clock domains are presented. This is complemented by the SystemJ source code view in which breakpoints, static or run-to-line, can be highlighted in with a blue line highlight to show current execution point in observed entity.

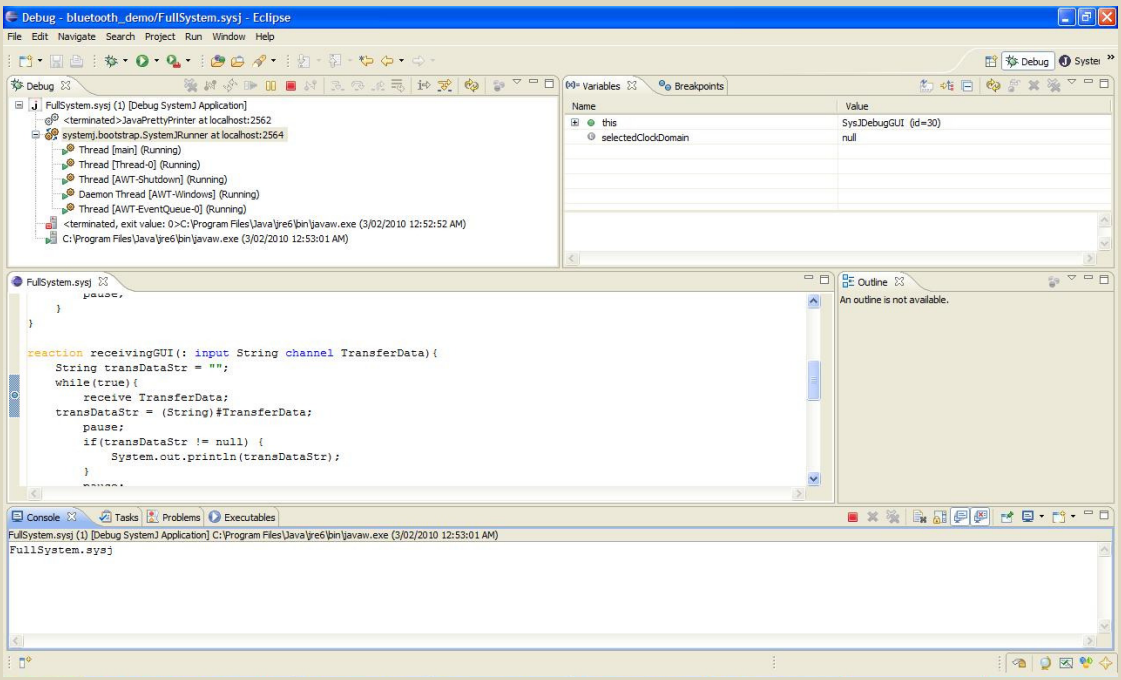

Figure 8 Eclipse SystemJ Developer – Debugger Perspective

Application Areas

SystemJ is a general-purpose system-level design and programming language. As such, it can be used for programming almost any software system. Especially by incorporating Java code into SystemJ, it enables inclusion of already existing Java code, as well as the huge number of existing libraries made by programmers for different purposes. Similar is valid for any programming language. However, there are still differences and strengths which make SystemJ particularly suitable for certain classes of applications where the traditional languages fail or are very difficult to use. One such example was previously presented smart distributed surveillance system, which requires not only a simple traditional language but much more in terms of middleware to be designed. We consider SystemJ particularly suitable with clear advantages in applications such as:

1. **Multi-participant collaborative applications**. These are systems in which multiple participants take part in an activity (such as writing a joint document, working on the same project, etc) on permanent or ad-hoc basis. Each participant activity can be supported/implemented by a SystemJ program and these programs, running on different participants' machines, constitute a system with multiple SystemJ programs, which communicate with each other via signals implemented through network communication.

2. **Distributed sensor networks with sensor fusion**. Smart distributed surveillance system demonstrated how SystemJ can be used to implement complex functionality. Sensor inputs (from video cameras, RF sensors, presence sensors etc.) can be abstracted by SystemJ signals and processed in a coordinated way. Also, communication between sensor nodes directly and with higher level of hierarchy in the sensor network is done through communication facilities of SystemJ). New sensors added to the system are supported by new concurrent entities in highly flexible and modular way.
3. **Multi-participant games on heterogeneous platforms**. This type of game is naturally supported by SystemJ. Its use is of primary benefit in implementing different common object and goals protected by SystemJ's synchronization and communication mechanisms, as well as in use of concurrent activities to implement different functionalities of each player in the game. Finally, as such games are distributed, communication through abstractions of signals and channels is of utmost value. Due to its portability and platform independence this may be one of the key advantages of the language.
4. **Light-weight middleware**. All above application areas are using communication as the main infrastructure for their implementation. SystemJ enables implementation of usual middleware functionality within the language mechanisms in very simple and efficient way removing a need for specialized middleware, which comes on the side of other programming languages. As such, it can be used in large corporate applications as the integrating element of existing applications, especially in systems that use Java in their implementation.
5. **Complex centralized and distributed simulators**. Simulators are usually implemented by decomposing functionality into application-specific simulation engine, event and input generators, observers and interpretation of simulation results and visualization. Even this level of decomposition clearly shows that those parts are naturally implemented by concurrent behaviors. Concurrency can be further found in each of them. Finally, each of them may run on a single processor computer, but also much more efficiently and altogether faster in some cases if allocated to different processors (multi-core systems) or even networks (distributed system).
6. **Security and surveillance systems**. We single out this type of systems as it combines functionalities typically found in sensor networks with the middleware, man-machine communication and server functions. As such it requires high-end computers even as its nodes (control box in our case) and complex communication schemes between individual cameras (or their control boxes) and higher level of hierarchy operator interfaces, servers.
7. **Medical and healthcare systems**. Modern medical and healthcare systems are naturally distributed with the high need for communication and collaboration. They face as the major challenge how to integrate parts together. For example, at the low level are the device which measure blood pressure, heartbeat, temperature and need to transfer information to the medical doctors who may respond and instruct the patient what to do. The doctors may be connected with emergency service to ask for transport of the patient. The measurement devices can be portable and connected to the patient while on the move and at work. They may have location technology (e.g. GPS) to help locate the patient if necessary. On the higher level medical records are usually distributed and held at different locations. They need to be merged in case of emergency. Also, the change in medical situation requires new medicines and

prescriptions, which are automatically forwarded to the pharmacy. The medicine can be further delivered by courier to the patient, delivery being triggered by the fact that it is ready. This type of scenario obviously requires concurrency, reactivity and connectivity on range of computer platforms, all being features of SystemJ.

8. **Smart power grid**. This is a huge area where generation and consumption of electrical power energy has to be controlled and optimized for all participants in the game. It can start on the low level of households which need to optimize consumption and reduce the waste of energy. It can be achieved by clever local automation systems connected into household system. Some households will be able to use their own energy generation to supplement the commercial one, and even contribute to the overall power system. Optimization is possible not only on the level of consumers, but also generators, which can chose which source will be activated at what time. All local and global optimization requires high level of connectivity and inherently has concurrency at all levels of system hierarchy.
9. **Intelligent environments**. Traditional view of home automation has its natural extension in intelligent environments, where humans communicate with automated system by using their sensing and motor capabilities. Humans can communicate with technical system using traditional ways of issuing commands (e.g. switches or keyboards on computers) , but also voice. They can be informed on status of the systems through visual means or by sound signals. Communication can be through computers, mobile devices or implanted devices in future. This type of system is highly connected, concurrent with the need for safe and secure communication, where SystemJ can provide key elements of integration.
10. **Robotics**. The requirements of robotics applications are very similar to intelligent environments. The difference is that some robots are mobile, but not necessarily all. They have sensors and actuators connected through a network of computers of different scale. Often they are put into situations that they need to react on certain events and situation from external environment. Even more, robots can communicate with humans or each to the other and collaborate in their operation, sometimes by referring to the central authority and sometimes just doing that on peer-to-peer level, which as an ideal scenario for SystemJ,.
11. **Truly distributed systems**. This is a generalization of features required by most of the above application-specific requirements. In simple words, SystemJ enables implementation and supports the execution of concurrent behaviors (synchronous or asynchronous) on individual computers in a distributed system and their communication in two major schemes: client-to-client and client-to-server. By enabling client-to-client (peer-to-peer) type of communication, it avoids major drawback of classical systems with servers as systems with single point of failure. It also improves system resilience on failures and their reliability.
12. **And many more**…. As it is obviously difficult and impossible to name all applications which will benefit of new way of computing enabled by SystemJ technology.

Literature

Here we present the list of published papers that describe SystemJ technology. However, it should be noted that the language is much more than just the papers. It is a creation into which long experience of many involved is actually built-in. Also, the SystemJ technology web-site will start publishing more practical documents and release software and demo examples, which will demonstrate how language can be used efficiently.

F. Gruian, P. Roop, Z. Salcic, I. Radojevic, SystemJ Approach to system-level Design, Proceedings of Methods and Models for Co-Design Conference, Memocode 2006, Napa Valley California, 2006, pp. 149-58. Piscataway, NJ, USA

A. Malik, Z. Salcic and P. S. Roop, Tandem Virtual Machine – An Efficient Execution Platform for GALS Language SystemJ, Asia-South Pacific Conference on Computer Architecture, 2008, p.1-8

A. Malik, Z. Salcic, A. Girault, A. Walker, S. L. Lee: A customizable multiprocessor for Globally Asynchronous Locally Synchronous execution, *7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, *ACM Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, *Madrid*, *23-25 September*, *2009*, p.120-129

A. Malik, Z. Salcic, P. Roop: SystemJ Compilation using the Tandem Virtual Machine Approach, *ACM Transactions on Design Automation of Electronic Systems*, *14*, *(3)*, p-, 2009

A. Malik, Z. Salcic, P. S. Roop and A. Girault. SystemJ: A GALS Language for System Level Design*, Journal of Computer Languages, Systems & Structures*, COMLAN, Elsevier, to appear, doi: 10.1016/j.cl.2010.01.001, 2010

SystemJ Summary

- **Asynchronous and synchronous concurrency mixed in GALS model of computation**
- **No limitation on number of concurrent behaviors**
- **Synchronization and communication resolved by the compiler**
- **System designer/programmer focuses on system functionality**
- **Highly portable – compiles to Java and runs on practically any JVM**
- **Uses Java for sequential programming**
- **Introduces safe mechanisms for reactivity and preemption on the language level**
- **Uses standard objects to communicate with environment including networking**
- **Runs on standard operating system, but can run on JVM only for specific applications**
- **Advanced Eclipse-based development environment (project management, language editor)**
- **Advanced Concurrent debugger**
- **Supports test-benches for efficient program testing**
- **Supports composition of multiple programs into bigger software systems**
- **Seamless transition from centralized to distributed platforms**
- **Enabled on embedded platforms which support any kind of JVM**

Requirements

- Eclipse Platform and Eclipse Java Development Tools Versions 3.5 and above
- Java Development Kit 1.6.0_13 and above

Development hosting

- Windows (XP, Vista, 7)
- Linux
- MacOS

For more information contact:

Auckland Uniservices Ltd. HTTP: www.uniservices.co.nz

HTTP: www.systemjtechnology.com

EMAIL: info@systemjtechnology.com