Abstract

In this paper, we propose a new system-level design language, called SystemJ. It extends Java with synchronous reactive features present in Esterel and asynchronous constructs suitable for modelling Globally Asynchronous Locally Synchronous systems. The strength of SystemJ comes from its ability to offer the data processing and encapsulation elegance of Java, Esterel-like reactivity and synchrony, and the asynchronous de-coupling of CSP all within the Java framework. Using standard Java environments, for specification and modelling, or specialised reactive embedded processors, for high performance implementation, the SystemJ design flow is extremely versatile. With the increasing attention that Java gets in embedded systems, SystemJ comes to address data and control, software and hardware, modelling and implementation in a unified manner.

1 Introduction

The design automation community is well versed with the concepts of different levels of abstractions of a given design. The well known and well understood levels are the physical level, the circuit level, the register transfer level (RTL) and the behavioural level. The physical level represents the lowest level of abstraction, while the behavioural level represents the highest level in this hierarchy. The higher levels of abstractions have been introduced to bridge the productivity gap. Although any design that is modelled at a higher level can also be modelled at a lower level, the effort and level of detail at the higher level would be substantially less than at any lower level. Normally, a human designer is needed to model a given design at the highest available level of abstraction. Subsequently, a compiler translates this model into lower levels. Each level is distinguished from the others by its own execution semantics, often referred to as a model of computation (MoC) [9]. For example, the MoC for the circuit level may be thought of as boolean equations to describe the data-path and finite state machines (FSMs) to describe the control. The MoC of the RTL may be a combination of FSMs with data-path [12] or as a microprogram. The semantics of the behavioural level has often been that of discrete event systems [9].

With the advent of ubiquitous computing systems, namely embedded systems, design automation community has been looking at an exponential increase in behavioural complexity and heterogeneity in digital systems. A system typically consists of a number of concurrent behaviours which process data and communicate with each other and with the external environment in various ways. A higher level of abstraction, the system level, is required to tackle these new design challenges. The major criteria distinguishing the system level from the lower levels of abstractions:

1. Separation of computation from communication: An embedded system is highly concurrent and often designed as a system on a chip that integrates several interacting and reusable components. This further requires that the specification clearly distinguishes communication from behaviour.
2. Mix of a range semantics or models of computation: As an embedded system is often heterogeneous, there is a need for using different MoCs suitable for the different types of computation involved.
3. Behavioural hierarchy: Differently from structural hierarchy, behavioural hierarchy allows for hierarchical levels in a specification where the various levels may use different MoCs. Behavioural hierarchy may be used to capture pre-emption or as a mechanism for refinement (from higher-level behaviour to a lower level and more detailed one).
4. Support for exceptions and exception handling: At system level, the designer should be provided with a smooth and explicit way to specify and handle exceptions, as in modern programming languages.
5. Mix of data-dominated and control-dominated processing: The system-level modelling paradigm must
allow for effective modelling of both control oriented and data (transformational) processing systems.

6. Support for formal verification: Since many embedded systems are inherently complex, heterogeneous, and often safety critical, there is a need for formal verification using a formal semantics.

The goal of this paper is twofold. We use the above six criteria to briefly compare existing languages for system-level modelling. We then use this comparison as a vehicle for our new system-level language, called SystemJ, proposed in this paper. SystemJ is a language that attempts to address all the above criteria within a single language framework and yet does not represent a fully new language. It is based on the already existing Java from which it draws many useful features. The most prominent one is that it can be executed in a standard Java environment, still giving many options to target other implementation technologies. The length of the paper allows us only to introduce the SystemJ language in a semiformal manner, along with a small case study used to illustrate its features by example.

The remainder of the paper is organised as follows. Section 2 takes a critical look at several of the system-level languages already available and introduces SystemJ. Section 3 details SystemJ language syntax and semantics, while section 4 describes the system design flow with SystemJ, along with the implementation choices of the current SystemJ environment. In section 5, we evaluate the advantages of SystemJ on real life applications. Finally, section 6 draws our conclusions and planned SystemJ developments.

2 Related Work

To emphasise the status of the current system-level languages and motivate the introduction of our new language, this section briefly reviews and compares some of the well known approaches. In addition, we also mention the Java based approaches most closely related to SystemJ.

2.1 Existing System-Level Languages

Figure 1 provides a classification of some well known languages which may be considered to be at a higher level of abstraction than hardware description languages. Languages such as SpecC [12] and SystemC [13] have been classified by us as system-level languages which have no formal semantics. While SpecC was initially developed as a C-like language with explicit features that separate communication and computation, SystemC is a C++ class library being developed both for hardware and software within a unified model. Some of the languages of interest that have a formal semantics may be further classified as either synchronous (Esterel [6, 28], Lustre [16], Signal [15], Argos [22], Statecharts [18] and ECL [21]), asynchronous (CCS [23], CSP [20], CFM networks in POLIS [4] and SHIM [10]) or globally asynchronous and locally synchronous or GALS (CRSM [29], CRP [7]) depending on the type of concurrent composition employed. Table 1 provides a comparison of a few of the well known languages based on the six criteria for presented in the previous section. In Table 1, “+++” denotes excellent, “+” very good, “+” good, and “-” minimal support for a given feature, respectively. It is fair to say that formal languages such as Esterel far exceed the informal languages like SystemC when compared on the above system-level modelling criteria. SpecC, while being much better than SystemC in terms of these criteria, lacks in two important aspects namely, support for multi-MoC and the support for formal verification. While Esterel excels in most aspects, it only supports a single model of computation called synchronous reactive or SR [9]. Another major limitation of Esterel is that it is a hierarchical language and relies on C for any complex data-path operations. While ECL overcomes this limitation of Esterel by modelling re-active constructs as well as data transformations all within a C-like language, it only supports the SR model of computation. SHIM is a more recent language that is specifically aimed at systems consisting of asynchronous, concurrently-running sequential processes that communicate exclusively with rendezvous through point-to-point channels. The language, uses a C-like syntax to specify both hardware and software parts of a system and the compiler synthesises RT-level VHDL from the parts marked as hardware and C for the software. CRSM is another recent language, that is based on an Argos-like [22] graphical syntax. It nicely incorporates most of the system-level language requirements and also supports both the synchronous semantics (like SR) and asynchronous composition (like CSP). Hence, it seems like an ideal graphical framework for visual modelling of embedded systems. The major limitation of CRSM, in our opinion, is the support for data-path operations. While CRSM constructs may be effectively employed to model control, data operations are handled by function calls as actions on transitions. This approach provides only limited data-path operations, suffering from the same limitation as Esterel due to hierarchical separation of data and control.

2.2 The Birth of SystemJ

Inspired by the fact that reactive languages provide a very nice support for system-level modelling, we designed SystemJ to combine Esterel-like reactivity, CSP-like asynchronous composition and an existing object-oriented language, Java. The basic idea of using Java to write reactive programs is not new. PureSR [25] is based on a Java library, with a specific synchronous semantics, different from Esterel. Java-Time [34] is a set of tools for successive formal
refinement, supporting an abstract reactive model based on Java. Jester [3] revolves around a translator to Esterel for the control path and Java for data processing. Junior [19] is based on a Java-written Esterel virtual machine, executing Esterel-like code. In contrast, SystemJ is intended to be more than just a reactive extension of Java. We envision SystemJ as an environment, ideal for specifying, modelling, implementing, and even formally verifying a large majority of reactive and real-time systems, or embedded systems, from the category of globally asynchronous locally synchronous systems.

The approach of extending an object-oriented language for system-level modelling has been successful for SystemC. However, unlike SystemC, SystemJ is carefully designed keeping the six objectives of system-level modelling in mind. SystemJ nicely separates computation and communication by adopting either pure synchronous composition of a set of reactive threads (called reactions in SystemJ) or an asynchronous, CSP-like composition of a set of reactions (constituting a system). Thus, in its initial version, SystemJ provides two different MoCs, SR and CSP. It provides very good support for behavioural hierarchy through reactive constructs similar to Esterel. It also has powerful support for exception handling through Esterel-like aborts and traps. Finally, SystemJ programs may be mapped to CRSM or DFCharts [26], to obtain its formal semantics and aid in its formal verification. The last row in Table 1 describes SystemJ using the previously introduced criteria.

With the recent progress in Java powered embedded processors, Java-to-hardware synthesis approaches [1, 2, 17] and reactive multi-processor systems [30–32], a fully automatic system synthesis process, from SystemJ source down to FPGA configuration becomes a definite possibility.

3 SystemJ – The Language

SystemJ extends the Java syntax and semantics with constructs describing reactivity and signal handling along with parallel composition of behaviours. It has support for designing GALS systems, such as systems on a chip (SOC) with multiple clock domains. The behaviour of each clock domain is captured by a programming unit called reaction. A reaction, in turn, may be composed of many concurrent sub-reactions that are purely synchronous. At the top level, a number of reactions synchronize to form a system. A system allows asynchronous coupling of synchronous reactions, which communicate with each other through channels, using CSP-like rendezvous. Thus, SystemJ follows the GALS paradigm. In the following, SystemJ is informally described through an illustrative example, introduced first in [21] as ECL code. A more formal description of SystemJ will follow in subsequent publications.

3.1 Example – A Protocol Stack

The example is a simple protocol stack that inputs bytes, assembles them into packets, and finally checks their CRC and destination address. In addition to the original example from [21], to show the GALS modelling capability of SystemJ, the protocol stack is asynchronously coupled with a test bench, that generates test vectors for the protocol stack. The SystemJ source code for the protocol stack application is given in Listing 1. The Assemble reaction receives bytes asynchronously on a channel (line 23) and assembles them into a packet that is sent synchronously through a valued signal for further processing (line 26). This packet is
awaited for in reaction Checkcrc (line 34), a Cyclic Redundancy Check is carried out on it and the outcome of this test is emitted through a boolean signal further (line 36). Note how both the data and data operations are neatly encapsulated in the Packet, thanks to Java object orientation, making the code more legible. The Prochdr reaction carries out an address match computation (line 47) in parallel with checking the crc_ok signal (lines 51–52). If the packet is faulty, the address calculation is aborted by emitting a signal (line 52), monitored by the address matching sub-reaction (line 46). Finally, the full protocol stack is another reaction, TheStack, which contains just a synchronous composition of the Assemble, Checkcrc, and Prochdr reactions. Note that the valued signals packet and crc_ok are local signals, hidden from the environment outside TheStack.

Listing 1. The protocol stack in SystemJ.

```java
class Packet {
  public final static int HDRSIZE = 6;
  public final static int DATASIZE = 56;
  public final static int CRCSIZE = 2;
  public final static int PKTSIZE = HDRSIZE+DATASIZE+CRCSIZE;
  protected byte[] packet;  
  public Packet() { packet = new byte[PKTSIZE]; }
  public int getCRC() {
    int cnt; Packet buffer = new Packet();
    while(true) {
      abort(reset) {
        for(cnt = 0; cnt < Packet.PKTSIZE; cnt++) {
          receive in_byte;
          buffer.buffer[cnt] = in_byte;
        }
        emit outpkt(buffer);
      } // abort
    }
  }
}

reaction Assemble(in channel reset, in channel Byte in_byte, out signal Packet outpkt) {
  int cnt; Packet buffer = new Packet();
  while(true) {
    abort(reset) {
      for(cnt = 0; cnt < Packet.PKTSIZE; cnt++) {
        receive in_byte;
        buffer.buffer[cnt] = in_byte;
      }
      emit outpkt(buffer);
    } // abort
  }
}

reaction Checkcrc(in channel reset, in signal Packet inpkt, out signal boolean crc_ok) {
  int crc = inpkt.computeCRC();
  emit crc_ok(crc == inpkt.getCRC());
}

reaction Prochdr(in channel reset, in signal Packet inpkt) {
  signal kill_check; boolean match_ok;
  while(true) {
    abort(reset) {
      await inpkt;
      emit;
    } // abort
  }
}
```

3.2 Local Synchrony

Locally, inside a top-level reaction, SystemJ provides synchronous reactive constructs similar to those of Esterel. Sub-reactions can be composed and nested using the synchronous parallel construct ("||"). The communication among synchronous reactions is carried out via signals. Signals can be read, written (emit, sustain), handled for (present), abandoned on (abort), used to suspend certain activities (suspend) and awaited for (await). Furthermore, they can be combined into expressions that can be used in the aforementioned constructs. Nevertheless, expressions only combine signal statuses, being undefined for signal values. As a tick delimiter, pause is also available. Mechanisms for pre-emption are provided via (weak) aborts, suspends and trap-exit, which can be arbitrarily nested. As in Esterel, the outermost abort, trap or suspend has the highest priority.

3.3 Global Asynchrony

Several top-level reactions may be coupled using asynchronous composition ("<->") and channels. Two asynchronous parallel reactions execute in different clock domains, at their own pace. Send and receive on pure channels can be used by reactions to synchronise across clock domains through rendezvous. Additionally, valued
channels transport data across clock domain boundaries. Semantically, channels are buffers of size one, linking one (source) reaction to an entire (sink) clock domain. The reset channel in the protocol stack application is visible and readable from anywhere inside TheStack’s clock domain. In this (sink) domain, channel statuses and values are sampled at the beginning of a tick, and remain unchanged for the whole tick. The rendezvous is confirmed back to the sender at the end of the (source) tick. Thus, the reactions involved in a rendezvous, do not continue their execution at the same moment, but only at their own clock domain tick boundary. In addition, multiple reactions from the same clock domain can read the same channel in the same tick, making for a multiple rendezvous (single send, multiple simultaneous receive). Furthermore, channels may be used in suspend and abort the same way as signals can, or even in complex expressions with other signals or channels.

3.4 SystemJ and Formal Verification

One of the reasons for introducing the synchronous execution model in Java is the solid formal support for synchronous languages. The longer term intention is to provide a constructive operational semantics for SystemJ similar to that of Esterel [5]. Currently, however, we examine the possibility of translating SystemJ to Communicating Reactive State Machines (CRSM) [29], which already have a sound formalism behind them.

States in a CRSM correspond to end of ticks or pauses in SystemJ. A transition, its input condition together with the output signals corresponds to an execution path between two pauses. The SystemJ asynchronous composition and channel communication can be easily translated into CRSM asynchronous coupling with rendezvous states. Furthermore, the SystemJ pre-emption through aborts and traps can be translated to states hierarchy in CRSM.

Although SystemJ cannot be mapped straightforward to pure CRSM, the extended CRSM (ECRSM) formalism [29] adds to CRSM all the other features that SystemJ needs. Namely valued signals, variables, entry and exit procedures, and expression guards on transitions from ECRSM can be smoothly used to take care of the data flow from SystemJ. Furthermore, parallel composition, which terminate when all its sub-reactions terminate, can be easily modelled by ECRSM sink states and nonpreemptive transitions. An example of translation to an extended CRSM from a SystemJ reaction from our working example is depicted in Figure 2. Two CRSM are composed asynchronously (with //), namely TestBench and TheStack. The latter is further composed of four synchronous CRSM. Three of them correspond to the reactions in Listing 1, while the fourth, Reset generator, is used to transform the asynchronous reset into an internal synchronous reset signal, resetJ. Note that rendezvous states (grey in the figure) are used to model send and receive of the reset and data asynchronous signals (channels). In the following we focus on explaining only how to derive the ECRSM for the Prochdr reaction (rightmost in Figure 2), the rest of the ECRSM being obtained in a similar manner. The eternal while loop is modelled by a hierarchical state, with a nonpreemptive transition (dashed line) having a true guard and no outputs. Whenever the internal state machine reaches its sink state (containing a grey blob), the nonpreemptive transition is immediately taken, meaning that the loop body is restarted. The abort block contains a parallel composition of state machines, and exits either through the pre-emptive transition (contiguous line) guarded by the reset signal, or through the nonpreemptive transition (dashed line) when both of the internal state machines reach their sink states. The internal state machines communicate through kill, check signal, allowing one to abort a lengthy address computation in the other.

To summarise, a SystemJ specification can be translated without great effort into an extended CRSM. The number of atomic states of the resulting state machine would correspond to the number of tick delimiters in the source, while the number of transitions to the number of possible control flows between tick delimiters. All the formal reasoning available for CRSM can then be indirectly applied to the SystemJ specification.

4 The SystemJ Design Flow and Execution

Depending on the end platform (standard desktop Java environment, embedded Java environment, or embedded platform with reactivity support), we envision three different ways to employ SystemJ in the system design flow, as in Figure 3.

One of the goal features of SystemJ is the ability to use standard Java tools and execute exclusively in a Java runtime environment (unlike Jester). Therefore, the necessary functionality had to be implemented as Java libraries. A Java package, named TReK (True Reactive Kernel) was written in this sense, implementing a kernel of reactive constructs with identical semantics as in Esterel (unlike PureSR). The ability to execute SystemJ specifications in any standard Java environment addresses however just the modelling and verification part of our intended Java-based design environment for embedded systems. An intermediate step in the design process with SystemJ would be to use a Java-enabled embedded processor platform (e.g. JOP, [33]) to prototype and verify the specification. Minimal hardware support is necessary for handling environment signals, such as memory mapped I/O. However, the ultimate goal is to employ reactive processors, which are Java processors extended with hardware for reactive constructs support, in a similar way as in [32]. The performance
boost from using specialised reactive processors should make SystemJ attractive for real embedded systems. This further extends to the use of reactive multi-processor systems and hardware/software co-design. A multi-processor system synthesis approach could be similar to HiDRA [31] or EMPEROR [8], where nodes are Java reactive processors instead. The advantage of a pure software implementation on a multi-processor platform comes from its generality, as new applications may be distributed and executed without modifying the underlying hardware.

SystemJ was designed to rely heavily on already existing Java compilers and runtime environments. In this sense, compiling a SystemJ application for a standard Java desktop environment consists mostly in a translation to standard Java and TReK calls. Nevertheless, it would be possible to employ a more specialised compilation method, either in a Jester-like manner, into an Esterel control and Java data processing. Bypassing Java altogether in order to obtain C code, in a manner similar to the Java2C translator [24] would be yet another way to use SystemJ. These latter approaches would use their own specialised scheduling mechanisms, without employing Java threads. The compilation effort would be considerably greater, while the portability offered by the Java runtime environment would be lost.

### 4.1 The Pre-processor

The main job of the pre-processor is to translate the SystemJ source code into standard Java code, involving also the TReK class library. The translation for a few SystemJ constructs is given in Table 2.

Named reactions are rewritten as classes extending Reaction, runnable as Java threads. Implicit (unnamed) sub-reactions are translated into inner classes, allowing them to access outer class members without the need to pass them as parameters. Parallel compositions are translated to corresponding calls to Reaction.synch_parallel and Reaction.asynch_parallel which take care of forking (joining) new threads in the right clock domains. Abort and trap constructs are translated into try-catch constructs. Exit traps become exception throws, while tick delimiters (pause, await) take care of throwing exceptions in case of active aborts. A number of other rewrites are also performed, but they are rather simple, and not described herein because of the lack of space.

Besides these relatively easy translation steps, the pre-processor needs to facilitate the job of the runtime scheduler, as follows. We refer to signals whose status is known as resolved. Emitting a signal sets its status to present instantly. A signal is considered absent when all its source reactions resolved it (without emitting it). When checking for presence of a signal, a reaction blocks until its status is known (resolved). Consequently, the earlier the status of a signal is known, the earlier the execution can proceed.

![Figure 2. The extended CRSM (simplified) of the protocol stack example from Listing 1](image)

![Figure 3. Uses of SystemJ](image)
Furthermore, resolving absent signals at the beginning of a tick, avoids deadlock situations such as cyclic waits. The pre-processor must, thus, insert resolves as early as possible in each tick for those output signals that are not emitted in that tick. Additionally, the pre-processing phase needs to include various checks that can be carried out statically, such as causality loops, channel (single source, single sink) and valued signal (single source) restrictions, and more. The current implementation of the SystemJ to Java translator employs rewritable reference attributed grammars through the JastAdd II tool [11], using a Java 1.4 front-end.

4.2 The TReK package

The True Reactive Kernel package offers the support for executing SystemJ on any desktop Java 1.5 environment. TReK comprises only sixteen classes and it employs the Java thread scheduling, try-catch mechanism and generics.

4.2.1 Scheduling

Reactions are run through ReactiveThreads, which are hidden from the programmer, but based on the Thread class. Each parallel composition of \( N \) reactions creates \( N - 1 \) new ReactiveThreads for the additional reactions, starts them, runs the \( N \)th reaction and then joins the remaining \( N - 1 \) threads. Exceptions, such as traps and aborts, may prematurely terminate reactions, and thus parallel compositions. Each clock domain manages explicitly two thread queues. In the LTFinishedQueue wait the threads that finished their local tick by executing a pause instruction. The last thread to complete the current local tick notifies all waiting threads and starts a new tick. The other important queue, SignalWaitQueue, is that of reactions waiting for a signal to be resolved (to present or absent). Whenever a signal gets resolved, all the waiting reactions are notified, re-evaluate the signals or expressions they are waiting on, and either continue their execution or go back to sleep. Note that a new tick may not be started unless the SignalWaitQueue is empty, which means that signals must be resolved in each tick. The sooner these are resolved, the earlier waiting threads can complete their current tick. That is the reason why the SystemJ to Java translator must insert signal resolves as soon as possible in each tick. Note also that reactions in channel sends and receives continue to execute empty ticks until the other side of the channel reaches the rendezvous point. In this way, channel operations remain sensitive to aborts, traps and suspends.

4.2.2 Signals

Both pure signals and valued signals are allowed in SystemJ. The state of any signal is described by its status: PRESENT, ABSENT or UNRESOLVED. A signal can be emitted or read by several reactions, which means that its status is known in a certain tick only after an emit (PRESENT in one reaction), or, if no emit occurs, only after all its possible sources have resolved it (ABSENT in all reactions). SystemJ signals are managed at clock domain level, as new ticks can start only once all signals have been resolved. The method of managing signals in TReK is described in the following.

Every clock domain has two unique bit arrays, describing the emitted signals (ES) and resolved signals (RS). A signal emission will set certain bits in both ES and RS, while signal resolves will only affect RS. At the beginning of a new tick, both of these arrays are cleared. The size of these bit arrays is given by the total number of output signals in each reaction. Thus, every position in the array uniquely identifies one output signal in one reaction. Each TReK signal has an associated Signal Composition Mask (SCM), which is bit array of the same size as ES and RS. The SCM of a signal has 1 on all positions corresponding to reactions that can emit that signal and 0 on all others. Thus, emitting a signal with a certain SCM is equivalent to a bitwise OR between ES (and RS) and SCM, and storing the result back in ES (and RS respectively): \( ES \leftarrow ES \lor SCM \). Checking whether the signal is resolved requires a simple bit-wise AND between RS and SCM and comparing the result with SCM (all sources must have resolved the signal) \( RS \land SCM = SCM \). Checking for presence is similar, except using ES.

Resolving signals is however dependent on the reaction calling the resolve. In this sense every Reaction has a constant bit array, of the same length with ES, RS and all SCMs, named Thread Signal Mask (TSM), which describes the signals output by the reaction. Resolving a signal with SCM from a thread with TSM is then equivalent to \( RS \leftarrow RS \lor (SCM \land TSM) \). To summarise, signal operations are simple bit-wise operations on arrays, easy to implement both as software and hardware.

Also, signals can be combined into expressions using OR, AND and NOT, that can be passed to present, await, abort and suspend. Note however that expressions operate on signal statuses, and not on signal values. Valued signals extend pure signals, also taking advantage of the Java 1.5 generics. They always retain the last value emitted on them, regardless of the order. For deterministic behaviour, only one source per valued signal and one emission per tick is allowed. Multi-source valued signals using composition functions (as in Esterel) can be modelled using single source signals and additional reactions.

4.2.3 Exceptions

Each reaction has attached a list of TReKExceptions, which can be of any of the sub-classes AbortException, Sus-
Table 2. A few SystemJ constructs and their translation in Java with TReK package.

<table>
<thead>
<tr>
<th>SystemJ</th>
<th>Java/TReK</th>
</tr>
</thead>
<tbody>
<tr>
<td>await(A or B and not C)</td>
<td>await(new Or(A, new And(B, new Not(C))))</td>
</tr>
<tr>
<td>signal byte S</td>
<td>SignalValued&lt;Byte&gt; S = new SignalValued&lt;Byte&gt;(&quot;S&quot;)</td>
</tr>
<tr>
<td>abort(E) { ... }</td>
<td>try { setAbort(E); ... } catch(AbortException ae) { handle(ae, E.getName()); } unsetAbort();</td>
</tr>
<tr>
<td>send C(5)</td>
<td>C.send(5)</td>
</tr>
<tr>
<td>reaction R( in signal int S1, out signal S2)</td>
<td>class R extends Reaction { SignalValued&lt;Integer&gt; S1; Signal S2; ... // more internal signals</td>
</tr>
<tr>
<td>reaction R( in signal int S1, out signal S2)</td>
<td>public R(SignalValued&lt;Integer&gt; S1, Signal S2)</td>
</tr>
<tr>
<td>class R</td>
<td>{ this.S1 = S1; this.S2 = S2; }</td>
</tr>
<tr>
<td>class R</td>
<td>public void run() throws TReKException { super.run(); ... }</td>
</tr>
<tr>
<td>reaction R( in signal int S1, out signal S2)</td>
<td></td>
</tr>
<tr>
<td>R1:R(S1,S2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Signal outputal[][] = {S2 },{S3});</td>
</tr>
<tr>
<td></td>
<td>sync_parallel(r1,outputal);</td>
</tr>
</tbody>
</table>

pendException or TrapException, in any order. Their priority is given by their position in the exception list. This list is traversed in every pause, at the end of the current tick and the beginning of the next in order to check if any exceptions are active. Aborts and traps are weak, meaning that they will be detected at the end of a tick, while suspends are strong, meaning that they are detected at the beginning of a tick. Abort and suspend exceptions have associated signal expressions that have to be evaluated (thus resolved) before deciding whether they are active. Whenever an abort or a trap is activated, pause throws the associated exception, which is caught by a try-catch block. However, this may be try-catch for a lower priority exception, case in which the exception is thrown further. Note that although there is a SuspendException, there is no need for a try-catch block for such exceptions, since they are never thrown. This is because suspends are always handled inside the pause, but at the same time they have to obey the priority rules set by the nesting of suspend, abort and trap blocks. Therefore, in the implementation we chose to handle aborts, traps and suspends in a uniform manner, through a unique list.

4.3 Hardware Support

Using the TReK library in a standard Java runtime environment is a portable way to execute a SystemJ application, even on embedded Java processors. Modelling at this level does not differ from the desktop case. However, interfacing the system with the environment (handling external signals and channels) is a task that requires some attention. A possibility is to allow the environment to access signals and channels through a shared memory, which might lead to performance loss and complicated synchronisation mechanisms. A more efficient way is to implement the signals and channels as hardware signals into or out of the processor, offering also hardware support for emitting, testing and reading signals [32]. In fact some of the embedded Java processors can be easily extended with custom byte-codes and additional hardware [14] in order to speed up the reactive constructs from the TReK library.

A reactive version of the Java Optimised Processor [33], a micro-programmed, three stage pipeline architecture, is currently under development. Unlike other reactive processors (see ReMIC [32]), this one offers efficient reactive support in a multi-tasking environment, where multiple reactions execute at the same time. Bit arrays and masks are used to emit, resolve, and test signals from reactions. For efficiency issues, only a single clock domain is supported by one reactive processor. Thus, SystemJ applications with multiple clock domains will run on multi-processor architectures, employing at least one processor per clock domain.

5 Experimental Evaluation

To test the SystemJ design flow, at least for the desktop platform, we conducted a number of experiments using, at first, small toy examples. However, in order to carry out a meaningful comparison between SystemJ and other system-level languages, we selected a stripped down version of an application first used in [27] to compare SystemC and Esterel. The frequency relay application detailed there is a system that measures the frequency (whose normal value is 50Hz) and its change rate in a power network, in order to protect the power system from overloading. Briefly, if the current thresholds indicate that the frequency is too low or it varies too fast, some loads are disconnected from the network by controlling a number of switches. Loads are gradually connected if the frequency and its rate of change improve. Figure 4 illustrates the architecture of the frequency relay specification in SystemJ without the testbench.
module. The specification consists of two clock domains named DataDominatedDomain and ControlDominatedDomain. The reactions in DataDominatedDomain perform signal processing operations which detect peaks in the input AC waveform. The time between every two consecutive peaks is sent to the frequency status reaction in ControlDominatedDomain, where the frequency and its rate of change are easily calculated. The values are compared against the thresholds that are received from a communication network through a reaction called the communication protocol. The comparison result is then emitted to the switch control reaction which adjusts the load level in the power system.

![Figure 4. Frequency relay in SystemJ.](image)

As the frequency relay has already been written in SystemC and Esterel, we only needed to rewrite it in SystemJ in order to compare all three implementations. The SystemJ source for frequency relay has around 301 lines of code, being shorter than its SystemC (382 lines) and Esterel (358 lines) versions. This is somewhat expected, as SystemJ combines the data processing and encapsulation characteristics of SystemC with the synchronous constructs of Esterel, taking the best of the two worlds. With respect to SystemC, SystemJ required fewer lines of code for control-dominated processes. In particular, the difference was obvious in the communication protocol, which makes extensive use of pre-emption. Pre-emption is best specified with statements such as abort and trap that are present in SystemJ but not in SystemC. On the other hand, the advantage of SystemJ with respect to Esterel was in the data-dominated processes. Both the averaging filter and the symmetry filter need arrays to specify delay lines. The arrays were contained inside C functions instead of Esterel code, and they had to be made static in order to keep their values between function calls. This is an example of unnatural solution which may arise when a language relies on another one for data processing. Even in the latest Esterel version that supports arrays (v7), limitations remain since it is difficult to create complex data structures.

When comparing the execution time of the three versions, SystemJ is slightly slower than SystemC, but both are considerably faster than the Esterel version. More exactly, the execution of the SystemC specification (compiled in MS Visual C++ 7.1) takes 6 seconds. SystemJ with JRE 1.5 in Eclipse 3.1.1 takes 23 seconds. Finally, the Esterel specification, using Esterel Studio v. 4, takes about 5 minutes to complete. Newer versions of Esterel Studio, however, might produce slightly faster code. All experiments were carried out on a Pentium 4, 3.2GHz with 1GB RAM machine. The reason for this discrepancy, as observed in [27], seems to come from the fact that the whole Esterel system has to run with one clock. The clock frequency is chosen to match the inputs with the highest changing rate, which means that this speed is unnecessarily high for some data-dominated parts. To summarise, a great deal of time is spent in computing data with unchanging inputs. This is avoided in SystemJ by using different clock domains for the data and the control dominated parts. It appears that in the recent multi-clock version of Esterel this problem can be alleviated in a similar manner.

6 Conclusions and Future Developments

In this paper, we briefly examined a number of system-level languages, and compared them according to six criteria essential for system-level specifications. We then introduced a new system-level language called SystemJ, which is an extension of Java with Esterel-like reactivity and asynchronous constructs. The result is a language that can model Globally Asynchronous Locally Synchronous systems, with support for both data-dominated and control-dominated applications. Being based on a Java library (TReK package) and employing a pre-processor for source-to-source translation, SystemJ may be readily used on a regular Java desktop platform, a Java enabled embedded system, or embedded systems with reactivity support. The ultimate architectural support for SystemJ is intended to be a multi-processor platform composed of Java-enabled embedded processors with reactivity support. We briefly and informally described the syntax and semantics of SystemJ, along with the current implementation of the necessary support. Finally, we presented the experimental results obtained from using a SystemJ approach, as opposed to SystemC and Esterel, in the case of a real-life application, namely frequency relay. To conclude our analysis, we are confident that SystemJ with its features inherited from Java and Esterel would make specification, modelling, and, with the right hardware support, implementation of embedded systems more efficient.

We plan to continue the development of SystemJ on several directions. We intend to give a formal semantics for SystemJ shortly. Tools for visualisation and debugging of SystemJ programs are currently being developed as Eclipse plug-ins. A new architecture for a reactive Java optimised processor is under investigation. Furthermore, we are looking at uses of SystemJ beyond design automation, such as security policies or SystemJ specific garbage collection.
Acknowledgement

This research has been supported by the University of Auckland research grant and was carried out during the first author’s post-doctoral fellowship at the Dept. of Electrical and Computer Engineering, the University of Auckland.

References