The Cardiac Pacemaker – SystemJ versus Safety Critical Java

Heejong Park, Avinash Malik, Muhammad Nadeem and Zoran Salcic
Department of Electrical and Computer Engineering
University of Auckland
Auckland, New Zealand
hpar081@aucklanduni.ac.nz, avinash.malik@auckland.ac.nz, muhammad.nadeem@auckland.ac.nz,
z.salcic@auckland.ac.nz

ABSTRACT

A cardiac pacemaker example is used to compare and contrast the *Synchronous Reactive* (SR) programming model of SystemJ with the SCJ programming model. Our pacemaker is implemented in the synchronous subset of the *Globally Asynchronous Locally Synchronous* (GALS) SystemJ, which extends the Java language with reactivity, concurrency and real-time constructs based on a formal mathematical framework. The use of different programming models results in different design choices and implementations. The SR programming model is driven by a logical clock, which clearly demarcates the state boundaries and is ideal for formal verification of functional and real-time properties. Unlike the preemptive scheduling model prescribed by the SCJ specification, the SystemJ program execution model is atomic and non-preemptive between two logical ticks, and as such it is statically schedulable without the need for a runtime scheduler. To check the effectiveness of the SystemJ approach, we implemented the cardiac pacemaker on three different execution platforms that demonstrate feasibility of guaranteed real-time of the pacemaker execution with a fraction of the used processor's resources.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: safety-critical

Keywords

SystemJ, Safety critical Java, safety-critical, case study, pacemaker

1. INTRODUCTION

Safety critical systems are systems whose failure can lead to catastrophic damage, including loss of life. Engineering safety-critical systems is a complex task, with many interdisciplinary mechanisms integrated via software programs. Since software plays such an essential role in safety-critical system design, safety-critical standards for software program certification [10] have been adopted by industry. *Safety critical Java* (SCJ) [14] is a community effort to introduce a safety-critical subset of *Real-time Java Specification* (RTSJ) [3] with the aim that Java applications adhering to the SCJ specification can be certified for safety-critical applications, such as the avionics applications certified to DO-178B standard.

Static code-analysis techniques, such as data-flow analysis and control-flow analysis, substantially help with the certification process. Moreover, with growing complexity of safety-critical software, *formal verification* – the ability to *prove* that semantics of a system are correct, independently of the modeled system [21] is gaining traction [4] in industry. In fact, formal verification of software systems is one of the grand challenges of computing [8] and the cardiac pacemaker specification [1] is proposed as a potential case study [16, 25].

Recently, a subset of the cardiac pacemaker specification was implemented as a case study of a safety-critical software application in SCJ [24]. The purpose of this case study was to highlight: (1) the expressive power of the Java language and the SCJ specification for designing high-integrity systems and (2) comparing and contrasting, to find the deficiencies in the SCJ implementation, to the Ravenscar ADA [6] implementation, since Ravenscar ADA is a widely adopted language for designing high integrity systems.

Many safety-critical software systems are *reactive* – programs that wait on occurrence of input events and then react to these events to produce one or more outputs. The cardiac pacemaker is no different. In this paper we argue for the need to program a reactive application with a reactive programming language, adhering to a precise formal semantics and *Model of Computation* (MoC). More specifically, we compare and contrast the design of the cardiac pacemaker software in the SystemJ [17] programming language, which extends the Java programming language with reactivity, concurrency (synchronous and asynchronous), and real-time

JTRES '14, October 13 - 14 2014, Niagara Falls, NY, USA

http://dx.doi.org/10.1145/2661020.2661030

specification within a framework of a formal MoC, versus the SCJ implementation [24] that does not have formal foundation. We are primarily interested in comparing the concurrency, reactivity, and timing semantics of a synchronous reactive MoC vis-à-vis the SCJ specification. The cardiac pacemaker is the perfect case study for such a comparison, because although with complex control requirements, the pacemaker has no use of dynamic memory allocation.

There are multiple reasons that this case study would be of interest to the SCJ and the synchronous programming communities: (1) this case study highlights the advantages and disadvantages of the classical [13] scheduling model of SCJ versus the clock-driven reactive model of synchronous languages à la Esterel [2], (2) SystemJ allows specifying real-time as first class language construct, which is tightly coupled with the semantics of the language, thereby allowing formal verification of not only functional properties, but also timing properties and (3) most importantly, the differences in the MoCs lead to two very different implementations, which are worth studying, in order to compare the different high-integrity design considerations that these two quite different approaches promote.

Much of the work [9, 15, 16] related to pacemaker implementation is geared towards formal verification of properties of the pacemaker implementation. Although we can formally verify functional and real-time properties [20] of our pacemaker implementation in SystemJ, using *Linear Temporal Logic*, describing this formal verification approach is out of the scope of this paper and the reader is referred to [20].

The rest of the paper is arranged as follows: the background information about the required pacemaker functionality and the SystemJ programming model are described in Section 2. Section 3 describes the detailed implementation of the pacemaker control logic in SystemJ. In Section 4, we describe the differences between the programming choices of the SCJ and the SystemJ implementations of the pacemaker. Section 5 then gives the experimental results of implementing the pacemaker on three different time analyzable platforms. Finally, we conclude in Section 6.

2. BACKGROUND

2.1 The overview of the pacemaker control logic and its SCJ implementation.

For sake of completeness, in this section, we give a brief description of the cardiac pacemaker control logic and an overview of its implementation in SCJ as described in [24]. For a complete description, the reader is referred to [24].

2.1.1 The cardiac pacemaker functionality

The primary functionality of a cardiac pacemaker is to maintain a regular heartbeat. In general, a pacemaker *senses* signals from the heart and *paces* the different chambers (Atrium and Ventricle) depending upon the received input signal and the operating mode. The official pacemaker specification [1] consists of different requirements/modes dependent upon the patient's condition. In this paper we consider two of these operating modes: **DDDR** and **DDIR**. The DDDR mode specifies that pacing is required in both (**D**ual) the chambers, the pacemaker is sensing in both (**D**ual) the chambers, responding by triggering or inhibiting the pace signals (**D**ual) in both the chambers, and the operating mode depends upon the physical activity of the patient (**R**ate). The DDIR mode, on the other hand, requires that the pacing signals be inhibited (**I**) in response to the sensed signals.

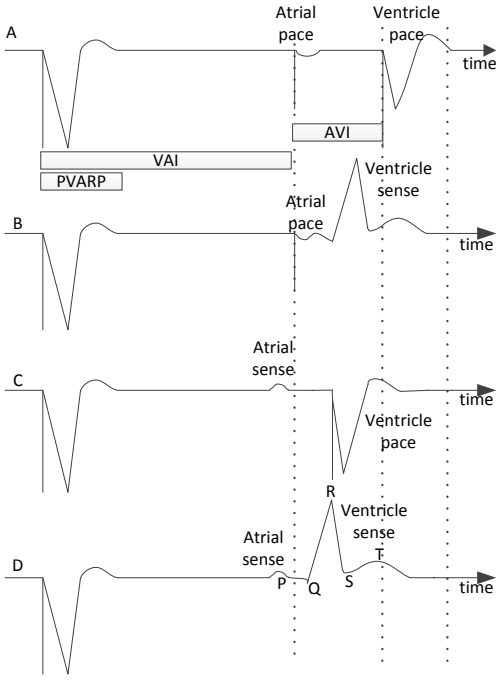

Figure 1. The DDDR operating mode scenarios. Scenario A indicates pacing activity in atrium and ventricle. Scenario B indicates only pacing the atrium. Scenario C indicates pacing in only the ventricle. Finally, D indicates a normal working heart.

Figure 1 shows the different scenarios of the DDDR operating mode of the pacemaker. In the case of a normal working heart, shown by scenario D, an electrical pulse contracts the Atrium (shown by the atrial sense signal, also called the P wave) a fraction of a second before the contraction of the ventricle (indicated by the ventricle sense signal, also called the QRS complex). This fraction of a second is enough to empty the blood from the atrium to the ventricle. In case of arrhythmia, after sensing the ventricle activity, the pacemaker waits for ventriculoatrial (VAI) interval, sensing for an atrium activity. If no activity is sensed, an atrial pace is generated (see scenario A). Similarly, a ventricle pace is generated after the atrioventricular (AVI) interval if no ventriclular activity is detected during this period. The pacemaker needs to ignore

false atrial activity after a ventricle pace denoted by the time length PVARP in scenario A. Finally, a T wave is generated after ventricle activity, which indicates expansion of the ventricle chamber, it is imperative that the pacemaker does not produce a pulse during the T wave.

The DDIR mode slightly differs from DDDR mode in scenario C. In this situation, the AVI timeout period is started as soon as the atrial activity is sensed, if the pacemaker is in the DDDR mode. On the other hand, the AVI timeout is started only after expiration of the VAI timer, if the pacemaker is in the DDIR mode, but the sensor is still allowed to sense ventricle activity as soon as the intrinsic atrial activity is occurred in the atria chamber. The pacemaker can switch from the DDDR to the DDIR mode, if the time between two atrial events (sense or pace) is less than some threshold, the *Mode Switching Interval* (MSI), on some consecutive occasions. A switch is made back to the DDDR mode, if the time between two atrial events is greater than some threshold for some consecutive occasions.

The time intervals for the different waves are shown in Table 1 for an upper heartbeat rate of 120 beats/minute. We use the exact same time intervals as the SCJ implementation in [24] of the cardiac pacemaker for a fair comparison.

Table 1. Time intervals for a single heartbeat pacing cycle

Time intervals	Purpose	Time (ms)
P wave length	Time for sensing atrium activity	110
Duration of pulse	Time for which pacing signals should be maintained	1
QRS complex length	Time for sensing ventricle activity	100
AVI length	Time for ventricle fill after atrial contraction	150
VAI length	Time between ventricle activity followed by atrial sense	850
PVARP length	Ignoring false atrium activity	350
MSI length	Time between atrial events to switch mode	500

2.1.2 The SCJ implementation of the cardiac pacemaker

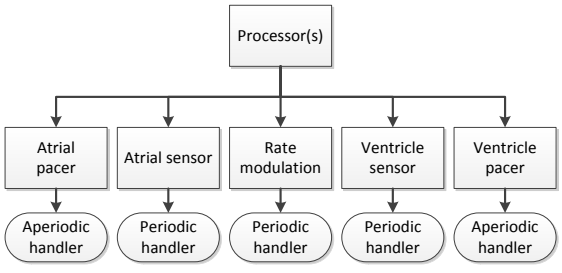

Figure 2. Pacemaker sensing and pacing components

The software architecture of the SCJ implementation of the pacemaker is replicated in Figure 2 from [24]. Sensing and pacing logic are implemented in different handlers. The sensing activity is implemented via periodic handlers released at half the length of the P wave and the QRS complex, respectively. These periodic handlers in turn release the aperiodic handlers, if no heart activity is sensed as described in previous subsection. The DDDR and DDIR modes are implemented as two separate SCJ missions and the switch is made via the sensed atrial activity kept in the immortal memory.

The aperiodic handlers are given the highest priority and a rate monotonic priority ordering policy is recommended for the different SCJ handlers. The design of the cardiac pacemaker resulted in the introduction of one-shot timers in the SCJ specification, which can be used to control the pacing activity rather than aperiodic handlers as is done in [24].

2.2 SystemJ syntax, informal semantics and model of computation

SystemJ is amenable for designing, modeling, and implementing reactive and concurrent software systems, which always follows the *Globally Asynchronous Locally Synchronous* (GALS) *Model of Computation* (MoC). A SystemJ program, as shown in Figure 3, may consist of one or more asynchronous concurrent behaviors (processes) called a clock-domain. A SystemJ program consists of a fixed number of clock-domains, which represent the top-level of a SystemJ program. Clock-domains execute asynchronously at their own pace, defined by their own logical clocks. This asynchronous concurrency fits naturally with the characteristics of embedded reactive applications. Each clock-domain can be further refined into one or more synchronous concurrent sub-behaviors, which are called reactions. Multiple reactions in the same clock-domain execute concurrently and advance their states in lockstep, following the logical clock of the clock-domain they belong to. Inputs for all reactions within a clock-domain are sampled at the beginning of the logical clock tick and the dependent outputs are generated at the end of the tick. During clock-domain execution, reactions advance in lock-step with the logical clock, known as a tick, and communicate with the external environment on these tick boundaries. From a logical point of view, a tick is considered instantaneous.

The communication, as shown in Figure 3, between

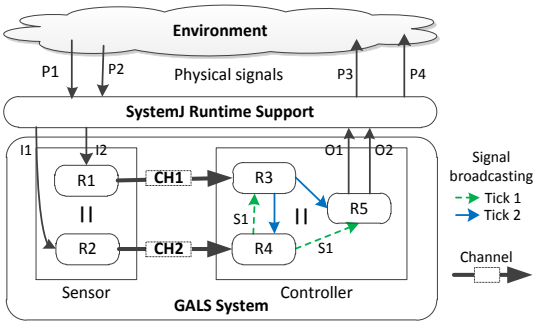

Figure 3. Graphical representation of SystemJ system

reactions and their individual environments and between different reactions in the same clock-domain is via objects called signals. Communication between reactions in different clock-domains is carried out via blocking, CSP [7] style, objects called channels.

The execution of reactions is explicitly defined by fine-grained constructive semantic rules of the GALS MoC. All SystemJ programs are correct by construction and are derived from the reactive kernel statements [17]. Because the GALS MoC incorporates the formal *Synchronous Reactive* (SR) MoC [2], a system designed in SystemJ is effectively a reactive system. Synchronous concurrency among multiple reactions within a single clock-domain is statically analyzed and compiled to efficient sequentialized and functionally deterministic code, which eliminates the overhead and possible errors of creating and handling interleaving in multiple interrupt service routines and multi-thread implementation as in traditional programming languages [11].

A complete list of SystemJ kernel statements is shown in Table 2 Sequential SystemJ statements are separated by a semicolon "`;`" similar to C and Java. The tick boundaries are explicitly specified with `pause` – every reaction in a clock-domain has to reach one of the pause statements in order to begin the next tick. It should be noted that pause or any other derived statements enclosing pause are the only statements that consume logical time in SystemJ. When a clock-domain pauses, all the output signals emitted in the instant become available to the environment, and all the input signals to be processed in the next tick are captured from the environment. Statements, such as `present`, `abort` and `suspend`, allow programmers to describe control-flow of a program based on presence or absence of the SystemJ signals. Signals allow broadcast-based communication between reactions within the same clock-domain whose statuses can be set to true for one tick using the `emit` statement. In addition, signals can also carry data in a form of Java object which, in this case, we call them valued signals. The `while` statement is a temporal loop that must contain at least one pause inside its body. Software exception similar to interrupt service routine is done through `trap` and `exit`. Lastly, SystemJ reactions are composed using the synchronous parallel operator (`||`) and make transition in lock-steps with the clock-domain's tick.

Table 2. SystemJ syntactic constructs

Statement	Description
`p1;p2`	p1 and p2 in sequence
`pause`	Consumes a logical instant of time (a tick boundary)
`[input] [output] [type] signal S`	Declaring a pure or valued signal
`emit S [(exp)]`	Emitting a signal with a possible value
`while(true) p`	Temporal loop
`present(S){p1}else{p2}`	If signal S is present do p1 else do p2
`[weak] abort ([immediate] S) {p}`	Preempt if S is present
`[weak] suspend ([immediate] S) {p}`	Suspend for 1 tick if S is present
`trap (T) {p}`	Software exception
`exit T`	Throw a software exception
`p1 || p2`	Run p1 and p2 in lock-step parallel
`[input] [output] [type] channel C`	Declaring input or output channel
`send C[(exp)]`	Sending data over the channel
`receive C`	Receiving data over the channel
`#C`	Retrieving data from a valued signal or channel
`wait_inbetween(M..N)`	Postpones the current reaction from executing the following statement between M and N time units
`wait_atleast(M)`	Postpones the current reaction from executing the following statement for minimum of M time units
`wait_exact(M)`	Postpones the current reaction from executing the following statement for exactly M time units

The Asynchronous part of SystemJ (i.e. inter-clock-domain) is *rendezvous* style message passing over channel. A reaction in the sending clock-domain passes data over a channel using the `send` statement. On the other hand, the data can be received by a reaction in the receiving clock-domain with `receive`. Channels are point to point. Sending or receiving reaction blocks until the other end is ready to participate in the channel communication. Data transferred through a channel or signal can be retrieved using the hash (`#`) operator.

Programmers can express timing behaviors of the system through three types of SystemJ constructs: `wait_inbetween(M..N)`, `wait_atleast(M)` and `wait_exact(M)`. Non-exact real-time delays can be modeled using the `wait_inbetween(M..N)` statement, which prevents a clock-domain or reaction from proceeding to the next statement for any time between M and N time units. Similarly, the `wait_atleast(M)` statement is used to postpone the control-flow for a minimum of M time units. However, the maximum postponement is unbounded. Finally, exact timing behaviors can be modeled using `wait_exact(M)` that is used to program exact timeouts, periodic tasks, etc. It is important to note that these real-time constructs are a part of the SystemJ language; they do not rely on the external resources such as timers. In fact, physical time is mapped to clock-domain logical ticks. The SystemJ tool chain obtains the worst case reaction time (WCRT) of a program for a specific execution platform, which is a basis of determining number of ticks satisfying the times specified in the wait statements [19].

3. DESIGN OF THE SYSTEMJ PACEMAKER CONTROL LOGIC

In this section, SystemJ implementation of the cardiac pacemaker is presented.

3.1 The pacemaker control logic in SystemJ – an overview

SystemJ is designed for implementing GALS systems, but a single clock-domain (synchronous program) suffices to implement the cardiac pacemaker functionality.

Figure 4 shows the implementation of the pacemaker as a SystemJ clock-domain. The single clock-domain consists of three synchronous parallel reactions called: AtriumSensorPacer, VentricularSensorPacer and OperatingModeHandler, which sense and pace the two chambers of the heart and change the operating mode of the pacemaker if needed. These *top-level* reactions are composed of further lower-level reactions.

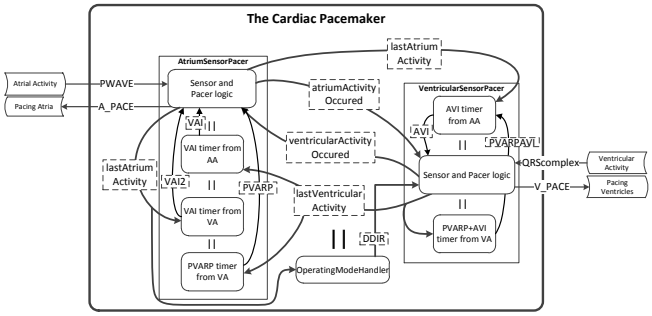

Figure 4. The cardiac pacemaker implemented as a synchronous program in SystemJ.

The AtriumSensorPacer reaction consists of the main control logic for sensing the incoming P wave from the atrium and pacing the atrium if no P wave is detected within VAI time units. Three concurrent reactions run in parallel with the main control logic: two VAI timeouts are implemented, using the `wait_exact` construct, one started after the last ventricular activity and the other after the last atrium activity. The third concurrent reaction implements the PVARP timeout after the last ventricular activity. If no P wave is detected within the VAI and after the PVARP timeouts, the AtriumSensorPacer reaction checks if the VAI timeout from the last atrium activity has expired, if so, the atrium is paced. Similar control logic is implemented for sensing and pacing the ventricle. All communication is via signals. The input from the heart is sensed via input signals, while the communication between different reactions in the clock-domain is carried out using the signal broadcast mechanism provided by the `emit` construct. Lastly, OperatingModeHandler reaction checks time intervals between two consecutive atrial events and changes the operating mode of the pacemaker between DDDR and DDIR modes.

```
1  Pacemaker(
2    input signal PWAVE, QRScomplex;
3    output signal A_PACE, V_PACE;
4  )->{
5    signal VAI,DDIR;
6    boolean signal lastVentricularActivity,lastAtriumActivity,
7                   atriumActivityOccured,
8                   ventricularActivityOccured;
9    // Initialization
10   emit ventricularActivityOccured(true);
11   emit atriumActivityOccured(false);
12   pause;
13   AtriumSensorPacer(:PWAVE,lastVentricularActivity,
14          lastAtriumActivity,atriumActivityOccured,
15          ventricularActivityOccured,VAI)
16   ||
17   VentricularSensorPacer(:lastVentricularActivity,
18          atriumActivityOccured,ventricularActivityOccured,
19          lastAtriumActivity,QRScomplex,VAI,DDIR)
20   ||
21   OperatingModeHandler(:lastAtriumActivity,DDIR)
22 }
```

Figure 5. Top-level SystemJ pacemaker clock-domain

```
1  reaction VentricularSensorPacer(:
2  output signal lastVentricularActivity,
3  output boolean signal atriumActivityOccured,
4  output boolean signal ventricularActivityOccured,
5  input signal lastAtriumActivity,
6  input signal QRScomplex,
7  input signal VAI,
8  input signal DDIR
9  )
10 {
11   signal AVI,PVARPAVI;
12   { // Ventricle sensor/pacer
13     while(true){
14       if(#ventricularActivityOccured == Boolean.FALSE){
15         // Reading a sensor
16         abort(immediate AVI){
17           trap(IntrinsicQRS){
18             while(true){
19               abort(immediate !QRScomplex){
20                 wait_exact(100 ms);
21                 exit(IntrinsicQRS);
22               }
23               pause;
24             }
25           } do{
26             emit lastVentricularActivity;
27             emit atriumActivityOccured(false);
28             emit ventricularActivityOccured(true);
29             }
30         } do{
31           // Release ventricular pacer
32           // Time elapsed since lastVentricularActivity >= PVARP+AVI
33           present(PVARPAVI){
34             emit lastVentricularActivity;
35             // Pace current for 1 ms
36             trap(T){
37               {sustain V_PACE;}||{wait_exact(1 ms); exit(T);}
38             }
39             emit atriumActivityOccured(false);
40             emit ventricularActivityOccured(true);
41           }
42           pause;
43         }
44       }
45       pause;
46     }
47   }
48   ||
49   {
50     {// AVI Timer
51       while(true){
52         abort(lastAtriumActivity){
53           wait_exact(150 ms);
54           sustain AVI;
55         } do{
56           present(DDIR){
57             await(immediate VAI);
58           }
59         }
60       }
61     }
62     ||
63     {// PVARP+AVI Timer
64       while(true){
65         abort(lastVentricularActivity){
66           wait_exact((350 + 150) ms); // PVARP + AVI
67           sustain PVARPAVI;
68         }
69       }
70     }
71   }
72   pause;
73 }
```

Figure 6. SystemJ implementation for the ventricular sensor and pacer control logic

3.2 The detailed SystemJ pacemaker control logic implementation

Figure 5 shows the SystemJ clock-domain code, called `pacemaker`. The main logic of the clock-domain starts inside the curly bracket followed by `->` (line 4). First, the internal signals for the communication between reactions are declared. As one can see, the signals can be either pure (line 5) or valued (line 6) depending on whether reactions are notifying simple events or also required to pass more complex data-structures (e.g. Java objects). After initialization, by setting values of some signals (lines 10-11), the clock-domain finally *forks* AtriumSensorPacer, VentricularSensorPacer and OperatinModeHandler reactions to run in parallel (lines 13-21). The signals used for communication between the reactions are passed as arguments.

AtriumSensorPacer and VentricularSensorPacer reactions consist of sensing and pacing algorithms for atrium and ventricle, respectively. Due to lack of space we only show the code for the VentricularSensorPacer reaction in Figure 6. Full source code of the pacemaker can be obtained from `https://github.com/hjparker/pacemaker.git`. In this reaction, there are two timers, implemented using `wait_exact` statement, which trigger the signals `AVI` and `PVARPAVI`, respectively upon expiration (lines 50-62 and 64-71) of their respective timeout periods. For instance, AVI timer reaction blocks for exactly 150 ms (line 53) before it starts sustaining the signal `AVI`, indicating atrioventricular interval has expired. Since the start time of `AVI` is the last atrium activity, the timer should restart for every atrial activity. This functionality is achieved through the reactive `abort` statement, which preempts the on-going computations of its enclosing body upon the presence of the signal `lastAtriumActivity`. In case of preemption, control-flow terminates the current execution (i.e. `wait_exact` or `sustain`) and jumps inside the `do` block (lines 55-59), which is the handler logic for the `abort` statement. According to the pacemaker specification, `AVI` timeout does not start until full `VAI` period has expired when the pacemaker operates in DDIR mode [1]. Therefore, depending on the mode start of the timer, measuring `AVI` period can be delayed until `VAI` expiration (line 56-58). Similarly, to prevent very frequent ventricular pacing, another timer has been used for emitting the signal `PVARPAVI` indicating that enough time has been elapsed from the last ventricular activity.

The ventricle sensor/pacer logic starts by checking the Boolean variable of the valued signal (line 14). The signal is set to false by the AtriumSensorPacer reaction to make sure only one of the reactions is responsive at any time. The ventricular activity is sensed when the signal `QRScomplex` is present *continuously* for 100 ms. Whenever the signal becomes absent between 0 to 100 ms, the timer resets to 0 and sensing activity restarts. This is achieved by preempting the timer on *negation* of the signal `QRScomplex` (line 19). If ventricular activity is sensed, VentricularSensorPacer reaction is disabled by emitting the signal `ventricularActivityOccured` with value true (line 28). Additionally, AtriumSensorPacer reaction is re-enabled by emitting the signal `atriumActivityOccured` with value false (line 27). Contrarily, if the ventricular activity is not sensed within `AVI` period, the control-flow preempts to the pacer logic (line 30-41). As explained previously, checking the presence of the signal `PVARPAVI` ensures absence of the ventricular activity just before starting the pacing activity (line 33). Sustaining the pacing signal (`V_PACE`) for 1 ms is achieved by synchronously composing the `sustain` statement with the `wait_exact` statement (line 37). As a result, the `trap` statement preempts the `sustain` statement after a 1 ms delay. Finally, two valued signals are emitted for switching to the AtriumSensorPacer as explained before.

There are a number of differences between the SystemJ implementation of the pacemaker control logic vis-à-vis the SCJ implementation. In the next section we highlight these differences.

4. A comparison between SystemJ the SCJ approach to the cardiac pacemaker control logic

In this section we highlight the differences in the SCJ and the SystemJ programming models and their consequences on the design choices.

4.1 Synchronous clock-driven reactivity versus the SCJ programming model

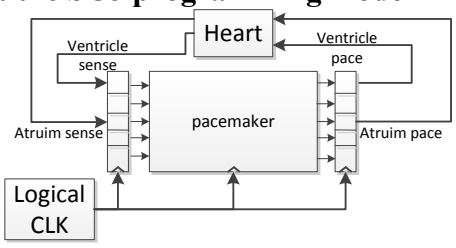

Figure 7. The clock-driven pacemaker control logic in SystemJ

The SystemJ clock-domain, like any other synchronous program, follows a clock-driven model, in contrast to SCJ's programming model. Every clock-domain interacts with the environment, in this case the heart, via a set of input and output registers (buffers in software). The period of the logical clock driving the SystemJ clock-domain is determined by the execution path in the SystemJ clock-domain. The worst case period also termed the *worst case reaction time* (WCRT), is determined by the critical path in the program. This statically analyzable WCRT [12] affects a number of design decisions:

- Periodic handlers: In the SCJ implementation, the designer explicitly programs the periodic handlers. In

case of the pacemaker two periodic handlers are run (see Figure 2) with half the period of the P wave and the QRS complex for sensing atrium and ventricle, respectively. The sampling period of the SystemJ control-logic on the other hand is *implicitly* less than or equal to the WCRT (orders of micro-seconds on average in case of the pacemaker controller).

- Multiple event capture and handlers: Unfortunately, SCJ specification does not define semantics of handling events on multiple signal lines occurring simultaneously. SystemJ programs sample events at the logical clock-edge via the input register as shown in Figure 7 and hence, multiple events occurring simultaneously (in the same logical tick) can be handled with ease.
- Aperiodic and sporadic event handlers: The clock-driven model places a *restriction* on the SystemJ programming model. One *cannot* handle aperiodic events in synchronous languages or within a single clock-domain in SystemJ, because every clock-domain's logical tick execution is atomic. Note that aperiodic events can be captured with the GALS model, but this is unnecessary in the pacemaker system. In fact, implicitly, every incoming event from the environment can be considered to be *sporadic* with the minimum inter-arrival time equal to the WCRT of the SystemJ clock-domain (or synchronous program in pure synchronous languages). These incoming events are handled with a periodically running clock-domain (or synchronous program) with the period and deadline equal to the WCRT in the worst case.

4.2 Scheduling model

Figure 8 shows the (partial) internal implementation of the pacemaker clock-domain. The internal signals inside the clock-domain are registered too. For example, the `atriumActivityOccured` signal emission, from AtriumSensorPacer reaction, is updated only at the end of the clock-domain logical tick and hence, this update is visible to the VentricularSensorPacer reaction only at the start of the next tick, i.e., the emission is visible only after a delay of 1 logical clock-tick in this case. This loose coupling of reactions within the clock-domain has multiple consequences:

- Priorities: All reactions within the clock-domain have equal priority, unlike in SCJ, where unique priorities need to be determined and assigned to different handlers.
- Task ordering: The loosely coupled communication model along with the synchronous lock-step (logical clock-driven) execution model allows running synchronous parallel reactions in any order, i.e., irrespective of which reaction is run first, the functional and timing properties are preserved and, hence, deterministic.
- Response time: Unlike in the classical scheduling model [5] that SCJ also follows; response time is *not* defined as the time from the release time to the completion of a handler. Instead, in SystemJ (and in synchronous languages in general) response time is defined as the time from the presence of one or more input events to emission of one or more dependent output events. Multiple synchronous parallel reactions might participate in the emission of the output event(s). Furthermore, the output event(s) might not necessarily be emitted in the same tick as the capture of input events, but might be emitted after multiple ticks, introduced explicitly (via `pause`) or implicitly (via loose communication mechanism as shown in Figure 8). The designers need to be aware of this difference in the definition and semantics of response time.

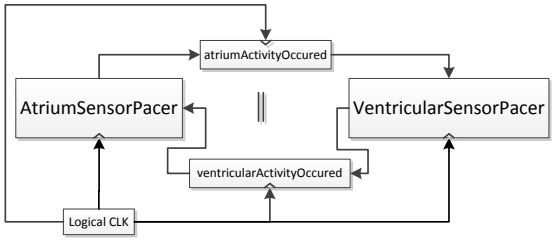

Figure 8. The clock-driven execution within the pacemaker clock-domain in SystemJ

4.3 Timing model

The current SCJ specification [14] allows using external high-resolution timers to trigger one-shot handlers. These timers run in parallel to the currently executing handlers on the processor and invoke a non-periodic time triggered task on a timeout. The inclusion of the one-shot timers, thanks to the pacemaker SCJ case study [24], allows triggering the Atrium and Ventricle pace signals on a timeout rather than via an internally generated software event.

SystemJ takes a different approach to implementing such one-shot timeouts. Specifying real-time behaviors via the `wait` statements (see Table 2), including one-shot timeouts, is a first class language construct in SystemJ, *unlike* other synchronous languages. SystemJ, *like* other synchronous languages is unable to utilize external timers. There are multiple reasons: (1) non-periodic time triggered tasks require preemption of the currently executing task upon timeout. The execution of the clock-domain in SystemJ is atomic, an external timeout cannot preempt a currently executing clock-domain and hence, using external timers is unsuitable, (2) SystemJ clock-domains and synchronous programs in general only capture their input at the start of the logical clock tick; hence, a timeout from an external timer needs to *exactly* align with the logical clock tick, which in turn restricts the resolution of the external timer to the WCRT in the worst case and, finally, (3) external timers and preemption constructs (`suspend` and

`abort`) do not interplay well [19]. Hence, we have proposed and implemented sound and complete algorithms that rewrite real-time timeouts into logical tick timeouts. This rewrite algorithm statically analyzes the clock-domain to determine the WCRT of the program, on the given platform, and then converts the programmer defined real-time specification into a loop that counts the number of logical ticks (equating to the designer specified timeout value) rather than wait for an external timeout, if it is feasible, else an error is generated, see [19] for further details.

There is no scheduling overhead in SystemJ, because every clock-domain is statically compiled into a single process for execution on a single processor. Hence, the designer defined real-time timeout specification is exact, unlike in SCJ, where the implementation needs to compensate (change) the designer defined timeout to include the scheduler execution and context switching overheads.

5. Experimental results

In this section we give the quantitative results of running the pacemaker control logic designed as a SystemJ clock-domain on timing analyzable platforms.

5.1 Brief description of the execution platforms

5.1.1 The tandem processor platform

One of the requirements for running a SystemJ program is the ability of the target platform to execute Java since the data-computations are described in Java. However, it was shown previously that the JVM is not the best option to describe the control part of the SystemJ program [22]. The tandem processor [22] is designed to execute the SystemJ programs more efficiently than only the JVM. The tandem processor architecture consists of REactive CO-Processor (ReCOP) and a data-processor. ReCOP is a small RISC processor core with specialized ISA extensions made specifically to directly support SystemJ's reactive and concurrent features. Furthermore, SystemJ's control-flow is efficiently mapped using ReCOP's ISA. A data-processor can be any processor for which a JVM is available or a processor which can execute Java bytecodes directly. In our experiments, Java Optimized Processor (JOP) [23] is used as a data-processor. A general overview of our tandem processor platform, called *TP-JOP*, is shown in Figure 9.

As one can see, SystemJ compiler automatically partitions the program into two components, control and data, which are mapped to ReCOP and JOP, respectively. ReCOP leads the program execution. When control-flow reaches Java statements, ReCOP calls JOP through a simple single buffer mechanism to carry out a Java data-call, the ReCOP is idle during the data-call processing. JOP on the other hand, waits for ReCOP's data-call, and executes the corresponding Java statement(s). The returning result from JOP to ReCOP is a single bit value that determines the further execution-flow of the SystemJ program.

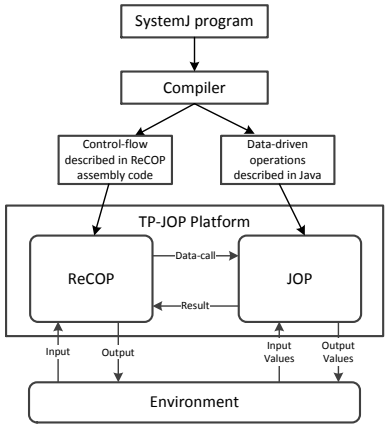

Figure 9. A graphical overview of TP-JOP platform

5.1.2 The Jop-Plus platform

In TP-JOP, the two processor cores are used at the same time, unnecessarily consuming implementation resources (in our case on An FPGA). Also, in case of control dominated applications, the JOP is underutilized as it is polling ReCOP without doing any useful work until a data-call is made.

JOP-Plus [18] is a processor that merges functionalities of the ReCOP core and the JOP core into a single processor that can execute two types of code: the control code that runs on ReCOP and the Java data-computations that execute on JOP in TP-JOP, by extending the instruction set of the original JOP while using single execution unit and data-path. It extends JOP with new components such as Register-File (RF), MAX unit, etc [18], without duplicating the main execution resources of JOP. This allows the resultant JOP-Plus core to appear as two *logical* processors allowing it to execute two different programming models. At any given time, the programming model under execution uses all the resources of the processor while being able to switch to the other programming model and vice versa. Figure 10 gives the basic idea of the JOP-Plus architecture.

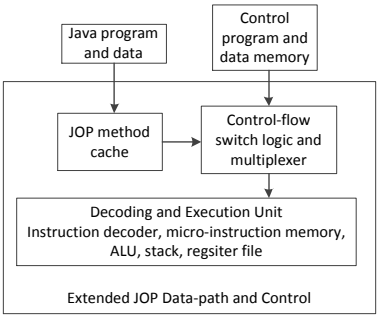

Figure 10. The JOP-Plus architecture

The compilation flow from SystemJ program to the JOP-Plus architecture is the same as in Figure 9.

The execution of the control-flow instructions is implemented by extending JOP. The separation of the control-flow instructions and data-driven operations is maintained by storing those two parts of the SystemJ

program in two different memories. The execution capabilities of JOP are extended with a number of new bytecodes, as well as with the microcodes needed for their implementation. The new core has single instruction decoding and execution unit and performs very efficient switching between the control-flow and data-driven programming models of SystemJ when necessary. This seamless integration simplifies the communication between control and data operations resulting in communication between the two via JOP's stack. This eliminates the use of separate control processor, thus consuming significantly fewer FPGA resources.

5.2 Quantitative comparison

We could not make any quantitative comparisons between SCJ and SystemJ implementation of the pacemaker functionality, because we could not run the SCJ implementation of the pacemaker on JOP, which is the only SCJ compliant platform available to us, due to some of the unimplemented SCJ specific libraries. Still, we decided to do several experiments and comparisons of the SystemJ implementation on different execution platforms, none of them requiring any modification of the SystemJ source specification. We have run the pacemaker example on three different types of platforms, JOP, TP-JOP and JOP-Plus, which are all capable of running SystemJ programs and are time analyzable.

Table 3 shows the logic element (LE) usage for these platforms on Cyclone II FPGA chip on Altera DE2-70 development board. All platforms were configured to run at clock frequency of 70 Mhz. It is shown that TP-JOP has the highest LE usage compared to the rest. We tested the same pacemaker implementation with the four different scenarios in Figure 1 as test benches. A more detailed description of the experimental setup along with the procedure to reproduce the results is given `https://github.com/hjparker/pacemaker.git`. Table 4 gives the results. A pure JOP (Java) implementation of the program is the slowest, while the JOP-Plus implementation results in the fastest measured average logical tick times. The TP-JOP speedup compared to just JOP is as expected, due to the dedicated processor for control-flow operations: the ReCOP.

The JOP-Plus speed up is due to multiple factors. In case of execution of *only* control-flow instructions, TP-JOP outperforms JOP-Plus, but it loses this advantage when Java based data-driven operations need to be executed, primarily because of the switching latency from ReCOP to JOP in TP-JOP, which is much larger than JOP-Plus. Secondly, JOP-Plus invokes the Java data-driven operations (wrapped in methods) directly from the control-flow without invoking the main method, which is used for polling purposes in the TP-JOP architecture, thus resulting in reduced call–graph depth and consequently reduced cache misses. The result of the average tick times has shown that the real-time requirements for the pacemaker (Table 1) can be met for all execution platforms. It is because the average tick times are an order of magnitude smaller than the time intervals required by the pacemaker specification.

Finally, the generated memory footprint is compared in Table 5. Here again JOP gives the worst results followed by TP-JOP and then JOP-Plus. The reduction is memory footprint is due to the specialized control-flow instruction set implemented in both TP-JOP and JOP-Plus.

6. CONCLUSIONS

In this paper we have described the implementation of a cardiac pacemaker in the SystemJ language and compared it with the previous safety-critical Java case study implementation. The pacemaker is a safety-critical system that requires hard real-time and functional correctness guarantees. The SystemJ programming model based on formal semantics provides these guarantees via linear temporal logic property verification, while it is extremely difficult, if not impossible, for the SCJ model.

The SystemJ implementation of the pacemaker significantly differs from the SCJ implementation, because SCJ implementation of the pacemaker uses the classical programming model of preemptive handlers, whereas the SystemJ pacemaker implementation is a single logical clock-driven task. The atomic logical clock-driven execution of the SystemJ implementation implicitly allows handling multiple events occurring simultaneously unlike SCJ, which does not specify the semantics of handling multiple events. The atomicity of a SystemJ clock-domain requires timing specification, specifically timeouts to be first class language constructs, whereas SCJ uses external timers combined with preemptive one-shot time-triggered

Table 3. Logic element usages for different types of execution platforms

Platforms	*Total logic elements used*
JOP	5,484
TP-JOP	7,677
JOP-Plus	6, 320

Table 4. Average tick times for each pacing scenario (us)

Platforms	*Scenario A*	*Scenario B*	*Scenario C*	*Scenario D*
JOP	218.92	207.44	215.57	208.21
TP-JOP	90.88	80.35	102.20	129.42
JOP-Plus	75	73	78	81

Table 5. Generated memory footprint

Platforms	*Control(KB)*	*Java(KB)*	*Total(KB)*
JOP	-	89.875	89.875
TPJOP	10	74.3	84.3
JOP-Plus	10	49	59

handlers to implement the same functionality. Atomicity along with conversion of real-time specifications into logical time also bodes well with the functional and real-time property verification of the SystemJ programs as semantics are well defined. SCJ specification on the other hand, although more flexible, makes formally verifying properties much harder due to the preemptive scheduling model.

As two final points we would like to highlight the qualitative and the quantitative efficiency of SystemJ vis-à-vis SCJ implementation of the pacemaker. The reactive and concurrency abstractions provided by the SystemJ language allow the designers to concentrate on the design itself. This becomes clear from the observation that the complete pacemaker control logic as a single SystemJ clock-domain is around 200 lines of SystemJ code, whereas the SCJ implementation with the same functionality is around 500 lines of Java code. Moreover, this abstraction does not necessarily lead to inefficient implementation, as observed from the micro-second execution times obtained during benchmarking.

7. REFERENCES

[1] Boston Scientific, 2007. Pacemaker system specification. Technical Report., http://www.cas.mcmaster.ca/sqrl/SQRLDocuments/PACEMAKER.pdf.

[2] Berry, G. and Gonthier, G., 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming 19*, 2, 87-152.

[3] Bollela, G., J. Gosling, B. Brosgol, P. Dibble, S.Furr and M. Turnbull, 2000. The Real-Time Specification for Java. Addison-Wesley.

[4] Boulanger, J.-L., 2012. Industrial Use of Formal Methods: Formal Verification. John Wiley & Sons.

[5] Bril, R., Lukkien, J., and Verhaegh, W.J., 2009. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Systems 42*, 1-3 (2009/08/01), 63-119.

[6] Burns, A., 1999. The ravenscar profile. *ACM SIGAda Ada Letters 19*, 4, 49-52.

[7] Hoare, C.A.R., 1978. Communicating sequential processes. *Communications of the ACM 21*, 8, 666-677.

[8] Hoare, C.A.R., Misra, J., Leavens, G.T., and Shankar, N., 2009. The verified software initiative: A manifesto. *ACM Comput. Surv. 41*, 4, 1-8.

[9] Jiang, Z., Pajic, M., Moarref, S., Alur, R., and Mangharam, R., 2012. Modeling and verification of a dual chamber implantable pacemaker. In *Proceedings of the the 18th international conference on Tools and Algorithms for the Construction and Analysis of Systems* (Tallinn, Estonia2012), Springer-Verlag, 2260535, 188-203.

[10] Johnson, L.A., 1998. DO-178B, Software considerations in airborne systems and equipment certification. *Crosstalk, October*.

[11] Lee, E.A., 2006. The Problem with Threads. *Computer 39*, 5, 33-42.

[12] Li, Z., Malik, A., and Salcic, Z., 2014. TACO: A Scalable Framework for Timing Analysis and Code Optimization of Synchronous Programs. In *Proceedings of the International Workshop on Embedded Multi-core Systems and Applications (IWMSA 2014)* (Congqing China2014). http://homepages.engineering.auckland.ac.nz/~amal029/p3.pdf.

[13] Liu, C.L. and Layland, J.W., 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM 20*, 1, 46-61.

[14] Locke, D., Andersen, B.S., Brosgol, B., Fulton, M., Henties, T., Hunt, J.J., Nielsen, J.O., Nilsen, K., Schoeberl, M., Vitek, J., and Wellings, A., 2013. Safety-Critical Java Technology Specification, Public draft, http://jcp.org/en/jsr/detail?id=302.

[15] Luu Anh, T., Man Chun, Z., and Quan Thanh, T., 2010. Modeling and Verification of Safety Critical Systems: A Case Study on Pacemaker. In *Secure Software Integration and Reliability Improvement (SSIRI), 2010 Fourth International Conference on*, 23-32,

[16] Macedo, H., Larsen, P., and Fitzgerald, J., 2008. Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System Using VDM. In *FM 2008: Formal Methods*, J. Cuellar, T. Maibaum and K. Sere Eds. Springer Berlin Heidelberg, 181-197.

[17] Malik, A., Salcic, Z., Roop, P.S., and Girault, A., 2010. SystemJ: A GALS language for system level design. *Computer Languages, Systems, & Structures 36*, 4, 317-344.

[18] Nadeem, M., Biglari-Abhari, M., and Salcic, Z., 2012. JOP-plus - A processor for efficient execution of java programs extended with GALS concurrency. In *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, 17-22,

[19] Park, H., Malik, A., and Salcic, Z., 2014. Time square - marriage of real-time and logical-time in GALS and synchronous languages. In *Proceedings of the The 20th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)* (Chongqing, China 2014). http://homepages.engineering.auckland.ac.nz/~amal029/RTCSA_final.pdf.

[20] Park, H., Malik, A., and Salcic, Z., 2014. WYPIWYE automation systems - an intelligent manufacturing system case study. In *Proceedings of the 19th International Conference on Emeting Technologies and Factory Automation (ETFA)* (Barcelona, Spain 2014). http://homepages.engineering.auckland.ac.nz/~amal029/ETFA_2014_final_authors.pdf.

[21] Roussel, J.-M. and Lesage, J.-J., Validation and verification of grafcets using state machine. In *Proceedings of IMACS-IEEE "CESA'96"*, pp. 758-764, http://hal.archives-ouvertes.fr/hal-00353188.

[22] Salcic, Z. and Malik, A., 2013. GALS-HMP: A heterogeneous multiprocessor for embedded applications. *ACM Trans. Embed. Comput. Syst. 12*, 1s, 1-26.

[23] Schoeberl, M., 2008. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture 54*, 1, 265-286.

[24] Singh, N.K., Wellings, A., and Cavalcanti, A., 2012. The cardiac pacemaker case study and its implementation in safety-critical Java and Ravenscar Ada. In *Proceedings of the the 10th International Workshop on Java Technologies for Real-time and Embedded Systems* (Copenhagen, Denmark 2012), ACM, 2388948, 62-71.

[25] Woodcock, J., 2006. First Steps in the Verified Software Grand Challenge. *Computer 39*, 10, 57-64.