# Programming Deterministic Reactive Systems with Synchronous Java

Christian Motika and Reinhard von Hanxleden and Mirko Heinold
Dept. of Computer Science
Christian-Albrechts-Unversität zu Kiel
24098 Kiel, Germany
Email: {cmot, rvh, mhei}@informatik.uni-kiel.de

***Abstract*—A key issue in the development of reliable embedded software is the proper handling of reactive control-flow, which typically involves concurrency. Java and its thread concept have only limited provisions for implementing deterministic concurrency. Thus, as has been observed in the past, it is challenging to develop concurrent Java programs without any deadlocks or race conditions.**

**To alleviate this situation, the Synchronous Java (SJ) approach presented here adopts the key concepts that have been established in the world of synchronous programming for handling reactive control-flow. Thus SJ not only provides deterministic concurrency, but also different variants of deterministic preemption. Furthermore SJ allows concurrent threads to communicate with Esterel-style signals. As a case study for an embedded system usage, we also report on how the SJ concepts have been ported to the ARM-based Lego Mindstorms NXT system.**

## I. Introduction

Embedded systems typically react to inputs with internal, state-based computations, followed by some output, as shown in Fig. 1. These computations often exploit concurrency, which can be implemented with Java threads. To prevent race conditions and deadlocks, Java provides synchronization primitives like semaphores and also higher level mechanisms like monitors. The synchronize keyword in a Java class introduces this concept implicitly. However, using these techniques, it is difficult to specify deterministic concurrent behavior without introducing non-determinism, as further discussed by Lee [13].

*Contributions:* We here present Synchronous Java (SJ), an approach that allows to directly embed deterministic reactive control-flow in Java, which encompasses concurrency and preemption. We side-step the traditional Java thread concept and its dependence on a—from an application point of view—unpredictable scheduler. Instead, SJ implements a light-weight application-level thread concept that combines coroutines with the synchronous model of computation (MoC). A case study shows how SJ can be used for solving common concurrent problems on reactive embedded targets.

*Outline:* In the next section, we discuss related work. Section III follows with a presentation of deterministic concurrency in SJ. Section IV illustrates the usage of SJ signals and preemption. Section V discusses some implementation aspects, including deployment on an embedded example platform. Section VI evaluates experimental results comparing SJ with traditional Java threads. We conclude in Section VII and give some outlook on future work.

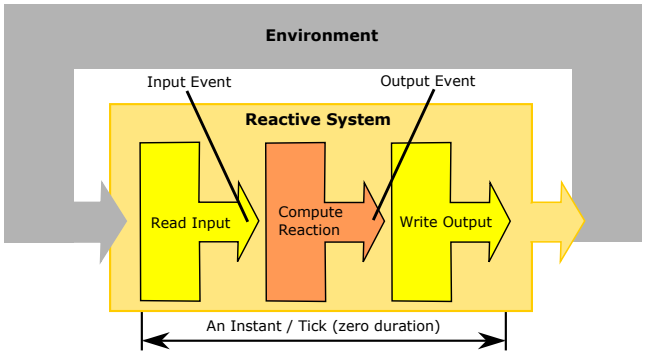


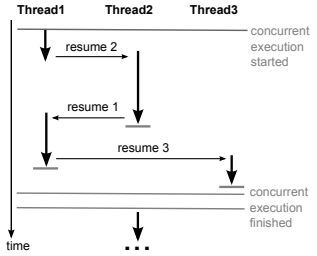

Fig. 1. Cyclic, discretized execution of a reactive embedded system.

## II. Related Work

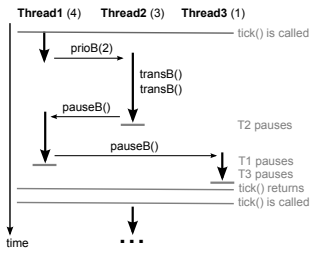
Nilsen [17] presented early ideas to use Java in embedded real time systems. The proposed extensions allow to analyze and measure timing and memory requirements of system activities and to specify a protocol how to add activities to a *real time executive* that is managing *resource budgets*. As a Real Time Java environment, Miyoshi [15] implemented prototype threads with special synchronization mechanisms as an extension package with minimal changes to the original Java Virtual Machine (JVM). Plsek et al. [18] also modified the JVM. These approaches cannot utilize the advantage of platform independence of the Java language, unlike SJ, which is itself implemented in Java and hence platform independent.

To gain predictable Java applications there is another category of solutions, e. g., by Schoeberl [21], which do not modify or specialize the JVM but supply specialized hardware that is able to execute Java Byte Code (JBC) natively. The Java Optimized Processor (JOP) [19] and the Reactive Java Optimized Processor (RJOP) [16] are both such hardware-based approaches. These could perfectly be combined with SJ, which addresses programming and scheduling issues.

The Real-Time Specification for Java [5] tightens Java w.r.t thread scheduling and synchronization allowing programs to run without interference from garbage collection so that timing constraints are provable. Safety Critical Java [11] is a standard facilitating programs capable of certification under standards such as DO-178B. It introduces *missions* as bounded sets of periodic reactive jobs. Schoeberl [20] extends this by *mission modes* as coarse application building blocks. These cover different modes of operation during runtime of real-time




(a) Classic coroutine style: threads resume each other directly




(b) SJ coroutines: application-controlled priorities (shown in parentheses) determine thread scheduling

Fig. 2. Comparison of coroutine concepts.

```
1  public class MySJProg extends SJProgram<StateLabel> {
2    enum StateLabel {STATE0, STATE1}
3
4    public MySJProg() {
5      super(STATE0, 1);     // Start at STATE0 with priority 1
6
7    public final void tick () {
8      while (!isTickDone()) {
9        switch (state()) {
10         case STATE0:
11           // ... some code ...
12           break;
13         case STATE1:
14           // ... some code ...
15           break;
16       }
17     }
18   }
19 }
```

Fig. 3. Structure of an SJ program.

applications. SJ could be utilized for implementing missions, in particular SJ makes most sense for single mission applications with a fixed number of threads.

One problem of Java threads is their performance depending on the actual implementation [23]. Another problem of Java threads is that the scheduler may interleave threads at arbitrary points during execution. The idea of coroutines [8] is to let threads cooperate, with themselves in charge of passing on control, instead of using a scheduler. Fig. 2a shows an example schedule of an execution with three coroutine threads. Thread1 resumes Thread2 at some specific and well-defined point during its execution. After Thread2 has finished its work completely, it resumes Thread1 again. After finishing its work, Thread1 gives control to Thread3. For implementing a coroutine scheduling in Java, there exist various possibilities. Using Java threads for doing this is cumbersome because it is not light-weight. JBC manipulation is a very low level addressing of this problem. Such solutions are restricted to fully-compliant JVM stacks, e. g., this will not work on Android. There are solutions to build a patched JVM for supporting coroutines more natively, e. g., the Da Vinci Machine [22]. There are other attempts to implement coroutines using Java Native Interface (JNI), loosing Java's platform independence. SJ tackles the coroutines-like scheduling problem in true Java by exploiting the switch-case statement combined with Java reflection. We also implemented an embedded variant of SJ that does not even use Java reflection. The advantage is a light-weight and platform-independent implementation. In addition, unlike the aforementioned approaches, SJ offers deterministic preemption.

Synchronous languages like Esterel [4] or Lustre [7] address concurrency and preemption in a precisely predictable and semantically well-founded way. The execution scheme follows the reactive model illustrated in Fig. 1. Physical time is divided into multiple discrete *ticks*. The reaction is conceptually considered to be atomic and to take no time, i. e., practically to be *fast enough* according to timing requirements that stem from the physics of the environment. The semantics prescribes the execution order of concurrent threads, which not only entails determinism, but also timing predictability [3]. Reactive C [9] is an extension of C. Inspired by Esterel, it employs the concepts of ticks and preemptions, but does not provide true concurrency. FairThreads [6] are an extension introducing concurrency via native threads. SJ does not use Java threads, but does its own, light-weight thread book keeping.

SJ has been largely inspired by Synchronous C (SC), also known as SyncCharts in C [25], which introduces deterministic and light-weight threads for the C language. Sec. IV later compares an SJ example with its SC counterpart. Köser [12] investigates the SC approach for modern multi-core computer architectures. SC, like SJ, can be used to implement the recently proposed *sequentially constructive* MoC, which loosens some restrictions the classical synchronous MoC by taking advantage of the sequential nature of C/Java-like languages [26]. The sequentially constructive MoC also provides an approach to automatically compute the priorities employed by SC and SJ. Precision Timed C (PRET-C) [1] similarly to SC enriches the C programming language inspired by synchronous languages, but is restricted to static execution orders among threads.

## III. Deterministic Concurrency in SJ

We now discuss the overall structure of SJ programs and how it provides deterministic, synchronous-style concurrency. Sec. IV then describes preemption and signal handling in SJ.

### A. SJ program structure

SJ is an extension to Java that is written in pure Java itself. Fig. 3 illustrates the basic structure of an SJ program. An SJ program extends the abstract class SJProgram which provides the *SJ operators*, of which Table I lists the most relevant ones discussed in the remainder of this paper.

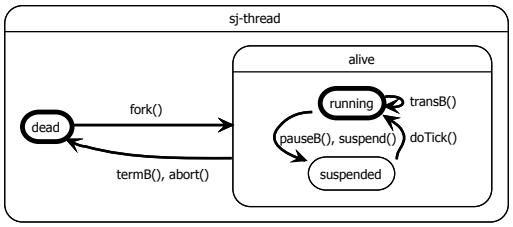
The enumeration StateLabel (line 2) defines a finite set of *states* that this program or system can be in. These states correspond to locations in the program, which in SJ are expressed as different cases in a switch statement; if Java had a goto statement, these states could simply be statement labels. Each SJ thread maintains a *coarse program counter* that corresponds to a particular state, or *continuation*. The constructor specifies the initial state of the *main thread* (see line 5), together with its *priority*. The main thread can create additional threads with the fork() operator.

The tick() method (lines 7–18) defines the behavior of the program for one tick. The while loop ensures that the computation of the complete reaction (tick), which may consist of several computational steps, is run until isTickDone() returns true, which indicates that all threads have finished the current

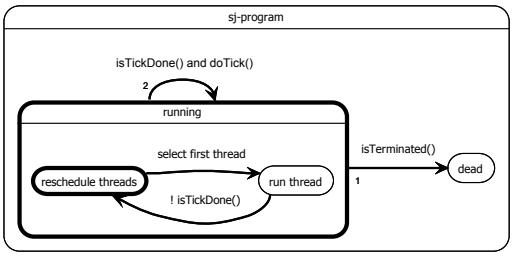
| SJ Operator | Explanation |
|---|---|
| *Thread management* | |
| fork($l$, $p$) | Fork a new descendant thread at label $l$ with priority $p$. A sequence of fork() operators must be terminated with a forkEndB(). |
| forkEndB($l$)* | Continue the current thread at label $l$ with the same priority. |
| joinDoneCB()* | Return true iff all descendant threads have terminated. |
| prioB($l$, $p$)* | Change the priority of the running thread to $p$, continue at label $l$. |
| *Pausing/terminating* | |
| pauseB($l$)* | Suspend execution for the current tick, continue in the next tick at label $l$. |
| termB()* | Terminate thread. |
| *Further control flow* | |
| gotoB($l$) | Jump to label $l$. |
| abort() | Recursively abort all descendants created by the current thread. |
| suspend() | Recursively suspend all descendants created by the current thread for the current tick. |
| transB($l$) | Shorthand for abort() and gotoB($l$). |
| *Signals* | |
| Signal $s$ | Initialize a pure signal $s$. |
| $s$.emit() | Emit a pure signal $s$. |
| $s$.isPresent() | Return true iff signal $s$ is present. |
| awaitDoneCB($s$) | First return false and pause, then return true iff signal $s$ has become present. |
| $s$.pre() | Returns the instance of signal $s$ at previous tick. |
| *Valued signals* | |
| ValuedSignal $v$=new ValuedSignal("$v$", MULTIPLY) | Initialize a valued integer signal $v$ combined with multiplication. |
| $v$.emit($val$) | Emit a valued integer signal $v$ with value $val$. |
| $v$.getValue() | Returns the value of valued signal $v$. |

TABLE I. SJ OPERATOR OVERVIEW. YIELDING OPERATORS ARE MARKED WITH AN ASTERISK (*).

tick. During each iteration of the while loop, the state() method call invokes a priority-based scheduler that returns the current state of the thread to be executed next, which is then used in the switch statement. The coroutine-like cooperative scheduling is realized by reaching a break that terminates the current case of the switch statement and leads to the next scheduler call. Therefore, the SJ operators that upon their completion require a scheduler call must always be followed by a break statement, hence we call these also *breaking* operators. An example is gotoB(), where a thread changes its state. There are also *conditionally breaking* predicates that must be followed by a break statement when they return false. An example is joinDoneCB(), which returns true iff all descendent threads have terminated; otherwise, the calling thread blocks and hence must break. As an aid to the programmer, (conditionally) breaking operators are appended with a (C)B. Some of the breaking operators are also *yielding*, marked with an asterisk in Tab. I. After completion of a yielding operator, another thread may become eligible for execution, for example, because a thread has finished its current tick and therefore calls the pauseB() operator. An example of a non-yielding operator that nonetheless calls the scheduler is gotoB(), which merely changes the coarse program counter of a thread that then immediately continues at the new state.




(a) Life cycle of an individual SJ thread




(b) Life cycle of a complete SJ program

Fig. 4. State diagrams for the reactive life cycle of an SJ program and its individual threads. Initial states have a bold outline.

An SJ program also contains a main() method, not shown in Fig. 3, which calls the tick() method whenever a reaction should take place. More precisely, it calls the doTick() wrapper that resets outputs and samples inputs before calling tick(). This is illustrated later in the example shown in Fig. 5b.

### B. SJ Cooperative Threads

Fig. 4a illustrates the life cycle of a thread. It can either be *dead* or *alive*. The main thread is alive by default, while other concurrent or child threads are initially dead. When being forked, a thread becomes alive. Alive threads can act as normal Java programs and execute code that has been specified for this thread within the aforementioned tick() method. This can be Java code mixed with SJ operators. Alive threads that still have work to do in the current tick are *running*, threads that are still alive but done for the current tick are *suspended*. Some SJ operators leave a thread running, such as transB(). Other operators, notably pauseB() and suspend(), leave the thread alive, but suspend it for the remainder of the tick. At the end of their work, threads usually terminate (termB()) or are aborted by a (transitive) parent thread with abort(). SJ allows for building trees of threads for specifying hierarchical relations and make preemptions possible. SJ keeps track of these relations and maintains the book keeping.

A *running program* repeatedly calls the doTick() method to perform the program reactions, see also Fig. 4b. Within a tick, the scheduler keeps selecting a thread from a queue of running threads. When this thread breaks (yields) and isTickDone() is false, the next thread is selected for continuing execution. If isTickDone() is true, the doTick() method returns and the SJ program is waiting for the next call of the doTick() method. This continues until the isTerminated() method call in the main method indicates that the program becomes *dead*.

*1) Thread Priorities:* Threads are always associated with a unique priority. As already mentioned, the initial priority of the main thread is defined in the constructor of the SJ program. For other threads, their initial priority is specified as an argument

when creating them with the fork() operator. Threads can change their priority with prioB(). The aforementioned state() method (Sec. III-A) keeps track of all threads and their current priorities, and schedules from the currently running threads the one with the highest priority.

*2) Thread Scheduling:* SJ threads run concurrently and hand over control from one thread to another, in contrast to normal Java programs where control-flow is characterized by method invocations and method returns. This cooperative thread scheduling is inspired by coroutines [8], but in contrast to typical coroutines, in SJ it is not the yielding thread that has to specify which thread should resume. The yielding thread merely relinquishes control, by reaching a break statement. Then, the scheduler choses the thread to resume, via the state() method. As this choice is driven by the thread priorities, these application-controlled, typically static priorities are crucial for ordering accesses to shared data within a tick. E. g., we can enforce a writers-before-readers discipline, which is commonly part of the synchronous MoC, by giving threads that write to a particular variable a higher priority than threads that read from that variable. Note, however, that even if we do not require strict writers-before-readers, the SJ program is still deterministic, as determinism is already implied by the underlying sequential nature of the tick() function that does not use the Java scheduler. This is exploited, e. g., in the sequentially constructive MoC [26].

Fig. 2b shows an example schedule of three threads. Thread1 starts the control because it has the highest priority of 4 when tick() is called. Thread1 executes some code. It then lowers its priority to 2 by calling prioB(2). After this priority change, Thread2 has the next highest priority of 3 and is selected by the state() method for continuation. In the same synchronous tick, Thread2 then executes some code including two transition changes with the transB() operator. This means that the coarse program counter maintained by SJ for Thread2 is changed for continuation to some other label, but this does not involve a thread re-scheduling, i. e., transB() is not yielding. After this, Thread2 calls pauseB() to indicate that it finished execution for this tick. state() now selects Thread1 again because it has the highest priority of 2 of all running threads. When Thread1 also calls pauseB() to indicate it has finished execution for this tick, finally, Thread3 with priority 1 is selected to run its code. When Thread3 calls pauseB(), no other thread needs to be scheduled for execution in this tick. Hence, the tick() method returns. The first thread to run in the next tick is again the one with the highest priority.

### C. The Producer-Consumer Example

The Producer-Consumer (PC) example in Fig. 5, inspired by the Producer-Consumer-Observer (PCO) example of Lickly et al. [14], is a small-scale application with two concurrent threads, a data producer and a data consumer. The threads jointly access some shared variable BUF, which is effectively a one-place buffer. This must be accessed in the usual fashion, where first the producer must write to BUF, then the consumer reads BUF, after which the producer may write again, and so forth.

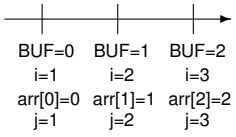
*1) Classical Java implementation:* In the program shown in Fig. 5a, the class PC creates the concurrent Producer and Consumer threads in its constructor. Both threads share a common

```
1  public class PC {
2    static final int TICKS = 100;
3    static Monitor monitor;
4
5    PC() {
6      PC.monitor = new Monitor();
7      new Thread(new Producer()).start();
8      new Thread(new Consumer()).start();
9    }
10
11   class Monitor {
12     boolean empty = true
13     int BUF;
14
15     synchronized void setBUF(int i) {
16       while (!empty) {
17         wait();
18       }
19       empty = false;
20       BUF = i;
21       notifyAll();
22     }
23
24     synchronized int getBUF() {
25       while (empty) {
26         wait();
27       }
28       empty = true;
29       int returnValue = BUF;
30       notifyAll();
31       return returnValue;
32     }
33   }
34
35   class Producer implements
36       Runnable {
37     void run() {
38       for (int i = 0;
39            i < TICKS;
40            i++) {
41         monitor.setBUF(i);
42       }
43     }
44   }
45
46   class Consumer implements
47       Runnable {
48     private int tmp;
49     private int[] arr = new int[8];
50
51     void run() {
52       for (int j = 0;
53            j < TICKS;
54            j++) {
55         tmp = monitor.getBUF();
56         arr[j % 8] = tmp;
57       }
58     }
59   }
60 }
```

(a) Implementation with standard Java threads

```
1  import sj.SJProgram;
2  import examples.PC.StateLabel;
3  import static examples.PC.StateLabel.*;
4
5  public class PC extends
6      SJProgram<StateLabel> {
7    enum StateLabel {
8      InitPC, Producer, Consumer }
9
10   static final int TICKS = 100;
11   private int BUF, i = 0, j = 0, tmp;
12   private int[] arr = new int[8];
13
14   public PC() {
15     super(InitPC, 2);
16   }
17
18   @Override
19   public void tick() {
20     while (!isTickDone()) {
21       switch (state()) {
22         case InitPC:          // Prio 2
23           fork(Consumer, 1);
24           forkEndB(Producer);
25           break;
26
27         case Producer:        // Prio 2
28           BUF = i;
29           i++;
30           pauseB(Producer);
31           break;
32
33         case Consumer:        // Prio 1
34           tmp = BUF;
35           arr[j % 8] = tmp;
36           j++;
37           pauseB(Consumer);
38           break;
39       }
40     }
41   }
42
43   public static void main() {
44     PC pc = new PC();
45     for (int t = 0;
46          t < PC.TICKS;
47          t++) {
48       pc.doTick();
49       if (pc.isTerminated())
50         break;
51     }
52   }
53 }
```

(b) Implementation with SJ threads




(c) A logical tick time line, illustrating some assignments of the first three ticks.

Fig. 5. The Producer-Consumer (PC) example.

Monitor buffer object. The Producer thread produces data in its run() method (lines 37ff), consumed by the Consumer thread in its run() method. There is no synchronization constraint explicitly specified, neither in the producer nor in the consumer thread, although the producer has to run before the consumer. All synchronization is expressed in the shared Monitor. It suspends threads trying to consume (getBUF()) data from an empty buffer and the ones trying to produce (setBUF()) data on a full (!empty) buffer. The constraint that the producer thread has to run before the consumer is realized only implicitly. With

notifyAll() all producer and consumer threads possibly waiting are awoken. These may wait() again afterwards immediately without doing anything (lines 17 and 26).

With this realization, scheduling has large influences on possible interleavings and the actual execution order that is totally unpredictable. Hence, execution time is also hard to predict. The situation becomes worse if one wants to add an additional observer thread like in the original example [14]. If the observer does not consume data but needs to run after the producer and before the consumer, this also has to be expressed in the Monitor class specifying the shared buffer. Overhead of poorly scheduled executions with unnecessary awoken threads will consequently grow. A related problem is the creation and killing of threads for simple tasks, which is also inefficient. An alternative is to re-use threads of a thread pool, which is more efficient but uses more system resources.

To summarize, the classical Java approach to concurrency suffers from the inability to explicitly specify scheduling constraints that are required for determinism. Such constraints are expressed only implicitly using coordination data structures like monitors. The scheduling constraints cannot be expressed in the producer or the consumer activities directly. Moreover the solution with Java threads has the overhead of potentially many additional but superfluous context switches between threads.

*2) The PC Example with SJ:* SJ allows for light-weight threads and more explicit control over scheduling. Consider Fig. 5b where the PC example is listed in Java using SJ constructs, according to the structure already discussed in Sec. III-A. Following the synchronous approach, the program behavior is broken up into discrete reactions, or ticks, which in this case correspond to one production/consumption cycle. The tick() method (lines 19ff), repeatedly invoked via doTick() in main() (lines 43ff), computes one tick.

Within the tick() method, the concurrent behavior of the program is specified by the switch statement and its different states (cases). The main thread starts at state InitPC with priority 2, as specified by the PC constructor (line 14ff). The fork() operator in line 23 creates a consumer thread with initial state Consumer and priority 1. The main thread subsequently assumes the role of the producer thread, and forkEndB() defines the next state of this thread to be Producer. The priority of that thread remains 2. Whenever threads are forked, one should give the scheduler again a chance to run, as one might possibly have created new threads with higher priorities than the already existing threads. Thus forkEndB() is a yielding operator that should be followed by a break statement (line 25), although here, this is not strictly necessary as the currently running thread could just fall through to the next case.

In the next iteration of the enclosing while loop, the scheduler run by the state() method selects the running thread with the highest priority, which in this case is 2, corresponding to the producer thread that resumes at Producer. This thread writes to BUF, increments i, and declares that it is done for the tick with pauseB(), specifying Producer as continuation point when starting this thread in the next tick. Next, the scheduler selects the consumer thread with priority 1, which does its computations until it pauses as well. Now isTickDone() returns true and tick()/doTick() returns to the main() method. However, both threads are still alive; in fact, they never terminate in this example. Therefore, pc.isTerminated() returns false, and doTick() is invoked again, until PC.TICKS ticks have been executed.

The behavior of the SJ program is also illustrated in the *logical tick time line* in Fig. 5c, which highlights some variable assignments taking place within the first three ticks.

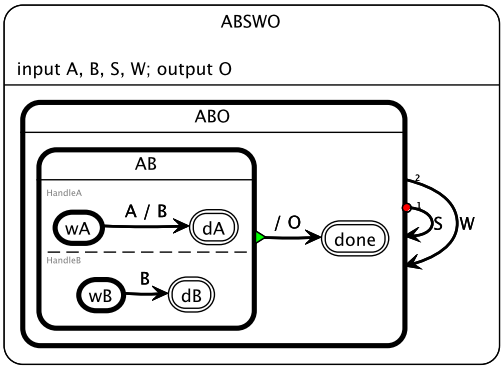
To summarize, the SJ program expresses deterministic concurrent control flow directly at the application level, without any need to invoke the Java scheduler. In every tick, the producer and the consumer run in lock-step and the producer always runs before the consumer. Threads are coordinated with explicit, user-controlled priorities, which provide the basis for a deterministic scheduling regime. Threads require minimal bookkeeping, they just have to keep track of a priority and execution state, and hence context switches are very light-weight.

## IV. Preemption and Signals

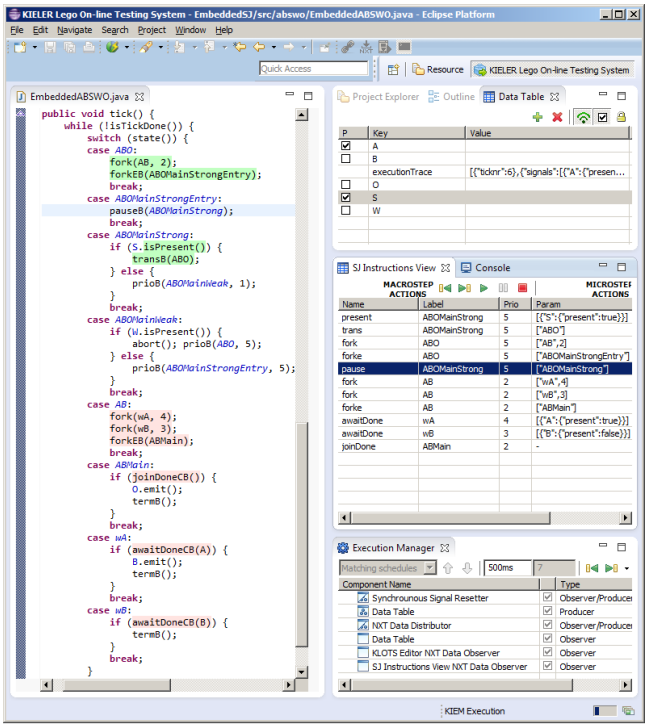
After discussing the core SJ concepts for handling concurrency in the previous section, we now cover further control-flow constructs, notably a set of preemption-related operators, and the capability to communicate among threads with synchronous-style signals.

In synchronous languages, a *signal* is defined by its *presence status* (*present* or *absent*) within a tick. Within a tick, a signal is absent by default, unless it gets *emitted* and is hence present. There are also *valued signals* which may be associated with a unique value, which is set during signal emission. The present status of interface signals are set by the environment before each tick. A signal can for example be used to trigger a preemption, which is illustrated in the ABSWO example shown in Fig. 6. To familiarize ourselves with ABSWO, we first have a look at the SyncChart version shown in Fig. 6a, which is a graphical means to precisely describe concurrent and preemptive behavior. We here describe SyncCharts to the extent needed to explain ABSWO, a full description of SyncCharts is given by André [2]. In SyncCharts, transitions are triggered by the presence of a specified signal, and in turn taking a transition can make a signal present. E. g., initially the reactive system is in states wA and wB, as well as in the enclosing states AB and ABO. However, when signal A becomes present in some tick, a transition wA to dA takes place in the same tick and emits B.

To illustrate preemption, ABSWO has two different preemptive self transitions from state ABO to ABO. The transition triggered by presence of S is a *strong preemption* (indicated by a red circle at the transition source), meaning that in each tick the transition's trigger (signal S) is tested *before* the behavior of the source state (ABO) gets executed. This implies that if S becomes present in a tick, then O cannot be emitted anymore in that tick. Conversely, the transition triggered by W is a *weak transition*, meaning that this transition is tested *after* ABO has been executed. If a state has multiple outgoing transitions, as is the case for ABO, then these transitions are statically ordered by a *transition priority*, indicated by numeric tail labels. Strong preemptions must be tested before weak transitions, therefore the transition triggered by S has priority 1 and the other transition has priority 2. Another transition type is the *normal termination* (indicated by a green triangle at




(a) Graphical SyncChart



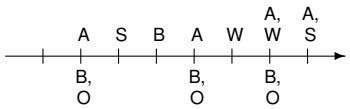

(b) KIELER view of SJ program running on a Lego Mindstorms NXT

```
1  public final void tick () {
2    while (!isTickDone()) {
3      switch (state()) {
4      case ABO:              // Prio 5
5        fork(AB, 2);
6        forkEndB(ABOMainStrongEntry);
7        break;
8      case ABOMainStrongEntry: // Prio 5
9        pauseB(ABOMainStrong);
10       break;
11     case ABOMainStrong: // Prio 5
12       if (S.isPresent()) {
13         transB(ABO);
14       } else
15         prioB(ABOMainWeak, 1);
16       break;
17     case ABOMainWeak: // Prio 1
18       if (W.isPresent()) {
19         abort();
20         prioB(ABO, 5);
21       } else
22         prioB(ABOMainStrongEntry, 5);
23       break;
24     case AB:               // Prio 2
25       fork(wA, 4);
26       fork(wB, 3);
27       forkEndB(ABMain);
28       break;
29     case ABMain:           // Prio 2
30       if (joinDoneCB()) {
31         O.emit();
32         termB();
33       };
34       break;
35     case wA:               // Prio 4
36       if (awaitDoneCB(A)) {
37         B.emit();
38         termB();
39       };
40       break;
41     case wB:               // Prio 3
42       if (awaitDoneCB(B)) {
43         termB();
44       };
45       break;
46   }
47 }
48 }
```

(c) SJ tick() method

```
1  int tick () {
2    MainThread (ABO, 5) { // Prio 5/1
3      FORK1(AB, 2);
4      while(1) {
5        PAUSE();
6        if (PRESENT(S))
7          TRANS(ABO);
8        PRIO(1);
9        if (PRESENT(W)) {
10         PRIO(5);
11         TRANS(ABO);
12       }
13       PRIO(5);
14     }
15   }
16
17   Thread (AB) {    // Prio 2
18     FORK2(wA,4, wB,3);
19     JOIN();
20     EMIT(O);
21   }
22
23   Thread (wA) {    // Prio 4
24     AWAIT(A);
25     EMIT(B);
26   }
27
28   Thread (wB) {    // Prio 3
29     AWAIT(B);
30   }
31
32   TICKEND();
33 }
```

(d) SC tick() method




(e) A logical tick time line that illustrates an execution trace consisting of eight discrete ticks. Each tick is annotated with the input signals from the environment (above) and corresponding output signals (below).

Fig. 6. ABSWO example in SJ and SC, illustrating preemption and the usage of signals. ABSWO concurrently waits for the signals A and B. If both have occurred, it emits output signal O. Note that input signal A is sufficient because signal B is emitted once signal A occurred. The behavior of ABO is reset strongly by signal S and weakly by signal W.

the transition source), which takes place when all concurrent *regions* within the transition source have terminated, i. e., have entered a *final state* (indicated by double outline). In ABSWO, state AB contains regions HandleA and HandleB, which in turn contain final states dA and dB, respectively. When both of these final states have been entered in a tick, the normal termination transition from AB to done is taken in that same tick. Transition triggers are per default *non-immediate*, meaning that they are always disabled in the tick when their source state is entered. In ABSWO, this prevents an instantaneous loop to be induced by the self transitions on ABO. It also prevents the transitions originating in wA and wB to be taken in a tick immediately after just entering ABO in that tick.

A possible execution trace is shown in the tick time line in Fig. 6e. No signals are present in the initial tick; in the second tick, the environment makes A present, which triggers in turn presence of B and O; in the third tick, the behavior is reset by S; and so on.

Fig. 6c shows the SJ tick() function which precisely corresponds to the SyncChart of Fig. 6a. The constructor (not shown here) starts the main thread at state ABO with priority 5. This forks off another thread at AB and continues at ABOMainStrongEntry, which then pauses for a tick. This pause corresponds to the non-immediate nature of the self-transitions on ABO. From the next tick on, the main thread first, running at priority 5, tests S and possibly takes a self-transition to ABO; if this transition is not taken, it then, at priority 1, tests W and possibly self-transitions; otherwise it again raises its priority again to 5, transfers control to ABOMainStrongEntry, and pauses. Concurrently, the thread started at AB first forks two threads wA and wB (lines 25 and 26) and then, in state ABMain, waits for them to terminate with joinDoneCB() and

then emits O. AB runs at priority 2, so it is executed after strong preemption on ABO (triggered by S) is tested, but before the weak preemption is tested. Also concurrently, the thread starting at wA tests whether A is present, and if so, emits B and terminates. The thread starting at wB similarly tests B and possibly terminates. These two threads run at priorities 4 and 3, respectively, thus when A is present, B will be emitted *before* it gets tested.

To summarize, SJ provides variants of deterministic preemption that allow the modeler to choose explicitly whether the preemption should prevent a preempted component to still execute the current tick or not. This is clearly preferable over most other typical implementations of preemption, where this choice is up to (unpredictable) scheduling decisions. The ABSWO example illustrates how signals can be used to trigger preemptions, but of course any Boolean expressions in standard Java can be used as preemption triggers as well. The ABSWO example also illustrates again how priorities can be used to statically control thread scheduling, which in this case allows to distinguish strong and weak preemptions, and to assure that any emissions of some signal take place before that signal gets tested. Traulsen et al. [24] further discuss how such transition priorities to implement SyncCharts can be synthesized automatically.

For a brief comparison between Synchronous Java and Synchronous C, Fig. 6d lists the equivalent Synchronous C (SC) program. The principles are exactly the same. However, as C/gcc have some capabilities that Java does not have, notably computed gotos and a powerful preprocessor, the C variant allows to hide most of the low-level control logic in SC macros. E. g., the AWAIT() macro automatically generates a continuation label, hidden to the user, based on the source code line number. Conversely, some SC macros are just a structuring aid without much functionality. E. g., the Thread($l$) macro simply terminates the preceding thread and generates a label $l$, nothing else is done at run time.

## V. Implementation Notes

An interesting part of SJ behind the scenes is the method isTickDone(). It returns true iff the current tick is done, i. e., when the internal queue of running threads is finally empty. At the beginning of a tick, all running threads are added to this queue ordered by their priority. If a thread calls prioB(), its position in the priority queue is re-arranged. A thread is removed from this queue when it calls pauseB().

Another central method is the state() method that implements the SJ dispatcher and does the actual scheduling. It returns the next thread state label for the switch-case statement to continue execution. This is the next label from the top of the ordered priority queue. Forking and terminating threads as well as the handling of signals requires additional internal book keeping.

A more detailed discussion of the implementation is given by Heinold [10]. SJ is implemented as a part of the KIELER[1] modeling framework. The SJ source code and the documentation is freely available under the Eclipse Public License (EPL) at the KIELER website.

For validation, we brought an embedded variant of SJ onto the ARM-based Lego Mindstorms Next Lego Computing Brick (NXT) device[2]. A debugging facility inside the KIELER platform offers the possibility to debug SJ programs running on the NXT device step-by-step. Fig. 6b shows a setup where the ABSWO example is running on the NXT and is debugged within the KIELER RCP. In the current macro tick the input signal A was set to be present in the upper Data Table Eclipse View, which serves as a user input facility. Running on the embedded device, the SJ ABRO program on the left reacted to this input as the termB() operation near the wA label is executed because the awaitDoneCB(A) operation finished its execution. All taken micro steps can be observed in the SJ Instructions View. A micro step consists of an SJ primitive, possibly with following Java code. For a selected micro step, already executed code is marked green in the editor and not yet executed code is marked red. Because the input signal B was not set to be present yet and hence the second wB thread has not yet terminated, the joinDoneCB() predicate is not yet true and the guarded code lines for emitting output signal O are not executed in this current macro tick.

## VI. Experimental Results

To illustrate the predictability and the efficiency of the SJ approach compared to Java threads, we compared the run times of the Java threads version and the SJ version of the PC example discussed in Section III. We ran both programs on an Intel Core 2 Duo P8700 @ 2.53 Ghz machine with 4GB of RAM and a 64 Bit JVM with a variable number of ticks, i. e. TICKS. Fig. 7 shows the execution time of each implementation over the variable number of ticks. For getting reasonable results, we made three experiments for each number of ticks and took the worst execution time. We considered tick numbers between 0 and 10.000 in linear steps of 1000. The results also show the speed-up.

The SJ version is faster (average of 1.75 times faster) compared to the Java thread version that has to struggle with more overhead due to possibly poorly scheduled executions. Another, perhaps more important difference is the variability of the worst-case run time. While the Java thread version is heavily unpredictable especially when it comes to more duty, i. e., more ticks, the SJ variant is much closer to a linear growth and hence more predictable. Both facts support our thesis that the SJ implementation is more light-weight and much more predictable.

## VII. Conclusion and Outlook

Properly synchronizing Java threads may become complex and problematic. We presented SJ as an adoption of the synchronous concepts for Java, and showed that SJ can help specifying concurrent threads in a light-weight and robust way. We also illustrated the use of preemption and predictable synchronous signal communication between concurrent threads of an SJ program. Another benefit is that such programs can run on platforms where a thread management may be too much overhead, e. g., like on embedded JVMs. As a case-study, we presented an embedded variant of SJ running on Lego Mindstorms.

[1] http://www.informatik.uni-kiel.de/rtsys/kieler

[2] http://mindstorms.lego.com

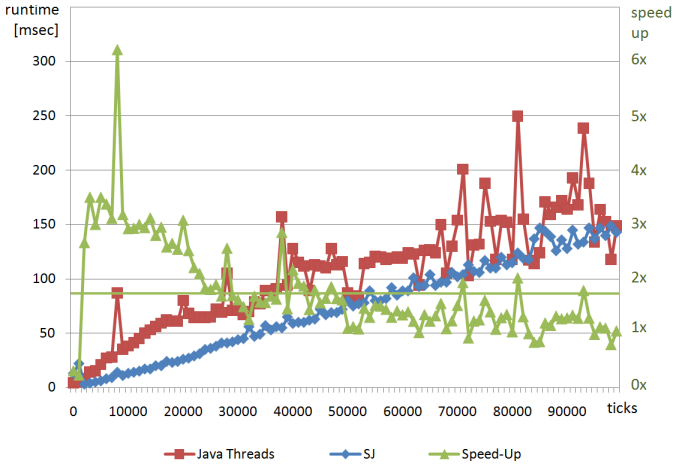



Fig. 7. Worst-case run times, SJ vs. standard Java threads, for the PC example.

In addition to providing deterministic reactive control flow, our experimental results indicate that SJ programs have a more predictable run time and are typically faster than Java threads. SJ can be considered a programming language as well as a target language for code generation from more abstract models, such as SyncCharts. SJ code is close to abstract specifications, as it directly supports concepts like states and transitions. SJ permits to implement synchronous data-flow applications, see the PC example, as well as control-driven applications, see ABSWO.

We plan to exploit SJ as an automated code generation target from SyncCharts and Esterel, possibly also Lustre, and to integrate and evaluate this in the context of KIELER. We further intend to enhance the development process of concurrent and preemptive SJ code with visual and interactive debugging possibilities. We also plan to introduce an intermediate format for the common part of SJ and SC and to validate SJ and SC simulators by leveraging the Ptolemy[3] Project of the UC Berkeley.

## REFERENCES


[1] S. Andalam, P. S. Roop, and A. Girault. Deterministic, predictable and light-weight multithreading using pret-c. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'10)*, pages 1653–1656, Dresden, Germany, 2010.

[2] C. André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996.

[3] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. von Hanxleden, R. Wilhelm, and W. Yi. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems*, 2013. Accepted.

[4] G. Berry. The foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[5] G. Bollella, J. Gosling, B. M. Brosgol, and P. Dibble. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[6] F. Boussinot. Fairthreads: mixing cooperative and preemptive threads in C. *Concurrency and Computation: Practice and Experience*, 18(5):445–469, Apr. 2006.


[3] http://www.ptolemy.org


[7] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'87)*, pages 178–188, Munich, Germany, 1987. ACM.

[8] M. E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[9] Frederic Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, 1991.

[10] M. Heinold. Synchronous Java, Sept. 2010. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science.

[11] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, Mar. 2009.

[12] N. Köser. SyncCharts in C auf Multicore, Oct. 2010. Dimploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science.

[13] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[14] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of Compilers, Architectures, and Synthesis of Embedded Systems (CASES'08)*, Atlanta, GA, USA, Oct. 2008.

[15] A. Miyoshi, T. Kitayama, and H. Tokuda. Implementation and evaluation of real-time Java threads. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, pages 166–175, San Francisco, CA, USA, Dec. 1997.

[16] M. Nadeem, M. Biglari-Abhari, and Z. Salcic. RJOP: a customized Java processor for reactive embedded systems. In *Proceedings of the 48th Design Automation Conference (DAC'11)*, pages 1038–1043, New York, NY, USA, 2011. ACM.

[17] K. Nilsen. Adding real-time capabilities to Java. *Commun. ACM*, 41(6):49–56, June 1998.

[18] A. Plsek, L. Zhao, V. H. Sahin, D. Tang, T. Kalibera, and J. Vitek. Developing safety critical Java applications with oSCJ/L0. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES'10)*, pages 95–101, Prague, Czech Republic, 2010. ACM.

[19] M. Schoeberl. A time predictable Java processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'06)*, pages 800–805, Munich, Germany, Mar. 2006.

[20] M. Schoeberl. Mission modes for safety critical java. In *Proceedings of the 5th IFIP WG 10.2 International Conference on Software Technologies for Embedded and Ubiquitous Systems (SEUS'07)*, pages 105–113, Santorini Island, Greece, 2007. Springer-Verlag.

[21] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture (JSA)*, 54(1–2):265–286, 2008.

[22] L. Stadler, T. Würthinger, and C. Wimmer. Efficient coroutines for the Java platform. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ'10)*, pages 20–28, Vienna, Austria, 2010. ACM.

[23] M. Sung, S. Kim, S. Park, N. Chang, and H. Shin. Comparative performance evaluation of java threads for embedded applications: Linux thread vs. green thread. *Inf. Process. Lett.*, 84(4):221–225, Nov. 2002.

[24] C. Traulsen, T. Amende, and R. von Hanxleden. Compiling SyncCharts to Synchronous C. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'11)*, pages 563–566, Grenoble, France, Mar. 2011. IEEE.

[25] R. von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proceedings of the International Conference on Embedded Software (EMSOFT'09)*, pages 225–234, Grenoble, France, Oct. 2009. ACM.

[26] R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, and O. O'Brien. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'13)*, Grenoble, France, Mar. 2013. IEEE.