INSTITUT FÜR INFORMATIK

Synchronous Java: Light-Weight, Deterministic Concurrency and Preemption in Java

Christian Motika, Reinhard von Hanxleden,
Mirko Heinold

Bericht Nr. 1213
October 2012

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Synchronous Java: Light-Weight, Deterministic Concurrency and Preemption in Java

Christian Motika, Reinhard von Hanxleden,
Mirko Heinold

Bericht Nr. 1213
October 2012

e-mail:
cmot@informatik.uni-kiel.de,
rvh@informatik.uni-kiel.de,
mhei@informatik.uni-kiel.de

Technical Report

Contents

List of Figures

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 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 applied in the context of Lego Mindstorms.

Key words: Synchronous, Java, Eclipse, SC, Coroutines, Scheduling, Determinism

1 Introduction

Embedded systems typically react to inputs with internal, state-based computations, followed by some output, as shown in Figure 1.1. These computations often exploit concurrency, which can be implemented with native Java threads. To prevent race conditions and deadlocks, Java provides synchronization primitives like semaphores and also higher level mechanisms like monitors. The `synchronize` keyword for a method 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 [11]. An alternative approach has been introduced by the synchronous language family.

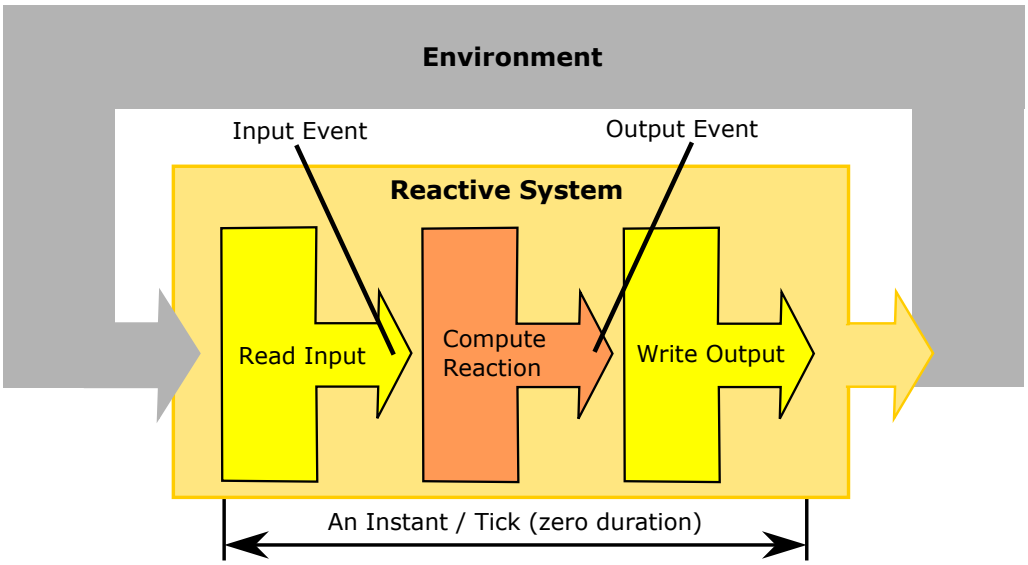


Figure 1.1: Synchronous view on the cyclic and continuous execution of a reactive system. The reaction is conceptually considered to be atomic and to take no time, i. e., practically to be *fast enough* according to timing requirements that stem from the physics of the controlled environment.

Synchronous Languages Synchronous languages like Esterel [4] or Lustre [6] address the problem of dealing with concurrency and preemption in a precisely predictable and semantically well-founded way. Synchronous languages can be used to express control-flow and data-flow in an abstract manner.

The synchronous execution scheme follows Figure 1.1, where output signals are considered to be instantaneously computed, i. e., in the same *tick*, from input signals coming

from the environment. Physical time is divided into multiple discrete ticks (see also Figure 5.1b). By construction, there cannot be any race conditions. Every part of a reaction computation happens at the same tick and is considered to take place at the same time (instantaneously with zero duration). This requires a well-defined computation order that is statically taken care of by the compiler of the synchronous language.

Hierarchy typically is used to specify preemptive behavior. Within a tick, synchronous languages allow to specify concurrent behavior with signals for communication purposes. For every tick, a signal has a defined *present status* declaring whether it is absent or present. It can not be both at the same time. The present status of all input signals must be set by the environment. Output and local signals are computed in a reaction and by default are absent unless they are *emitted* in the reaction computation of a tick.

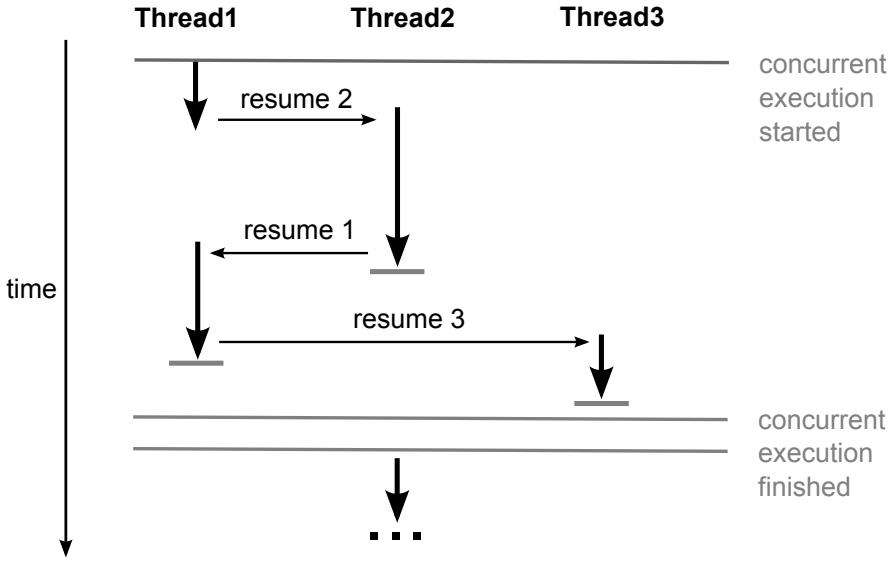


Figure 1.2: Cooperative coroutine threads: No scheduler is required and threads explicitly resume each other at specific synchronization points.

Coroutines A problem of Java threads is that the scheduler may interleave threads at arbitrary points during execution. The idea of coroutines [7] is to let threads cooperate, with themselves in charge of passing on control, instead of using a scheduler. After a coroutine thread has finished a chunk of its execution, it can explicitly resume another coroutine thread and may be later resumed itself by a third one.

Figure 1.2 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. Java threads come with overhead of locking, context switching, and kernel scheduling. In contrast, coroutines are supposed to be light and to execute fast as a very flexible and alternative way of dealing with concurrency. However, compared to Java threads they come with less synchronization possibilities.

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 is a variant of coroutines. We further introduce other synchronous concepts like signals to support deterministic communication between concurrent threads and preemption of hierarchical threads. 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 3 gives a first overview of SJ primitives. Section 4 follows with a more in-depth discussion of deterministic concurrency in SJ. Section 5 illustrates the usage of SJ signals and preemption. Section 6 gives implementation details. In Section 7 we present a Lego Mindstorms case study. Section 8 evaluates experimental results comparing SJ with traditional Java threads. Finally, we conclude in Section 9 and give some outlook on future work.

2 Related Work

Nilsson [15] presented early ideas to use Java in embedded real time systems. The proposed extensions allowed 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 [13] implemented prototype threads with special synchronization mechanisms as an extension package with minimal changes to the original Java Virtual Machine (JVM). Plsek et al. [16] 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 [18], 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) [17] and the Reactive Java Optimized Processor (RJOP) [14] are both such hardware-based approaches. These could perfectly be combined with SJ, which addresses programming and scheduling issues.

Synchronous C (SC) [20] introduces deterministic and light-weight threads for the C language. Synchronization and scheduling are based on priorities and computed gotos. Resulting SC programs remain fully C compliant because SC primitives boil down to C macros. Like SC, SJ also extends a programming language within itself. Because C offers computed gotos and Java does not, SJ exploits the `switch-case` statement. Auxiliary labels and additional internal book keeping are required.

Reactive C [8] is an extension of C. It is inspired by Esterel and employs the concepts of ticks and preemptions, but does not provide true concurrency. FairThreads [5] are an extension introducing true concurrency implemented via native threads. There are also macros for expressing automata. SJ does not use Java native threads, but does its own, light-weight thread book keeping.

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. SJ additionally is able to deal with signals and thread hierarchy, offering dynamic priorities and dynamic thread switching.

As already pointed out, the scheduling of SJ threads is similar to coroutines [7]. 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[1]. 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. There exist also an embedded variant of SJ not even using Java reflection. The advantage is a light-weight and platform independent concurrency implementation.

[1] http://openjdk.java.net/projects/mlvm/

3 Overview of SJ Primitives

SJ is an extension to Java that is written in pure Java itself. An SJ program `extends` the abstract class `SJProgram` offering the SJ synchronous primitives. These primitives are listed in Table 3.1 and are divided into two categories: (1) Control-flow primitives enabling deterministic concurrency and preemption, and (2) signal primitives enabling deterministic communication.

We here provide a first, summary overview of these primitives. A more in-depth discussion with examples follows in Sections 4 and 5.

Control-Flow Primitives For forking new SJ threads (in the following we will refer to SJ threads as *threads*), the `fork()` primitive is used, as explained further in Section 4.4. SJ internally keeps track of all forked threads. Joining all terminated descendants, i. e., forked threads, can be accomplished using the `joinDoneCB()` predicate. A sequence of forks must be ended by a call to `forkEB()` that continues the current forking thread at the specific label.

The `B` at the end of the primitive's name indicates that a Java `break` must follow this primitive. This also applies to all other primitives ending with a `B`. The reason is that after such a primitive, a possible dynamic re-scheduling of all concurrently running threads may be required. The Java `break` ensures this. A more detailed discussion follows in Section 4.3. For simplicity reasons, we will mostly omit the `B` in the following sections, except for source code examples.

Defining synchrony bounds is used to declare a chunk of code, for a tick per thread. At the end of this chunk of code it is necessary to declare this boundary using the `pauseB()` primitive that will continue the current thread at the given label for the next tick. Before, it will give other concurrent threads the chance to finish their execution of the current tick. To declare that a thread has finished all its work for this and *all* following ticks the `term()` primitive is used.

For influencing the control-flow, SJ offers a `gotoB()` primitive that instantaneously jumps to a defined label the next time the `while` loop of the `tick()` method is restarted. Section 4.3 gives more insights into the `while` loop and the `tick()` method. The `abort()` primitive recursively and strongly aborts all descendant threads that were forked by the current thread if there where any, `suspend()` suspends them for the current tick. Implementing a strong abort transition from one state represented by a label to another is accomplished using the `transB()` primitive. For implementing weak aborts, the `prioB()` primitive must be used to ensure that the abort happens with a lower priority than other code that should run before. It lets one modify the priority of the current thread for the current or for future ticks. Preemption is explained in more depth in Section 5.

SJ Primitive	Explanation
`fork(`l`, `p`);`	Forks a new thread at label l with priority p. All forked threads are descendants of the current thread and can be aborted by `abort()`.
`forkEB(`l`);`	Continues the current thread at label l with the same priority. The last `fork()` primitive must be a `forkEB()`. This primitive must be followed by a `break`.
`joinDoneCB();`	Joins all descendant threads that have been forked until these threads terminate by calling `term()`. This primitive must be followed by a `break`.
`pauseB(`l`);`	Stops the execution of the calling thread for the current tick. In the next tick this thread will continue at label l with the same priority. This does not effect the execution of parallel threads. This primitive must be followed by a `break`.
`termB();`	Terminates the currently running thread if all work of it has been done. This primitive must be followed by a `break`.
`gotoB(`l`);`	Jumps to label l. This primitive must be followed by a `break`.
`abort();`	Recursively aborts all descendants created by the current thread.
`suspend();`	Recursively suspends all descendants created by the current thread for the current tick.
`transB(`l`);`	Shorthand for `abort()` and `gotoB(`l`)`. This primitive must be followed by a `break`.
`prioB(`l`, `p`);`	Lowers the priority of the running thread to p and later continues at label l. This causes a new schedule in the same tick where the thread with the highest priority will execute next. This primitive must be followed by a `break`.
`Signal` s	Initializes a pure signal s.
`ValuedSignal` v`=new ValuedSignal("`v`", MULTIPLY);`	Initializes a valued integer signal v combined with multiplication.
s`.emit()`	Emits a pure signal s.
v`.emit(`val`);`	Emits a valued integer signal v with value val.
s`.isPresent()`	Enables branches: Predicate for testing whether signal s is present or not.
v`.getValue()`	Retrieves the value of valued signal v.
s`.pre()`	Retrieves the instance of signal s at previous tick.

Table 3.1: Sequential control-flow primitives for deterministic concurrency with transitions and preemption actions. SJ Signal primitives for deterministic communication.

Signal Primitives Signals can be defined as pure signals or as valued signals. Like in Esterel or SyncCharts, pure SJ signals only carry their present status. Local and output signals are absent by default for a tick. Emitting (`emit()`) a signal makes it become present for the tick. Signals can be emitted multiple times while computing the reaction of a tick.

Valued signals are pure signals that carry a value in addition to their present status. In the current implementation of SJ this is limited to integer values. The value of a valued signal changes when it is emitted with a new, other value. It can be retrieved by calling the `getValue()` method. Valued signals can also be emitted multiple times with different values within a tick because a *combination function* must be defined. This combination function must meet the criteria of being associative and commutative, e. g., multiplication or addition satisfy these criteria for integer values. Using `isPresent()` on a signal, one can test for the present status of it. All potential writers, which possibly emit the signal, must have run before. Otherwise a run time error will be raised. The `pre()` method lets one access a signal's instance of previous ticks, e. g., to test for the present status or the value of a signal in the previous tick.

Pure signals can be defined as instances of the `Signal` class. Java reflection is used to initialize them automatically. In an embedded variant of SJ (used in Section 7) without Java reflection, pure signals also need to be explicitly initialized. Valued signals always need an explicit initialization that additionally declares a combination function.

4 Deterministic Concurrency

4.1 The Producer-Consumer (PC) Example

The Producer-Consumer (PC) example in Figure 4.1a, inspired by the Producer-Consumer-Observer (PCO) example of Lickly et al. [12], is an example of concurrency with Java threads. It is a simplified version with a producer and a consumer thread only but not an observer. The class `PC` specifies a producer and consumer Java thread in its constructor that will run concurrently. Both Java threads share a common `monitor` buffer object created in Line 6. The producer Java thread is going to produce data in its `run()` method (Lines 32–36) and the consumer Java thread is going to consume the produced data in its `run()` method (Lines 43–48). There is no synchronization constraint specified, neither in the producer nor in the consumer Java thread, although the producer obviously has to run before the consumer. The buffer size is one (`BUF`), which is an additional constraint. All synchronization is expressed in the shared buffer object (`monitor`). It suspends Java 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 Java thread has to run before the consumer is realized only implicitly. With `notifyAll` (Lines 19 and 26) all producer and consumer Java threads possibly waiting are awoken. They may set themselves to `wait` again (Lines 16 and 23) afterwards immediately without doing anything. Scheduling has large influences on possible interleavings and the actual execution order that is totally unpredictable. Hence, execution time is also hard to predict.

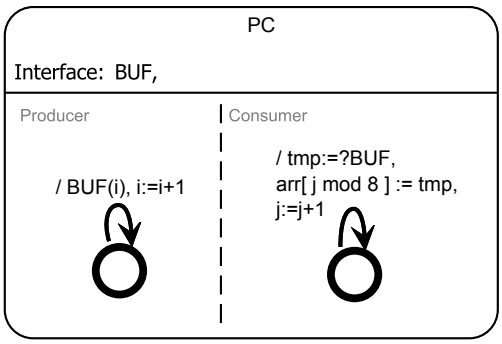
The situation becomes worse if one wants to add an additional observer thread like in the original example [12]. If the observer does not consume data but needs to run after the producer and before the consumer, this also has to be expressed in the `Monitor` class specifying the shared buffer. Overhead of poorly scheduled executions with unnecessary awoken Java threads will consequently grow. A related problem is the creation and killing of Java threads for simple tasks, which is also inefficient. An alternative is to re-use Java threads of a thread pool, which is more efficient but uses more system resources.

4.2 The PC Example in SyncCharts

To illustrate how a reactive application is typically implemented in a synchronous setting, Figure 4.1b shows the SyncCharts [2] version of the PC example. A common pattern for embedding reactive control is to have a *tick function* that reads in the input, computes the reaction, and possibly updates the system state. The SyncCharts version shows the behavior done per tick. The tick function synthesized from this SyncChart would have

```java
public class PC {
  static final int TICKS=100;
  static Monitor monitor;

  PC() {
    PC.monitor = new Monitor();
    new Thread(new Consumer()).start();
    new Thread(new Producer()).start();
  }

  class Monitor {
    boolean empty = true
    int BUF;
    synchronized void setBUF(int i) {
      while (!empty) {
         wait();
      }
      empty = false; BUF = i;
      notifyAll();
    }
    synchronized int getBUF() {
      while (empty) {
         wait();
      }
      empty = true; int returnValue=BUF;
      notifyAll();
      return returnValue;
    }
  }

  class Producer implements Runnable {
    void run() {
      for (int i=0; i < TICKS; i=i+1) {
        monitor.setBUF(i);
      }
    }
  }

  class Consumer implements Runnable {
    private int tmp;
    private int[] arr = new int[8];

    void run() {
      for (int j=0; j < TICKS; j=j+1) {
        tmp=monitor.getBUF();
        arr[j % 8]=tmp;
      }
    }
  }
}
```

(a) Java threads



(b) SyncChart

```java
import sj.SJProgram;
import examples.PC.StateLabel;
import static examples.PC.StateLabel.*;

public class PC extends SJProgram<StateLabel> {
  // Declare all used thread state labels.
  // Producer and Consumer are SJ thread entry
  // labels and also continuation labels for
  // the broken up loops.
  enum StateLabel {
    InitPC,
    Producer,
    Consumer
  }
  static final int TICKS = 100;
  private int BUF, i, j = 0, tmp;
  private int[] arr = new int[8];

  // The constructor initializes the SJ program
  // with a starting label (InitPC) and an
  // initial priority 1.
  public PC() {
    super(InitPC, 1);
  }

  @Override
  // Main tick funtion
  public void tick() {
    while (!isTickDone()) {
      switch (state()) {

        // Forking producer with priority 2.
        // Consumer continues with priority 1.
        case InitPC:
          fork(Producer, 2);
          forkEB(Consumer);
          break;

        case Producer:
          BUF = i;
          i = i + 1;
          pauseB(Producer);
          break;

        case Consumer:
          tmp = BUF;
          arr[j % 8] = tmp;
          j = j + 1;
          pauseB(Consumer);
          break;

        }
      }
    }

    public static void main() {
      // Create an instance of an SJ program.
      PC pc = new PC();
      for (int tick = 0;
           tick < PC.TICKS; tick++) {
        // Call its doTick() method that
        // resets all signals and calls tick().
        pc.doTick();
        if(pc.isTerminated()) {
          break;
        }
      }
    }
}
```

(c) SJ threads

Figure 4.1: Producer-Consumer (PC) example. Three different implementations of the same program are given: (a) Java threads, (b) SyncCharts [2], and (3) SJ threads.

to be called TICKS times to get the behavior equivalent to the Java version shown in Figure 4.1a.

SyncCharts can be seen as the graphical counterpart to the textual Esterel [4] synchronous language. The dotted line defines two concurrent activities, called *regions* in SyncCharts. The left region specifies the Producer and the right region specifies the Consumer thread. Each region has an initial state, shown with a bold border line.

Logically, the producer and the consumer execute both tick-wise in lock-step, thus there is no need to further synchronize these threads. Write-before-read scheduling constraints are considered automatically by the SyncCharts compiler or simulator. SyncCharts that cannot be scheduled in a way that satisfies all write-before-read constraints are rejected at compile time.

Listing 4.1: SJ Program Structure

```
1  public class MySJProg extends SJProgram<StateLabel> {
2    enum StateLabel {STATE0, STATE1}
3
4    public MySJProg() {
5      super(STATE0, 1); // Start at STATE0
6    }                   // with priority 1.
7
8    public final void tick() {
9      while (!isTickDone()) {
10       switch (state()) {
11         case STATE0:
12           // ... some code ...
13           break;
14         case STATE1:
15           // ... some code ...
16           break;
17       }
18     }
19   }
20 }
```

4.3 SJ Synchronous Reactive Control

As SJ brings synchronous and reactive concepts to the Java language, it also comes with a `tick()` method. Listing 4.1 illustrates the basic structure of an SJ program.

An SJ program is a Java class that extends `SJProgram`. It defines a finite set of states where this program or system can be in (Line 2). The constructor specifies the initial state (Line 5) together with a thread priority, discussed later in Section 4.4. The `tick()` method defines the behavior of the program in each of its states (Lines 8–19). The `while` loop ensures that the computation of the complete reaction (tick) that may consist of several computational steps is run until `isTickDone()` returns true. Computations that occur within one tick are considered to take no time and constitute the reaction. SJ introduces special Java methods, i. e., the SJ primitives introduced in Section 3, see also Table 3.1. During one reaction/tick these primitives allow for jumping from one state to another, to concurrently run code of several states as threads, and to create a hierarchy of such threads for enabling preemption as explained in Section 5. A

special `pause()` primitive states the tick boundary of a thread and possibly allows to transfer control to other running threads.

Figure 4.1c shows how to call an SJ program from within other Java code. In Line 60 the SJ program instance is created. Line 66 shows how to call the `doTick()` method, which is a wrapper for the implemented `tick()` method additionally resetting output signals to be absent by default before the next reaction is computed.

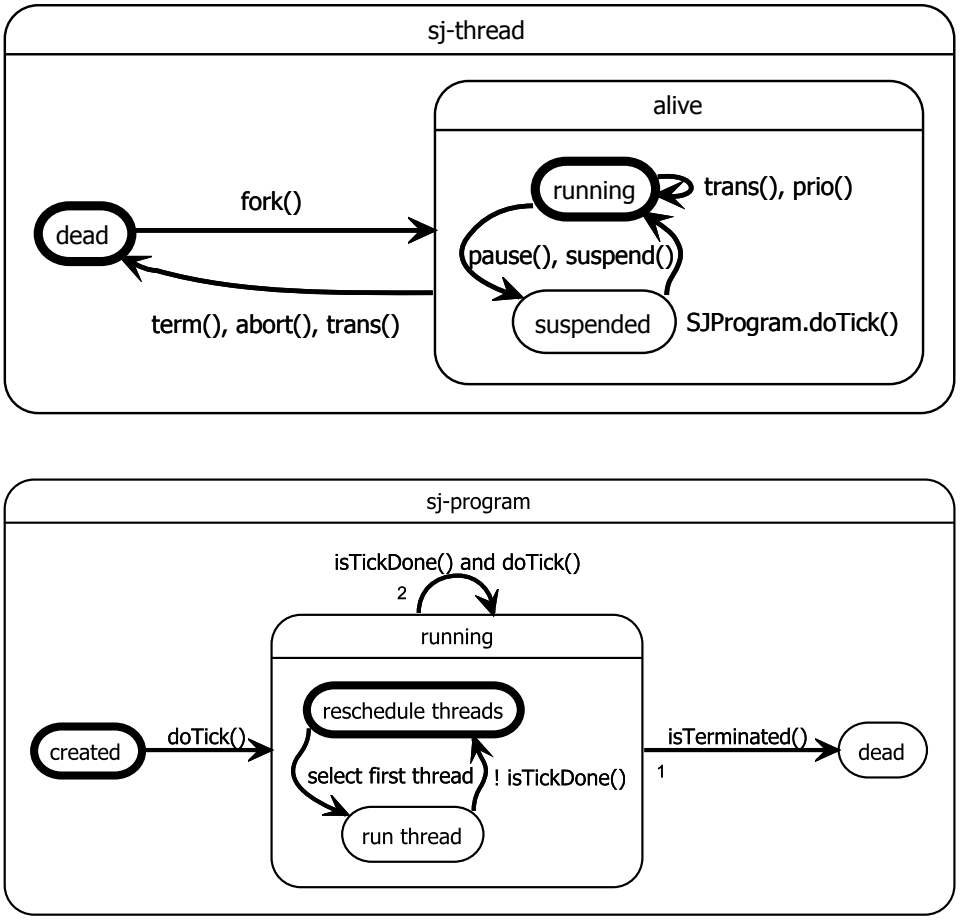


Figure 4.2: Life cycle of an SJ thread (top), and life cycle of an SJ program (bottom), using the SyncChart notation (see also Section 4.2).

4.4 SJ Cooperative Threads

SJ Java programs have one main thread of control. Its behavior is defined in the `tick()` method. To specify concurrent behavior, additional threads can be defined using the `fork()` primitive.

Figure 4.2 shows 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 have to be forked and initially are dead. When being forked, a thread becomes alive. Alive threads can act

as normal Java programs and execute code that has been specified within the `tick()` method for this thread. This can be Java code mixed with SJ primitives. Threads are able to perform transition changes by calling `trans()` to instantaneously jump from one labeled thread state to another. They change their priority using the `prio()` primitive. Finally, they can call `pause()` or `suspend()` to finish execution for the tick. At the end of their work, threads usually terminate (`term()`) or are aborted (`abort()`) from their parents as 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.

While running, an SJ program continuously reschedules threads and selects and runs the first one of a priority queue, see also Figure 4.2. After this thread declares reaching its tick boundary or an SJ primitive that may require a rescheduling 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.

Thread Priorities Threads always are associated with a priority. The priority of the main thread is defined in the constructor of the SJ program, e. g., Line 9 of Listing 4.1. Threads are scheduled by the `state()` method. It keeps track of all threads and their current priorities and always executes the thread with the highest priority. Within a synchronous tick, i. e., before the current tick is done and while the `tick()` method executes its `while` loop, a thread can only lower its priority using the `prio()` primitive. This is a necessary restriction to guarantee that the order of execution is deterministic and hence the behavior is deterministic too.

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 [7], but in contrast to typical coroutines, threads in SJ do not specify that another specific thread should resume, but only when *some* other thread may resume. Then, a separate dispatcher choses the thread to resume. It decides this dynamically using the priorities given to the threads. Hence, these priorities are crucial for influencing the scheduling of threads. E. g., if two threads communicate, the writer needs to have a higher priority than the reader of a value.

Figure 4.3 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 `prio(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, the Thread2 executes some code including two transition changes by calling the `trans()` method. This means that its thread program counter maintained by SJ is changed but this does not involve a thread rescheduling. After this, Thread2 calls `pause()` to indicate that it finished execution for this tick. `state()` now selects Thread1 again because it has the highest priority of

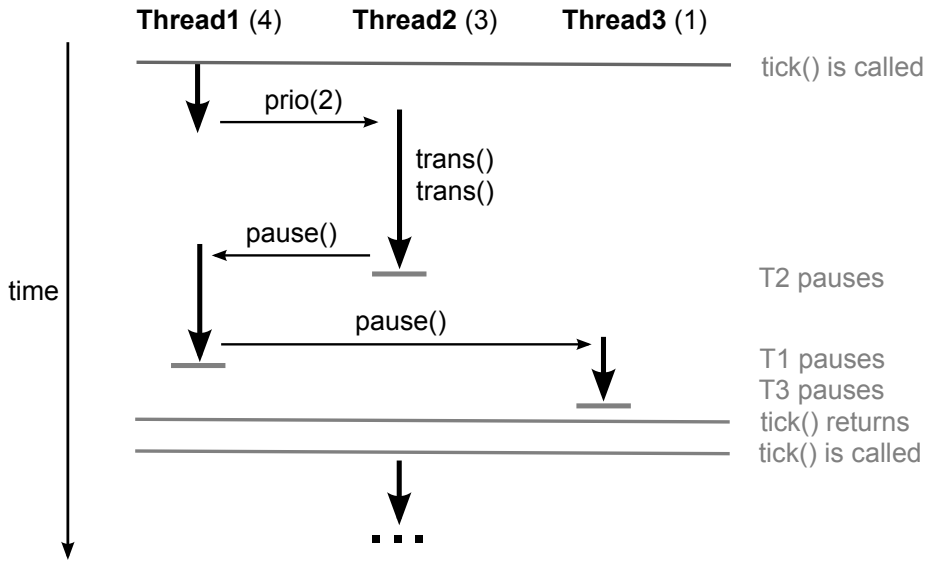


Figure 4.3: Cooperative SJ threads continue each other at specific synchronization points, e. g., prio() or pause(), similar to coroutines in Figure 1.2. A dispatcher dynamically chooses the next thread to continue based on priorities, given in parenthesis.

2 of all non-suspended, i. e., running, threads. When Thread1 also calls `pause()` to indicate it has finished execution for this tick, finally, Thread3 with priority 1 is selected to run its code. When Thread3 calls `pause()` no other thread needs to be scheduled for execution in this tick. Hence, the `tick()` method returns. Scheduling of following ticks may look similar to this tick. The first thread to run in the next tick is the one with the highest priority. This may be a different thread compared to previous ticks as threads may decide and are allowed to raise their priority for future ticks.

Threads, States and Labels Practically, thread states in SJ belong to labels of the `switch-case` statement. Forking a thread using the SJ `fork()` primitive means specifying a label where the thread starts its control. Jumping to other thread states from there changes the current thread's coarse program counter that is linked to a state label. Loops or other code that takes time, i. e., contains a `pause()` primitive, need to be broken up and the behavior needs to be expressed using labels and transitions. This is because SJ needs to restart the `while` loop for scheduling another thread in its `state()` method as explained in the previous section. An example is shown in Figure 5.1, where the `while` loop in Line 8 of the SC version is broken up in the SJ version and implemented using the label `ABOMainStrong` and a `prioB()` primitive, jumping back to this label in Line 32.

4.5 The PC Example with SJ

Figure 4.1a shows the SJ version of the PC example, where a Producer writes data that are read by a consumer. The Java thread implementation was presented in Section 4.1. A problem with this solution was that the scheduling constraints need to be expressed implicitly using coordination data structures like monitors. The scheduling constraints could not 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.

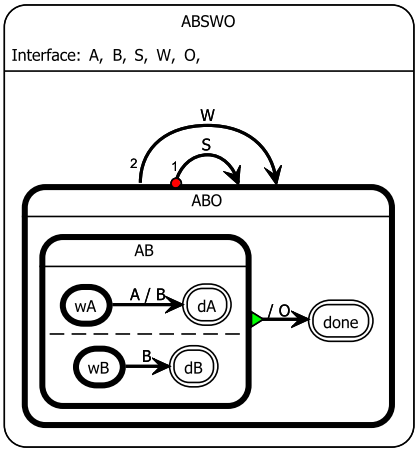
Using the concepts of SJ allows for having a light-weight concept of threads, minimum overhead, and a more explicit control over scheduling. Consider Figure 4.1c where the PC example is listed in Java using SJ constructs. The main part of the `tick()` method is the body of the `switch-case` statement. Essentially, it contains two kinds of labels, (1) labels representing thread entry points, e. g., `InitPC`, `Producer`, and `Consumer` and (2) labels representing different execution states of these threads, e. g., `Producer` and `Consumer` for expressing the broken up loops.

The initial state of the main thread is defined in the constructor of Line 23 and defines `InitPC` of Line 34. In Line 35 the producer thread is created using the SJ `fork` construct. Line 36 defines the continuation of the main thread that becomes the consumer thread with priority `1` at label `Consumer`. Scheduling constraints can be expressed explicitly in SJ using priorities. Because the producer has to run before the consumer, it has the higher priority of `2` compared to the main/consumer thread priority `1`.

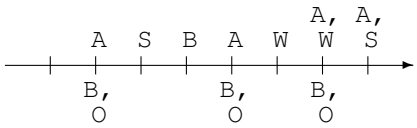
The SJ variant of the example shows concurrent execution of Java code that does not require Java threads and that uses a very light form of context switches with static (internal) thread IDs and their dynamic execution states (labels). Using priorities, the execution order is deterministic and predictable. In every tick, the producer and the consumer run in lock-step and the producer always runs before the consumer.

5 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 primitives, and a the capability to communicate among threads with synchronous-style signals.



(a) ABSWO SyncChart



(b) A logical tick time line that illustrates an execution trace, with input signals from the environment (above) and corresponding output signals (below).

```
1 public final void tick() {
2  while (!isTickDone()) {
3   switch (state()) {
4    case ABO:
5     fork(AB, 2);
6     forkEB(ABOMainStrong);
7     break;
8    case AB:
9     fork(wA, 4);
10    fork(wB, 3);
11    forkEB(ABMain);
12    break;
13   case wA:
14    if (awaitCB(A)) {
15     B.emit(); termB();
16    } break;
17   case wB:
18    if (awaitCB(B)) {
19     termB();
20    } break;
21   case ABMain:
22    if (joinDoneCB()) {
23     O.emit(); termB();
24    } break;
25   case ABOMainStrong:
26    if (S.isPresent()) {
27     abort(); prioB(ABO, 5);
28    } prioB(ABOMainWeak, 1);
29   case ABOMainWeak:
30    if (W.isPresent()) {
31     abort(); prioB(ABO, 5);
32    } prioB(ABOMainStrong,5)
        ;
33   break;
34  }
35 }
36 }
```

(c) SJ `tick()` method

```
1 int tick() {
2  MainThread (6) {
3   GOTO(ABO, 5);
4  }
5
6  Thread (ABO) {
7   FORK1(AB, 2);
8    while(1) {
9     if (PRESENT(S)) {
10     ABORT; PRIO(5);
11      GOTO(ABO);}
12     PRIO(1);
13     if (PRESENT(W)) {
14      ABORT; PRIO(5);
15      GOTO(ABO);}
16     PRIO(5);
17   }
18 }
19
20 Thread (AB) {
21  FORK2(WaitA,4, WaitB
        ,3);
22  JOIN;
23  EMIT(O);
24  HALT;
25 }
26
27 Thread (WaitA) {
28  AWAIT(A); EMIT(B);
29 }
30
31 Thread (WaitB) {
32  AWAIT(B);
33 }
34
35 TICKEND;
36 }
```

(d) SC `tick()` method

Figure 5.1: 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. The behavior of ABO is reset strongly by signal S and weakly by signal W.

In synchronous languages, signals are used for communication. For causality reasons, all writers to a signal that potentially could make a signal present in a tick must run before any reader. Typically, the compiler takes care of scheduling writers before readers, e. g., the SyncChart in Figure 4.1b there is no need to specify such a synchronization constraint.

The SJ programmer has more control over the execution path using priorities and modifying priorities during such a computation, as explained in Section 4.4. Additionally, SJ offers the concept of deterministic preemption to Java. The ABSWO example shown in Figure 5.1 illustrates preemption and the usage of signals. It is a modified version [19] of the ABRO example of Andr [3], the *hello world* of the synchronous world. Observe that the SJ Java code is much more comparable to the original SyncCharts specification than a Java thread implementation (see Figure 4.1) would be.

To illustrate preemption, ABSWO has two different preemptive reset self-loop transitions from state ABO to ABO. One is labeled with signal W as a trigger, the other is labeled with signal S as a trigger. The first one is a weak transition, the second one is a strong transition as indicated by the red circle. The difference in short is that when aborting ABO weakly, reactions, e. g., the emission of signal O, within the current tick that originates from within the left macro state, are permitted and not suppressed. In contrast when aborting ABO strongly, a possible emission of O is suppressed.

Communication Scheduling Signals can be used in SJ for a deterministic communication between concurrent threads. Signal B is emitted in the transition from wA to dA. Hence, signal B clearly serves for communicating between the two concurrent regions. Starting in state wA in the first region the system waits for the signal A to become present. In the second region below the system concurrently also awaits the presence of signal B. This happens to be the case either if signal B is set to be present by the environment or if signal A is set to be present by the environment and therefore the transition between wA and dA is taken that emits signal B. As soon as both concurrent regions are in their final states dA and dB, the system takes the *normal termination* transition marked with a green arrow and emits the output signal O all in the same tick.

Consider Figure 5.1c, which can be derived from the SyncChart in Figure 5.1a. To start the concurrent waiting for signal `A` and signal `B`, it forks two threads `wA` and `wB` in Lines 9 and 10. In Line 11 it specifies where to continue the current thread, which is at label `ABOMain`. Here the join-behavior is implemented that is described by the normal termination transition in the corresponding SyncChart. In Lines 14–16 and Lines 18 and 20 the behavior of awaiting signal `A` and signal `B` is defined. Note that in Line 15 signal `B` is emitted after successfully awaiting the presence of signal `A`. Scheduling gets interesting already at this point. To ensure that potential emitters of `B` run before readers, the priority of the thread `wA` in Line 9 is chosen to be `4` and hence to be higher than the priority `3` of thread `wB` specified in Line 10.

Preemption Scheduling Preemption allows for aborting descendant forked threads. It can be expressed in SJ in a weak and a strong form using priorities, as explained as

follows. The join-behavior in Lines 22–33 is crucial for ensuring a weak or strong form of leaving and re-entering the corresponding ABO state of the SyncChart represented by the label ABO in the SJ code. In general for implementing the behavior of both self-loop transitions, in both cases the code awaits either signal S or signal W in Lines 26 and 30, respectively. Afterwards, it aborts all descendant threads in Lines 27 and 31, resets the priority using the prioB() primitive of Lines 27 and 31, and re-enters the ABO thread state. Because the main thread starts with a priority of 5 in Line 6, this is also the priority of the ABOMain thread for the present test in Line 26. It means that testing for the presence of signal S has highest priority even over both concurrent threads that have lower priorities. Iff signal S is present, the abort of Line 27 takes place before any other internal reaction forked within ABO. This implements the strong abortion of the corresponding SyncChart. The prioB() primitive in Line 28 lowers the priority of the ABOMain thread from 5 to 1. Hence, all code afterwards that tests for the presence of signal W in Line 30 is scheduled after any other internal reaction forked within the ABO state. This implements the weak abortion of the corresponding SyncChart. Before re-entering ABO in the following tick, the priority of 5 has to be restored (Lines 27 and 31) so that the strong preemptive transition will be possible again.

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. This is clearly preferable over most other typical implementations of preemption, where this choice is up to (unpredictable) scheduling decisions.

6 SJ Implementation

We now cover the key concepts of the SJ implementation, Heinold [10] provides a more detailed discussion. SJ is implemented as a part of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1] modeling framework [9]. The SJ source code and the documentation is freely available under the Eclipse Public License (EPL) at the KIELER website.

An interesting part of SJ behind the scenes is the method `isTickDone()`. It returns true iff the current tick is done, i. e., the internal queue of running threads is finally empty. At the beginning of a tick, all non-suspended and non-terminated threads are added to this queue ordered by their priority. If a thread calls `prio()` its position in the priority queue is re-arranged. Because threads within a tick can only lower their priority it is always possible to re-schedule them *later* in this queue. A thread is removed from this queue when it calls `pause()`.

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 of threads as well as the handling of signals requires additional internal book keeping.

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

7 Lego Mindstorms Case Study

Lego Mindstorms[1] is an easily accessible embedded device with an ARM-based Next Lego Computing Brick (NXT) as its heart. It can be programmed using Java for Lego Mindstorms (LeJOS) where there is some support for the Eclipse platform. For validation, we brought an embedded variant of SJ onto the NXT device. A debugging facility inside the KIELER platform offers the possibility to debug SJ programs running on the NXT device step-by-step.

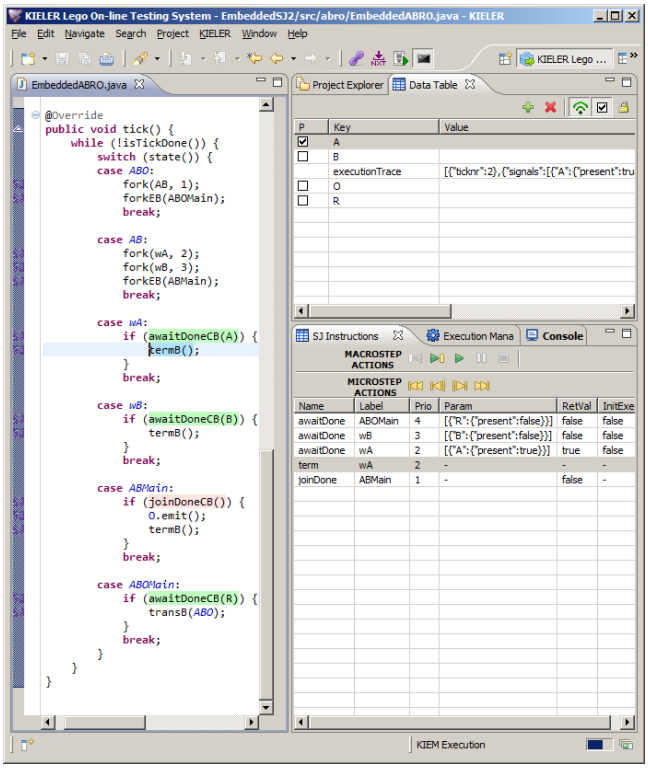


Figure 7.1: ABRO SJ program being debugged in KIELER while running on a Lego Mindstorms.

Figure 7.1 shows a setup where the ABRO example [2] is running on the NXT and is currently 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

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

because the `awaitDoneCB(A)` operation finished its execution. All taken micro steps can be observed in the SJ Instructions View. A micro step constitutes 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.

8 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 4. 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`.

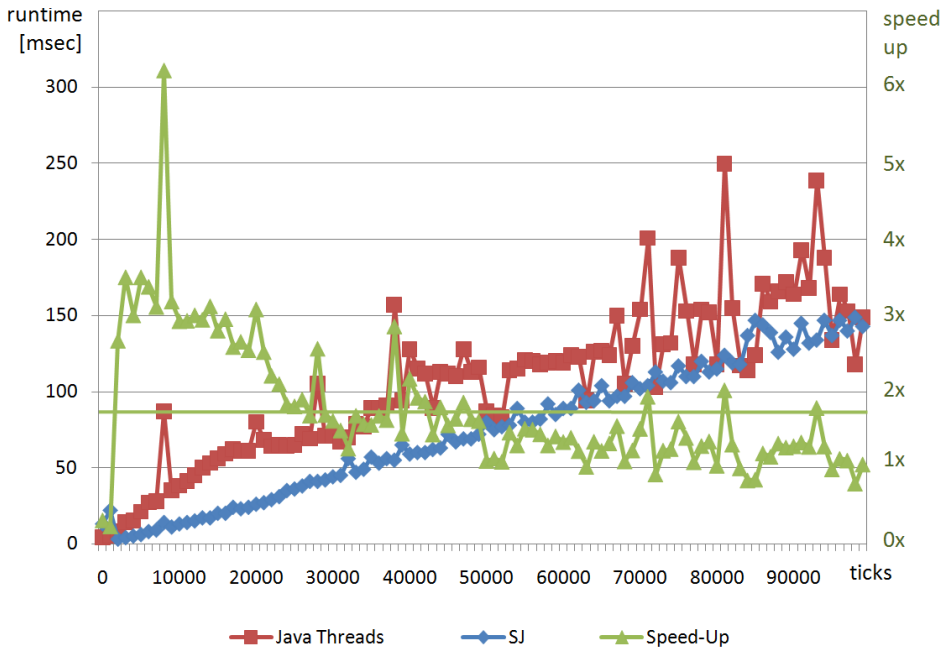


Figure 8.1: Worst-case run times, SJ vs. standard Java threads, for the PC example.

Figure 8.1 shows the execution time of each implementation over the variable number of ticks. For getting more 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. However the 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.

9 Conclusion and Outlook

We argued that solving synchronization problems with Java threads may become complex and problematic. We presented SJ as an adoption of the synchronous concepts for Java. We showed that SJ can help specifying concurrent threads in a light-weight and more robust way making use of deterministic synchronous concepts that allow for explicit expression of scheduling constraints. 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.

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 graphical data-flow models (e. g., PC example in Figure 4.1) or control-flow models (e. g., ABSWO example in Figure 5.1).

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[1] Project of the UC Berkeley.

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

Bibliography

[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, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[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] C. André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.

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

[6] 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, New York, NY, USA, 1987. ACM.

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

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

[9] H. Fuhrmann and R. von Hanxleden. Taming graphical modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6394 of *LNCS*, pages 196–210. Springer, Oct. 2010.

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

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

[12] 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, USA, Oct. 2008.

[13] A. Miyoshi, T. Kitayama, and H. Tokuda. Implementation and evaluation of real-time Java threads. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 166 –175, dec 1997.

[14] 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.

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

[16] 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, New York, NY, USA, 2010. ACM.

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

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

[19] 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.

[20] 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.