Computing (2015) 97:281–307
DOI 10.1007/s00607-014-0416-7

Light-weight Synchronous Java (SJL): An approach for programming deterministic reactive systems with Java

Christian Motika · Reinhard von Hanxleden

Received: 30 June 2013 / Accepted: 18 June 2014 / Published online: 30 July 2014

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 Light-weight Synchronous Java (SJL) approach presented here adopts the key concepts that have been established in the world of synchronous programming for handling reactive control-flow. Thus SJL not only provides deterministic concurrency, but also different variants of deterministic preemption. Furthermore SJL allows concurrent threads to communicate with Esterel-style signals. As a case study for an embedded system usage, we also report on how the SJL concepts have been ported to the ARM-based Lego Mindstorms NXT system. We evaluated the SJL approach to be efficient and provide experimental results comparing it to Java threads.

Keywords Reactive systems · Deterministic concurrency · Deterministic preemption · Synchronous programming · Light-weight scheduling · Java

Mathematics Subject Classification MSC 68N15 · 68N19

C. Motika (✉) · R. von Hanxleden
Department of Computer Science, Christian-Albrechts-Unversität zu Kiel, 24098 Kiel, Germany
e-mail: cmot@informatik.uni-kiel.de

R. von Hanxleden
e-mail: rvh@informatik.uni-kiel.de

1 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. Since Java 1.5, semaphores are part of Java's java.util.concurrent package. The thread concept introduces intense non-determinism. It is up to the scheduler to select one of the active threads to run and others to yield. To overcome this situation synchronization techniques like monitors or semaphores can be used to prune away non-deterministic behavior typically to avoid race conditions when accessing shared resources. This is not the problem that SJL tackles in first place. By means of using queues and barrier synchronization, monitors or semaphores can be further exploited to enforce scheduling constraints. Lee [16] argues that this is approaching the problem backwards. Introducing scheduling constraints into a highly non-deterministic concurrent program afterwards in order to achieve a deterministic scheduling is a difficult task and typically makes the program hard to read and hard to maintain. Synchronization techniques like monitors and semaphores for implementing deterministic concurrent behavior additionally often lead to deadlock situations that are hard to detect and understand, as further discussed by Lee [16]. Neither monitors nor semaphores are able to prevent serious scheduling overhead. In contrast to pruning away non-determinism from concurrent Java programs (by specifying scheduling constraints) we argue to use deterministic synchronous concepts from the beginning.

Contributions

We here present SJL, an approach that allows to directly embed deterministic reactive control-flow in Java, which encompasses concurrency and preemption. We side-step

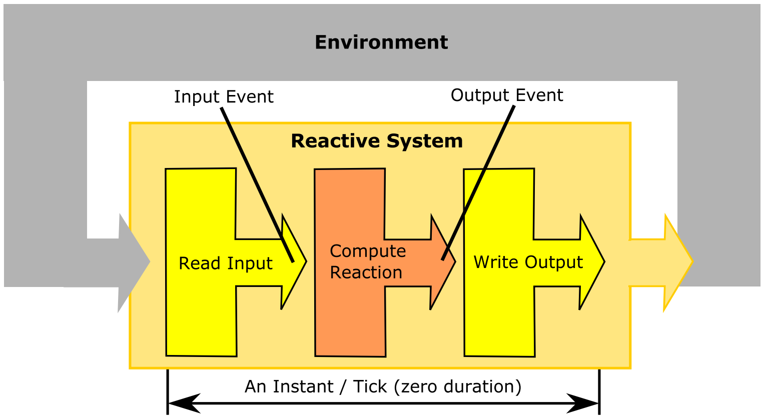


Fig. 1 Cyclic, discretized execution of a reactive embedded system computing a reaction to control the environment stimulated by some input from the environment

the traditional Java thread concept and its dependence on a—from an application point of view—unpredictable scheduler. Instead, SJL implements a light-weight application-level thread concept that combines coroutines with the synchronous model of computation (MoC). A case study shows how SJL can be used for solving common concurrent problems on reactive embedded targets.

Outline

In the next section, we discuss related work. Section 3 follows with a presentation of deterministic concurrency in SJL. Section 4 illustrates the usage of SJL preemption and how to mimic the behavior of synchronous-style signals. Section 5 discusses some implementation aspects, including deployment on an embedded example platform. Section 6 presents a Lego Mindstorms case study. Section 7 evaluates experimental results comparing SJL with traditional Java threads. We conclude in Sect. 8 and give some outlook on future work.

2 Related work

Nilsen [21] 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. The extensions also allow to specify a protocol that codifies how to add activities to a *real time executive* that is managing *resource budgets*. As a Real Time Java environment, Miyoshi [18] implemented prototype threads with special synchronization mechanisms as an extension package with minimal changes to the original Java Virtual Machine (JVM). Plsek et al. [22] also modified the JVM. These approaches cannot utilize the advantage of platform independence of the Java language, unlike SJL, which is itself implemented in Java and hence platform independent. Furthermore, SJL does not need any specific real-time environment or real-time constructs in order to ensure predictability and deterministic execution. Synchronous languages separate the concerns of physical timing and functionality. Because physical timing is orthogonal to determinism that SJL tackles, it can smoothly be combined with any Java real-time implementation.

To gain predictable Java applications there is another category of solutions, e. g., by Schoeberl [25], 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) [23] and the Reactive Java Optimized Processor (RJOP) [20] are both such hardware-based approaches. These could perfectly be combined with SJL, 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 [14] 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 [24] extends this by *mission modes* as coarse application building blocks. These cover different modes of operation during runtime of real-time applications. SJL could be utilized for implementing missions, in particular SJL makes most sense for single mission applications with a fixed number of threads.

2.1 Coroutines

One problem of Java threads is their performance depending on the actual implementation [27]. 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. 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 [26]. There are other attempts to implement coroutines using Java Native Interface (JNI), loosing Java's platform independence. SJL 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 SJL that does not even use Java reflection. The advantage is a light-weight and platform-independent implementation. In addition, unlike the aforementioned approaches, SJL offers deterministic preemption.

2.2 Synchronous languages

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. SJL does not use Java threads, but does its own, light-weight thread book keeping.

Precision Timed C (PRET-C) [1] similarly to Synchronous C (SC) [11] enriches the C programming language inspired by synchronous languages, but is restricted to static execution orders among threads. SJL additionally is able to deal with thread hierarchy, offering dynamic priorities and dynamic thread switching.

2.3 Synchronous C

SJL has been largely inspired by Synchronous C (SC), also known as SyncCharts in C [11], which 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 operators boil down to C macros. SC additionally borrows ideas from Precision Timed Machines (PRET) [17] w.r.t. predictability and advance the deadline instruction from a very low level timing specification to a more abstract execution order specification. Like SC, SJL also extends a programming language within itself. SJL does this for Java in order to allow for deterministic reactive control-flow specifications. Because C offers computed gotos and Java does not, SJL exploits the switch-case statement. Auxiliary labels and additional internal book keeping are required. As C and Java have different capabilities to express control flow, SC operators and SJL operators also differ. Section 4 later compares an SJL example with its SC counterpart in more detail. Köser [15] investigates the SC approach for modern multi-core computer architectures. SC, like SJL, can be used to implement the recently proposed *sequentially constructive* MoC [12], which loosens some restrictions the classical synchronous MoC by taking advantage of the sequential nature of C/Java-like languages. The sequentially constructive MoC also provides an approach to automatically compute the priorities employed by SC and SJL.

2.4 Synchronous Java

With Synchronous Java (SJ) [19] we already presented an earlier version of SJL and extended this in several ways. For simplicity reasons we relinquished the definition of special signals that may have unnecessary overhead during execution. We here present how to accomplish the same semantic meanings by using just standard Java variables. We redesigned some basic SJ operators to be significantly faster and simpler to be use. We present further implementation details of the light-weight reimplementation. Finally, we report on updates of our experimental results to validate our latest developments.

3 Deterministic concurrency in SJL

We now discuss the overall structure of SJL programs and the dynamic behavior of threads and their coroutine-like scheduling in order to provide deterministic, synchronous-style concurrency. Section 4 then describes preemption and signal handling in SJL.

3.1 SJL program structure

SJL is an extension to Java that is written in pure Java itself. Figure 2 illustrates the basic structure of an SJL program. An SJL program extends the abstract class SJProgram which provides the *SJL operators*, of which Table 1 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 SJL are expressed as different cases in a switch statement; if Java had a goto statement,

```
 1 public class MySJProg extends SJLProgram<StateLabel> {
 2   enum StateLabel {STATE0, STATE1}
 3
 4   public MySJLProg() {
 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 }
20 }
```

Fig. 2 Structure of an SJL program

Table 1 SJL operator overview yielding operators are marked with an asterisk (*)

SJL operator	Explanation
Thread management	
fork(l, p)*	Fork a new descendant thread at label l with priority p. A sequence of fork() operators should always be terminated with a gotoB(), prioB() or pauseB() to ensure the current forking thread advances its program counter and the scheduler is invoked
join()	Return true iff all descendant threads have terminated. If there is any descendant thread that has not yet terminated join() will return false
prioB(p, l)*	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
haltB()*	Shorthand for pauseB(l) where l is the current label
Further control flow	
gotoB(l)	Jump to label l
abort()	Recursively abort all descendants created by the current thread
transB(l)	Shorthand for abort() and gotoB(l)

these states could simply be statement labels. Each SJL 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 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 SJL 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. As an aid to the programmer, breaking operators are appended with a B. Some of the breaking operators are also *yielding*, marked with an asterisk in Table 1. 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.

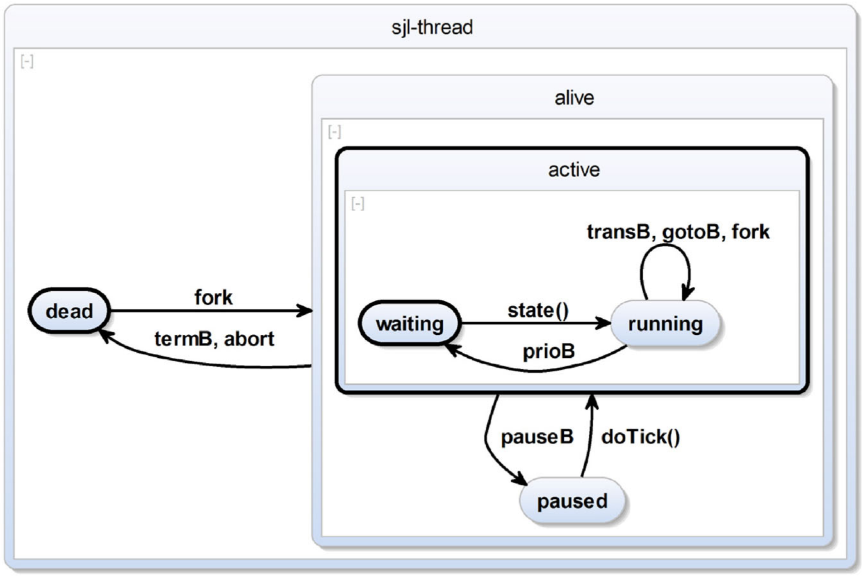
An SJL program also contains a main() method, not shown in Fig. 2, which calls the tick() method whenever a reaction should take place. More precisely, it calls the doTick() wrapper that re-activates all threads that are *alive* and have paused in the last tick computation before calling tick() to compute the current tick. The main() method is illustrated later in the example shown in Fig. 5c. The SJL program and thread life cycles are explained in more details hereafter (cf. Fig. 3).

3.2 SJL cooperative threads

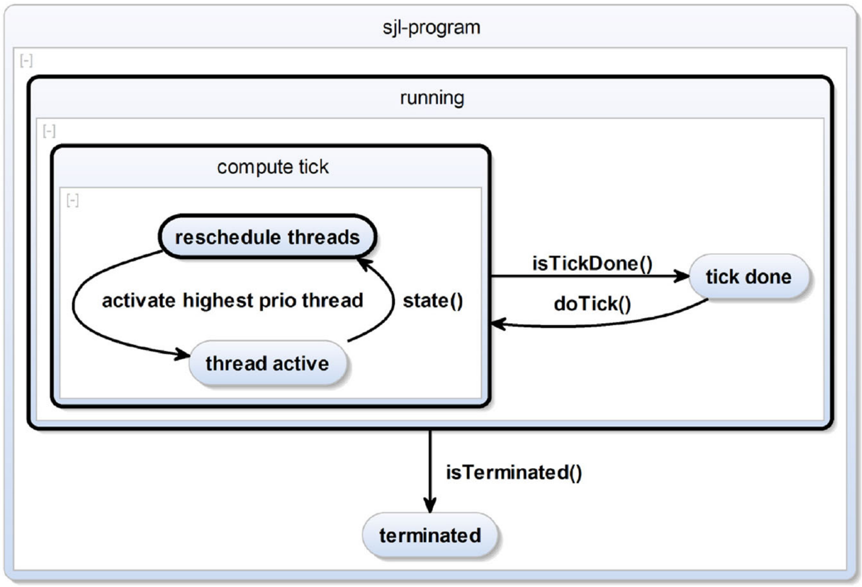
Figure 3a illustrates the life cycle of a thread. It can either be *dead* or *alive*. The main thread is *alive*, *active* and *running* by default, while other concurrent or child threads are initially *dead*. When being forked, a thread becomes *alive* and *active* but firstly *waiting* unless it is selected by the SJL scheduler, i. e., the state() method call, to run. Running 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 SJL operators.

Alive threads that still have work to do in the current tick remain *active*, threads that are still *alive* but done for the current tick are *paused*. Some SJL operators leave a thread running, such as transB(). The pauseB() operator leaves the thread *alive*, but suspends it for the remainder of the tick. As shown in Fig. 4, threads can yield and give control back to the SJL scheduler by lowering their priority using the prioB() operator. Afterwards they are *waiting* to be selected by the scheduler to continue execution in the current tick. At the end of their work, threads usually terminate (termB()) or are aborted by a (transitive) parent thread with abort(). SJL allows for building trees of threads for specifying hierarchical relations and make preemptions possible. SJL 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. 3b. Within a tick, the scheduler keeps selecting a thread from a queue of *running* threads. When this thread breaks (yields) and isTickDone() is still false, the next thread is selected for continuing its execution. If isTickDone() is finally true, i. e., there are no schedulable *active* threads left for the tick, the doTick() method

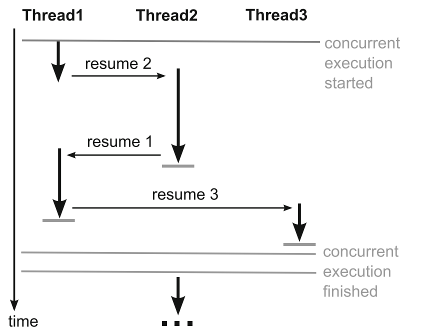


(a) Life cycle of an individual SJL thread

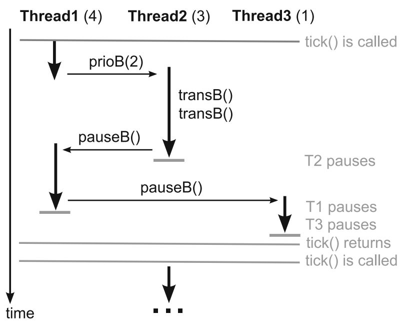


(b) Life cycle of a complete SJL program

Fig. 3 State diagrams for the reactive life cycle of an SJL program and its individual threads, initial states have a bold outline



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



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

Fig. 4 Comparison of cooperative thread scheduling concepts

returns and the SJL 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 *terminated*.

3.2.1 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 SJL 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 (Sect. 3.1) keeps track of all threads and their current priorities, and schedules from the currently running threads the one with the highest priority.

Note that priorities used by SJL are not dynamically used for resolving resource contention problems or scheduling decisions that may be required by scheduling policies like Earliest Deadline First (EDF). In contrast, priorities implicitly define a static sequential schedule that the scheduler will process. Because of the distinct sequential nature of SJL schedules, the computations are completely deterministic. Additionally note that the priority inversion problem is ruled out by construction because of the static sequential schedule and the synchronous tick boundary. Each tick corresponds to the release of all concurrent tasks/threads. The scheduler will enforce the thread with highest priority to run first in the deterministic sequential execution for a tick. This sequential execution is not dependent on any shared resources. But as the example in Sect. 3.3 reveals, ensuring a specific protocol for accessing shared resources may result in specific priorities, e. g., considering data dependencies and a write-before-read policy.

3.2.2 Cooperative thread scheduling

SJL's underlying thread scheduling is a cooperative, coroutines-like scheduling. The idea of coroutines [8] is to let threads cooperate, with themselves in charge of passing on control, instead of using a scheduler. Figure 4a 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.

SJL 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 SJL 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. This choice is driven by the thread priorities which are application-controlled and typically static. Hence, the priorities are crucial for ordering accesses to shared data within a tick. For example, 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 SJL 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 [12]. Hence, using SJL one is able model the constructive semantics [4] of SyncCharts [2] and the sequentially constructive semantics of Sequentially Constructive Charts (SCCharts). For SyncCharts, signals can be codified as explained in Sect. 4.

Figure 4b 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 SJL 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.

Table 2 summarizes the similarities and differences of SJL thread scheduling compared to coroutines. Additionally it compares standard Java threads with both.

3.3 The Producer–Consumer example

The Producer–Consumer (PC) example in Fig. 5, inspired by the Producer–Consumer–Observer (PCO) example of Lickly et al. [17], is a small-scale application with two

Table 2 Similarities and differences of coroutines and SJL cooperative thread scheduling, additionally compared to Java threads

	Coroutines	SJL	Java threads
Similarities			
Threads decide to yield	Yes	Yes	No
Arbitrary interleaving	No	No	Yes
Coarse program counter	Yes	Yes	No
Differences			
Scheduler present	No	Yes	Yes
Next thread selected by	Thread	Scheduler	Scheduler
Selection based on	Code	Priorities (highest)	JVM + OS dependent
Threads yield by	Explicit resume	prioB(), pauseB(), termB()	n/a

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. As in the original example of Lickly et al. [17], exemplarily, the computations of the consumer are two split: First receiving the value of BUF in a local variable tmp and second performing some further internal computations with the received value, e. g., writing it to an internal array arr.

3.3.1 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 Monitor buffer object.

In this example a monitor is used as a higher level synchronization mechanism. Other possibilities for implementing synchronization would have been the usage of lower level synchronization mechanisms like semaphores. Both mechanisms can be used to manage concurrent access to shared resources. This is typically done by using queues (monitors) or barrier synchronization (semaphores). Nonetheless there is no predictable scheduling as the scheduler may freely (non-deterministically) chose a thread to continue. Synchronization techniques are then used to prune away this non-determinism. Hence, neither monitors nor semaphores are able to prevent serious scheduling overhead.

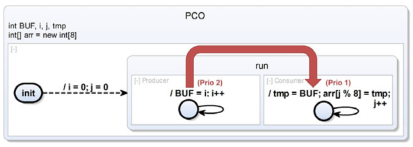
The Producer thread produces data in its run() method (lines 35ff), consumed by the Consumer thread in its run() method. There is no synchronization constraint explicitly specified, neither in the producer nor in the consumer thread, although the producer has to run before the consumer. All synchronization 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

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

  PC() {
    PC.monitor = new Monitor();
    new Thread(new Producer()).start();
    new Thread(new Consumer()).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++) {
        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++) {
        tmp = monitor.getBUF();
        arr[j % 8] = tmp;
      }
    }
  }
}
```

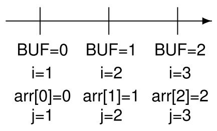
(a) Standard Java threads implementation

```
import sjl.SJLProgram;
import examples.PC.StateLabel;
import static examples.PC.StateLabel.*;

public class PC extends
    SJLProgram<StateLabel> {
  enum StateLabel {
      InitPC, Producer, Consumer }

  static final int TICKS = 100;
  private int BUF, i = 0, j = 0, tmp;
  private int[] arr = new int[8];

  public PC() {
      super(InitPC, 2);
  }

  @Override
  public void tick () {
    while (!isTickDone()) {
      switch (state()) {
        case InitPC:          // Prio 2
          fork(Consumer, 1);
          gotoB(Producer);
          break;

        case Producer:        // Prio 2
          BUF = i;
          i++;
          pauseB(Producer);
          break;

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

  public static void main() {
    PC pc = new PC();
    for (int t = 0;
          t < PC.TICKS;
          t++) {
      pc.doTick();
      if (pc.isTerminated())
        break;
    }
  }
}
```

(c) Implementation with SJL threads



(b) SyncCharts implementation



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

Fig. 5 The Producer–Consumer (PC) example

notifyAll() all producer and consumer threads possibly waiting are awoken. These may wait() again afterwards immediately without doing anything (lines 17 and 25).

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 [17]. 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.

3.3.2 The PC example with SJL

SJL allows for light-weight threads and more explicit control over scheduling by the application itself. Consider Fig. 5c where the PC example is listed in Java using SJL constructs, according to the structure already discussed in Sect. 3.1. 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 gotoB() 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 gotoB() 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 again. As explained earlier these computations are receiving the BUF in a local variable tmp that then is written to an internal array arr. After the Consumer pauses as well the 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 SJL program is also illustrated in the *logical tick time line* in Fig. 5d, which highlights some variable assignments taking place within the first three ticks.

To summarize, the SJL 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. Hence, the deterministic execution schedule is ensured by priorities that are used by SJL constructs to sequentialize the access to BUF in order to maintain the write-before-read requirement. In general, 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.

3.3.3 Comparison of SJL program and SyncCharts implementation

SyncCharts is a graphical synchronous language with Esterel semantics. A full description of SyncCharts is given by André [2]. The PC program can also be specified using the SyncCharts formalism as shown in Fig. 5b.

At this point it is important to note that the underlying synchronous tick-wise computation of SJL and SyncCharts is fundamental for achieving determinism. In each tick the SJL program and the SyncCharts do only a finite amount of computations that are distinctly sequentialized by the given priorities in the SJL program and by the data dependencies in the SyncChart.

After initializing i and j the SyncChart immediately transitions to the run state. In the run state there are two concurrent regions, specifying the behavior of the Producer and the behavior of the Consumer. Regarding the shared variable BUF there is a dependency from the writing Producer to the reading Consumer as visualized by the red arrow. Hence, in each synchronous tick the Producer will be scheduled before the Consumer. The thread priorities for the Producer (2) and for the Consumer (1) used in the SJL implementation are derived from this data dependency. These priorities statically determine the order in which both SJL threads are executed in each synchronous tick.

4 Preemption and signals

After discussing the core SJL 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.

4.1 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. SJ [19], the predecessor of SJL, included an explicit synchronous signal class.

On the one hand this eases the use of synchronous-style communication in SJ. For example one thread could emit a signal s using s.emit() and another thread could test for its presence status using s.isPresent() within the computation of a reaction. Additionally, these signals always carry their present status of all previous ticks. The previous instance of a signal can be accessed by calling s.pre() on a signal s.

On the other hand these explicit signals of SJ makes it necessary to explicitly initialize them with a special initSignals() method that makes additional use of Java reflection. But Java reflection is not available on all embedded platforms. Further shortcomings are reduced efficiency and higher memory requirements during runtime where simple communication results in various method calls on various necessary signal objects. Moreover, holding previous instances available requires additional memory and computation in order to create new instances of every signal for each tick. Altogether the signal management also obfuscates and blows up the SJ implementation where signals might not always be desirable (cf. Fig. 5).

In contrast to SJ, SJL has no explicit signals but the same behavior can be accomplished by using boolean variables as shown in Fig. 6. The following rules must be satisfied in order to mimic synchronous signals correctly:

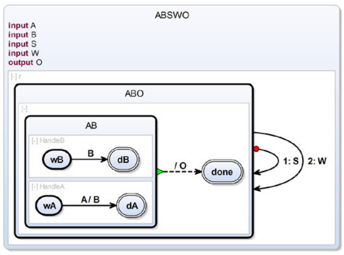
1. Signal variables representing local and output signals must be reset to false (absent) before each tick computation (cf. lines 2–3) and
2. signal variables may be set to true (present) but must not be set to false (absent) during a tick computation.

Alternatively, boolean signal variables representing local and output signals can also be reset to false (absent) in some other part of the enclosing program *before* calling doTick(). The emission of a signal s corresponds to setting a boolean variable s to true. Testing for presence of a signal s corresponds to testing a boolean variable s for whether it is true.

Valued signals can be realized by using an additional variable to store the value in combination of a boolean variable to store the present status as described before.

For both, SJ and SJL, priorities must ensure that all code writing to a signal (emitting it) executes before all code reading this signal (testing for its presence) during a tick computation.

To summarize, SJL can mimic signals with a one tick live span of its present status. This can be used to model signals as used in synchronous languages like Esterel, SyncCharts, or SCCharts. Additionally SJL can handle any Java variables and arbitrary Java objects with a live span that persist over the ticks. An example was given in Fig. 5 where an integer buffer BUF was used for communication and an integer array arr for thread-internal further computations.



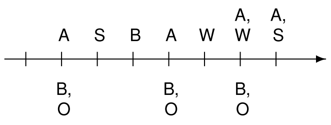
(a) Graphical SyncChart of ABSWO

```
int tick () {
  MainThread (ABO, 5) { // Prio 5/1
    FORK1(AB, 2);
    while(1) {
      PAUSE();
      if (PRESENT(S))
        TRANS(ABO);
      PRIO(1);
      if (PRESENT(W)) {
        PRIO(5);
        TRANS(ABO);
      }
      PRIO(5);
    }
  }

  Thread (AB) {    // Prio 2
    FORK2(wA,4, wB,3);
    JOIN();
    EMIT(O);
  }

  Thread (wA) {    // Prio 4
    AWAIT(A);
    EMIT(B);
  }

  Thread (wB) {    // Prio 3
    AWAIT(B);
  }

  TICKEND();
}
```

(b) SC tick() method

```
public final void tick () {
  O = false;
  while (!isTickDone()) {
    switch (state()) {
    case ABO:               // Prio 5
      fork(AB, 2);
      gotoB(ABOMainStrongEntry);
      break;
    case ABOMainStrongEntry: // Prio 5
      pauseB(ABOMainStrong);
      break;
    case ABOMainStrong: // Prio 5
      if (S) {
          transB(ABO);
      } else
          prioB(ABOMainWeak, 1);
      break;
    case ABOMainWeak: // Prio 1
      if (W) {
        abort();
        prioB(5, ABO);
      } else
          prioB(5, ABOMainStrongEntry);
      break;
    case AB:               // Prio 2
      fork(wA, 4);
      fork(wB, 3);
      gotoB(ABMain);
      break;
    case ABMain:        // Prio 2
      if (join()) {
        O = true;
        termB(); break;
      };
      gotoB(ABMain); break;
    case wA:             // Prio 4
      if (A) {
        B = true;
        termB(); break;
      };
      gotoB(wA); break;
    case wB:             // Prio 3
      if (B) {
        termB(); break
      };
      gotoB(wB); break;
    }
  }
```

(c) SJL tick() method



(d) 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 SJL and SC, illustrating weak and strong preemption. It also illustrates the usage of synchronous-signal-style boolean variables for communication. 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

4.2 Preemption

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. For example, 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 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. 6d. 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.

Figure 6c shows the SJL tick() function which precisely corresponds to the SyncChart of Fig. 6. In line 2 the output signal O is first reset to false (absent) for each call of tick(), i. e. for each reaction computation. This is in accordance to the explanation of the previous paragraph that illustrates how a synchronous-style signal handling can be implemented using standard Java variables. 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 (line 10). 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 (line 13) and possibly takes the (strong preemptive) self-transition to ABO (line 14); if this transition is not taken, it then, at priority 1, tests W (line 19) and possibly takes the (weak preemptive) self-transition (line 21); otherwise it again raises its priority again to 5, transfers control to ABOMainStrongEntry (line 23), and pauses (line 10). Concurrently, the thread started at AB first forks two threads wA and wB (lines 26 and 27) and then, in state ABMain, waits for them to terminate with join() (line 31) and then sets O to true, i. e., it emits O to be present in the case of a join (line 32) and terminates (line 33). It self-transitions otherwise (line 35). 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 (line 37), and if so, emits B (line 38) and terminates (line 39). It self-transitions otherwise (line 41). The thread starting at wB similarly tests B and possibly terminates or self-transitions. 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, SJL 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. [28] discuss how transition priorities to implement SyncCharts can be synthesized automatically.

For a brief comparison between Synchronous Java and Synchronous C, Fig. 6b 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. For example, 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. For example, the Thread(l) macro simply terminates the preceding thread and generates a label *l*.

5 Implementation notes

An interesting part of SJL behind the scenes is the method isTickDone(). It returns true iff the current tick is done, i. e., when the internal queue of *active* threads is finally empty. At the beginning of a tick, all *alive* threads are considered active by adding them 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(). It is then no longer considered to be *active*.

Another central method is the state() method that implements the SJL dispatcher and does the actual scheduling. It returns the next thread state label for the switch-case statement to continue execution. This is the label that is the state of the program

```
1   /** The coarse program counter of a thread. */
2   private ArrayList<State> coarseProgramCounter;
3   /** The current thread that is executing ( identified by its priority ). */
4   private int currentThread;
5   /** Alive threads where alive means NOT aborted and NOT terminated. */
6   private int[] alive ;
7   /** Active threads where active means NOT paused for a tick. */
8   private int[] active ;
9   /** The descendants of a thread (identified by its priority , encoded in bits). */
10  private int [][] descendants;
11  /** The parent of a thread ( identified by its priority ). */
12  private int[] parent;
```

Fig. 7 Basic data structures of an SJL program

counter of the next *active* thread from the top of the ordered priority queue. Forking and terminating threads as well as the handling of signals requires additional internal book keeping. This book keeping and further details about the scheduling implementation are sketched in the following paragraphs.

5.1 Basic SJL data structures

SJL needs some basic data structures to do the aforementioned bookkeeping and thread management. Figure 7 shows these structures. The coarseProgramCounter (line 2) is an array that stores for every thread (identified by its unique priority) a continuation label (of type State). The currently executing thread's id, i. e., its priority, is stored in currentThread (line 5). Two important fields are alive (line 8) and active (line 11). alive is an integer array that stores up to 32 *alive* threads per entry. active is an integer array that stores up to 32 *active* threads per entry. Refer to Fig. 3 for an explanation of thread states such as *alive* and *active*. Threads that are not contained in alive are considered to be *dead*. Threads that are not contained in active are considered to be *paused*. All threads in active must also be contained in alive. The fields descendants (line 14) and parent (line 17) are used for storing parent-child relationships. An entry of descendants is an integer array for a parent thread identified by its priority which holds up to 32 child threads per entry. An entry of the parent array for a child thread identified by its priority points to the id of its parent thread.

In the following the doTick method is discussed that uses some of the data structures and is responsible to call the implemented tick() method as indicated in Sect. 3.1.

5.2 doTick() implementation

The tick() method implements the reaction computation of an Light-weight Synchronous Java program. In order to stimulate the SJL program from outside the implementing class the doTick() is called because it encapsulates the call to tick(). As shown in Fig. 8 it will first copy all entries of alive into the active array (line 3). Hence, all *alive* threads that may have *paused* in the previous tick will become *active* again for the current tick computation and can then be selected by the SJL scheduler, as explained in the next paragraph. doTick() will then internally call the tick() method to

```
1   public boolean doTick() {
2       // Clone the alive threads means making all alive threads enabled for the current tick
3       setCopyFrom(active, alive);
4       // Run the tick () method
5       tick ();
6       // Return whether the program terminated
7       return isTerminated();
8   }
```

Fig. 8 Implementation of doTick() method that is repeatedly called until it returns false

```
1   protected State state() {
2       currentThread = bitScanReverse(active);
3       if (currentThread != −1) {
4           return coarseProgramCounter.get(currentThread);
5       }
6       return null;
7   }
```

Fig. 9 Light-weight scheduler of an SJL program

compute the reaction (line 5) and returns (line 7) whether the program has terminated, i. e., whether alive contains no more threads.

5.3 Scheduler implementation

The scheduler is called whenever a restart of the switch–case statement is enforced by a break, as explained in Sect. 3.1. In this case the state() method returns the next continuation point of the currently executing thread. Underneath the state() method call the SJL scheduler always selects the thread with the highest priority and returns its program counter. This is fulfilled by performing a bit scan reverse operation of all *active* threads, as shown in Fig. 9 (line 2). This is how the priority queue is implemented. The following line will lookup the resumption case label of the current thread in the coarseProgramCounter array (line 4). The switch–case statement will then jump to this case and resume its execution there. Note that bitScanReverse() should never return −1 which would indicate an error condition (line 6) caused by calling state() with no *active* threads.

As a side-effect state() will update currentThread (line 2) that is used internally by several SJL operators. Some of them are discussed in the next paragraph.

5.4 SJL operator implementations

There are several SJL operators, as presented in Sect. 3.1. We will further discuss the internals of some important representatives of these operators in the following. Figure 10 lists the code of gotoB(), pauseB(), fork(), termB() and join().

gotoB() just updates the coarseProgramCounter array entry of the currentThread to the desired resumeState (line 3).

pauseB() first updates the coarseProgramCounter using gotoB() (line 8). Then it removes the currentThread from the *active* threads (line 10) such that it

```
 1      protected void gotoB(State resumeState) {
 2          // Update the state label
 3          coarseProgramCounter.set(currentThread, resumeState);
 4      }
 5
 6      protected void pauseB(State resumeState) {
 7          // Update the state label
 8          gotoB(resumeState);
 9          // Remove the thread from the active ones (it is paused now)
10          setDelete(active, currentThread);
11      }
12
13      protected void fork(State forkedState, int prio) {
14          // Create a new thread (identified by prio)
15          coarseProgramCounter.set(prio, forkedState);
16          // Add child and parent relations
17          setAdd(descendants[currentThread], prio);
18          parent[prio] = currentThread;
19          // Add new thread (identified by prio) as an alive & enabled one
20          setAdd(alive, prio);
21          setAdd(active, prio);
22      }
23
24      protected void termB() {
25          // Remove the thread from the active and the alive ones
26          setDelete(active, currentThread);
27          setDelete(alive, currentThread);
28      }
29
30      protected boolean join() {
31          // True iff all children terminated, i.e., none of the children are contained in alive,
            hence sets are disjoint
32          return (setDisjoint(alive, descendants[currentThread]));
33      }
```

Fig. 10 Light-weight implementation of some SJL operators

is *paused* now for the current tick until doTick() will make it *active* again for computing the next tick's reaction.

fork() creates a new entry in the coarseProgramCounter array (line 15) for the new created child thread with the priority prio where forkedState is the first starting point (coarse program counter state) of this new thread. It will then add the child as an descendant (line 17) for the current forking parent thread. It will finally set the parent relationship accordingly (line 18) before it makes the new child thread *alive* (line 20) and *active* (line 21) by adding it to the according arrays.

termB() just removes the currently executing thread from both, the active (line 26) and the alive (line 27) array.

join() returns true (line 33) iff all direct children have been terminated, i.e., none of the direct child threads are listed in the descendants entry for the current parent thread. Hence, the sets alive and descendants[currentThread]) must be disjoint. Otherwise join() returns false.

```
private void setAdd(int[] set, int id) {
            int group = getGroupId(id);
    set[group] |= (1 << getIdInsideGroup(group));
}

private void setDelete(int[] set, int id) {
            int group = getGroupId(id);
    set[group] &= ~(1 << getIdInsideGroup(group));
}

private void setCopyFrom(int[] set1, int[] set2) {
    for (int i = 0; i < set2.length; i++) {
        set1[i] = set2[i];
    }
}
```

Fig. 11 Excerpt of underlying bit-level internal set operations

The above SJL operator implementations and the doTick() method make use of some bit-level operations such as setAdd(), setDelete(), or setDelete(). These are discussed in the next paragraph.

The set operations use an efficient, compact bit-vector representation for sets to implementat the SJL operators. This means that every thread id, i. e., every thread priority is projected to a specific bit in the integer arrays, e. g., alive or active. These bits are grouped by 32 bits per integer array entry. Some of the underlying set operations are sketched in Fig. 11.

5.5 Further details and context

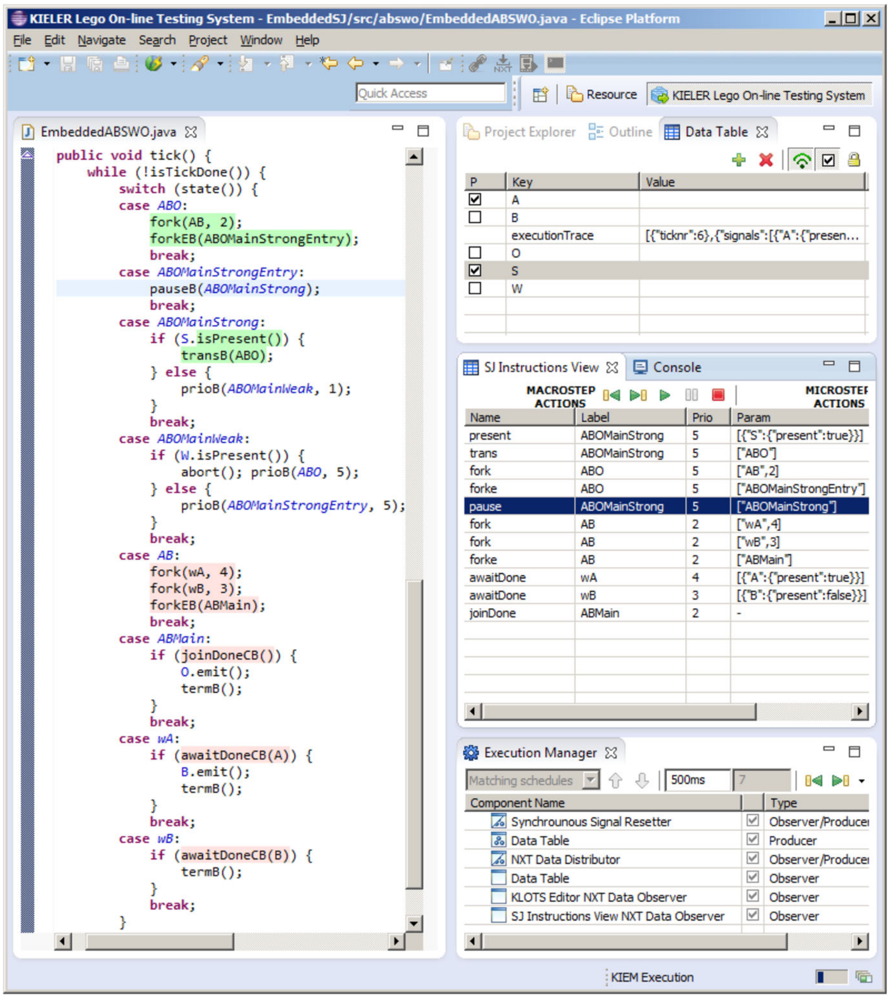
A further, more detailed discussion of the SJ implementation is given by Heinold [13]. SJL and SJ are implemented as a part of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1] modeling framework. KIELER [10] focuses the pragmatics of modeling tools to enhance the practical modeling interaction between the modeler and the modeling tool. A key enabler is automatic layout for graphical models. As described by Fuhrmann et al. [10], objectives are the creation, the editing and the browsing of graphical and textual models, statically but also dynamically during simulation runs. Both SC and SJL serve as designated targets for simulating synchronous models and exploiting semantic issues. The SJL and the SJ source code and the documentation is freely available under the Eclipse Public License (EPL) at the KIELER website.

6 Lego Mindstorms case study

Lego Mindstorms[2] 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

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

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



```
public void tick() {
    while (!isTickDone()) {
        switch (state()) {
        case ABO:
            fork(AB, 2);
            forkEB(ABOMainStrongEntry);
            break;
        case ABOMainStrongEntry:
            pauseB(ABOMainStrong);
            break;
        case ABOMainStrong:
            if (S.isPresent()) {
                transB(ABO);
            } else {
                prioB(ABOMainWeak, 1);
            }
            break;
        case ABOMainWeak:
            if (W.isPresent()) {
                abort(); prioB(ABO, 5);
            } else {
                prioB(ABOMainStrongEntry, 5);
            }
            break;
        case AB:
            fork(wA, 4);
            fork(wB, 3);
            forkEB(ABMain);
            break;
        case ABMain:
            if (joinDoneCB()) {
                O.emit();
                termB();
            }
            break;
        case wA:
            if (awaitDoneCB(A)) {
                B.emit();
                termB();
            }
            break;
        case wB:
            if (awaitDoneCB(B)) {
                termB();
            }
            break;
        }
```

P	Key	Value
☑	A	
☐	B	
	executionTrace	[{"ticknr":6},{"signals":[{"A":{"presen...
☐	O	
☑	S	
☐	W	

Name	Label	Prio	Param
present	ABOMainStrong	5	[{"S":{"present":true}}]
trans	ABOMainStrong	5	["ABO"]
fork	ABO	5	["AB",2]
forke	ABO	5	["ABOMainStrongEntry"]
pause	ABOMainStrong	5	["ABOMainStrong"]
fork	AB	2	["wA",4]
fork	AB	2	["wB",3]
forke	AB	2	["ABMain"]
awaitDone	wA	4	[{"A":{"present":true}}]
awaitDone	wB	3	[{"B":{"present":false}}]
joinDone	ABMain	2	-

Component Name		Type
Synchrounous Signal Resetter	☑	Observer/Producer
Data Table	☑	Producer
NXT Data Distributor	☑	Observer/Producer
Data Table	☑	Observer
KLOTS Editor NXT Data Observer	☑	Observer
SJ Instructions View NXT Data Observer	☑	Observer

Fig. 12 KIELER view of SJ program running on a Lego Mindstorms NXT

Mindstorms (LeJOS)[3] where there is some support for the Eclipse platform as well. For validation, we brought an embedded variant of SJL's predecessor SJ onto the Lego Mindstorms 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 12 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

3 http://lejos.sourceforge.net.

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.

7 Experimental results

With SJL we brought concepts borrowed from synchronous languages to Java in order to specify deterministic concurrency. In our experiments our goal was to measure the gain in predictability and efficiency of these constructs. Hence, we compared Java with synchronous concepts to Java without synchronous concepts. To illustrate the predictability and the efficiency of the SJL approach compared to Java threads, we compared the run times of the Java threads version and the SJL version of the PC example discussed in Sect. 3. We also included a benchmark of the earlier SJ version [19] to compare it with the new light-weight approach, presented in this paper.

The specification of the PC program in a synchronous language such as SyncCharts was given in Fig. 5b. Synchronous languages typically are compiled into a host language like C or Java in order to execute them. In this compilation step the deterministic scheduling order must be preserved. Traulsen et al. [28] shows a compilation from SyncCharts to SC code where this deterministic scheduling order is expressed using priorities. This idea can be reused for compiling SyncCharts to SJL. In order to be able to compare the predictability and efficiency of executions we here chose Java as the host language. Hence, the SJL version of the PC example shown in Fig. 5c can be seen as the implementation for the SyncChart version shown in Fig. 5b.

We ran all three programs on an Intel Core 2 Duo T9800 @ 2.93 Ghz machine with 8GB of RAM and a 64 Bit JVM with a variable number of ticks, that were equal to the number of data produced/observed by each implementation. So for each variable number of ticks, all implementations had the same task to fulfill.

Figure 13 shows the execution time of each implementation over the variable number of ticks and the speed up of the SJL variant over the Java threads implementation. 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 already faster (average of 1.75 times faster) compared to the Java threads version that has to struggle with more overhead due to possibly poorly scheduled executions. Moreover the SJL version is considerably faster (average of 4.5 times faster) compared to the Java threads version due to its very light-weight bit-level implementation strategy. Another, perhaps more important difference is the variability of the worst-case run time. While the Java threads version is heavily unpredictable especially when it comes to more duty, i. e., more ticks, the SJL as well as the SJ variants are much closer to a linear growth and hence more predictable. Both facts

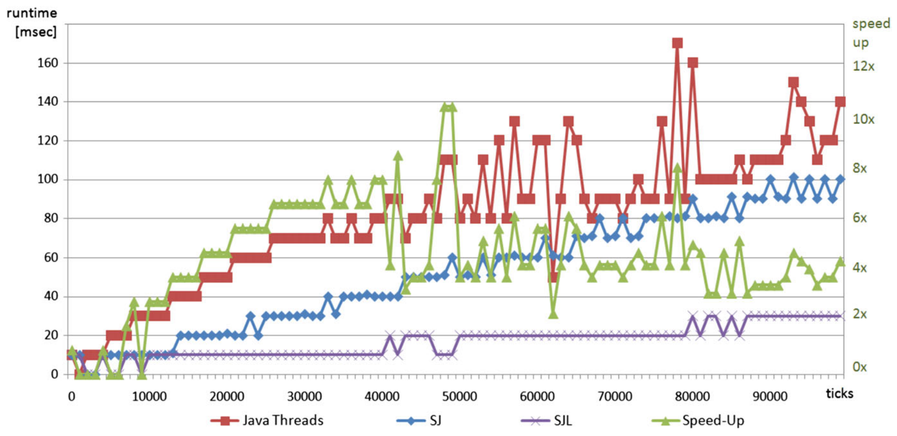


Fig. 13 Worst-case run times, SJL vs. SJ vs. standard Java threads, for the PC example

support our thesis that SJ and, even more so, SJL is much more light-weight and predictable than standard Java.

8 Conclusion and outlook

Properly synchronizing Java threads may be complex and problematic. We presented SJL as an adoption of the synchronous concepts for Java, and showed that SJL can help specifying concurrent threads in a light-weight and robust way. We also illustrated the use of preemption and how to mimic synchronous-style signals for predictable communication between concurrent threads of an SJL program. Another benefit is that such programs can run on platforms where a thread management may be too much overhead, e. g., 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 SJL programs have a more predictable run time and are typically faster than Java threads. SJL can be considered a programming language as well as a target language for code generation from more abstract models, such as SyncCharts. SJL code is close to abstract specifications, as it directly supports concepts like states and transitions. SJL permits to implement synchronous data-flow applications, see the PC example, as well as control-driven applications, see ABSWO.

We plan to exploit SJL as an automated code generation target from SyncCharts, SCCharts, 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 SJL code with visual and interactive debugging possibilities. We also plan to validate SJL and SC simulators by leveraging the Ptolemy[4] Project of the UC Berkeley.

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

References

1. Andalam S, Roop PS, Girault A (2010) Deterministic, predictable and light-weight multithreading using pret-c. In: Proceedings of the conference on design, automation and test in Europe (DATE'10). Dresden, Germany, pp 1653–1656
2. André C (1996) SyncCharts: a visual representation of reactive behaviors. Technical report RR 95–52, rev RR 96–56, I3S, Sophia-Antipolis, France
3. Axer P, Ernst R, Falk H, Girault A, Grund D, Guan N, Jonsson B, Marwedel P, Reineke J, Rochange C, Sebastian M, von Hanxleden R, Wilhelm R, Yi W (2013) Building timing predictable embedded systems. In: Proceedings of ACM transactions on embedded computing systems, (accepted)
4. Berry G (2000) The foundations of Esterel. In: Plotkin G, Stirling C, Tofte M (eds) Proof, language and interaction: essays in honour of Robin Milner. MIT Press, Cambridge
5. Bollella G, Gosling J, Brosgol BM, Dibble P (2000) The real-time specification for Java. Addison-Wesley Longman Publishing Co., Inc., Boston
6. Boussinot F (2006) Fairthreads: mixing cooperative and preemptive threads in C. Concur Comput Pract Exp 18(5):445–469
7. Caspi P, Pilaud D, Halbwachs N, Plaice JA (1987) Lustre: a declarative language for real-time programming. In: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on principles of programming languages (POPL'87), ACM, Munich, Germany, pp 178–188. doi:10.1145/41625.41641
8. Conway ME (1963) Design of a separable transition-diagram compiler. Commun ACM 6(7):396–408. doi:10.1145/366663.366704
9. Boussinot F (1991) Reactive C: an extension of C to program reactive systems. Soft Pract Exp 21(4):401–428
10. Fuhrmann H, von Hanxleden R (2010) Taming graphical modeling. In: Proceedings of the ACM/IEEE 13th international conference on model driven engineering languages and systems (MoDELS'10), LNCS, vol 6394. Springer, Berlin, Heidelberg, pp 196–210. doi:10.1007/978-3-642-16145-2_14
11. von Hanxleden R (2099) SyncCharts in C-A proposal for light-weight, deterministic concurrency. In: Proceedings of the international conference on embedded software (EMSOFT'09), ACM, Grenoble, France, pp 225–234
12. von Hanxleden R, Mendler M, Aguado J, Duderstadt B, Fuhrmann I, Motika C, Mercer S, O'Brien O (2013) 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), IEEE, Grenoble, France
13. Heinold M (2010) Synchronous Java Bachelor thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science
14. Henties T, Hunt JJ, Locke D, Nilsen K, Schoeberl M, Vitek J (2009) Java for safety-critical applications. In: 2nd international workshop on the certification of safety-critical software controlled systems (SafeCert 2009)
15. Köser N (2010) SyncCharts in C auf multicore dimploma thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science
16. Lee EA (2006) The problem with threads. IEEE Comput 39(5):33–42
17. Lickly B, Liu I, Kim S, Patel HD, Edwards SA, Lee EA (2008) Predictable programming on a precision timed architecture. In: Proceedings of compilers, architectures, and synthesis of embedded systems (CASES'08), Atlanta, GA, USA
18. Miyoshi A, Kitayama T, Tokuda H (1997) Implementation and evaluation of real-time Java threads. In: Proceedings of the 18th IEEE real-time systems symposium (RTSS'97), San Francisco, CA, USA, pp 166–175. doi:10.1109/REAL.1997.641279
19. Motika C, von Hanxleden R, Heinold M (2013) Programming deterministice reactive systems with synchronous Java (invited paper). In: Proceedings of the 9th workshop on software technologies for future embedded and ubiquitous systems (SEUS 2013), IEEE Proceedings, Paderborn, Germany
20. Nadeem M, Biglari-Abhari M, Salcic Z (2011) RJOP: a customized Java processor for reactive embedded systems. In: Proceedings of the 48th design automation conference (DAC'11), ACM, New York, NY, USA, pp 1038–1043. doi:10.1145/2024724.2024952
21. Nilsen K (1998) Adding real-time capabilities to Java. Commun ACM 41(6):49–56. doi:10.1145/276609.276619
22. Plsek A, Zhao L, Sahin VH, Tang D, Kalibera T, Vitek J (2010) 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), ACM, Prague, Czech Republic, pp 95–101. doi:10.1145/1850771.1850786

23. Schoeberl M (2006) A time predictable Java processor. In: Proceedings of the design, automation and test in Europe conference (DATE'06). Munich, Germany, pp 800–805
24. Schoeberl M (2007) 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), Springer, Santorini Island, Greece, pp 105–113. http://dl.acm.org/citation.cfm?id=1778978.1778991
25. Schoeberl M (2008) A Java processor architecture for embedded real-time systems. J Syst Arch (JSA) 54(1–2):265–286
26. Stadler L, Würthinger T, Wimmer C (2010) Efficient coroutines for the Java platform. In: Proceedings of the 8th international conference on the principles and practice of programming in Java (PPPJ'10). ACM, Vienna, Austria, pp 20–28
27. Sung M, Kim S, Park S, Chang N, Shin H (2002) Comparative performance evaluation of java threads for embedded applications: Linux thread vs. green thread. Inf Process Lett 84(4):221–225. doi:10.1016/S0020-0190(02)00286-7
28. Traulsen C, Amende T, von Hanxleden R (2011) Compiling SyncCharts to Synchronous C. In: Proceedings of the design, automation and test in Europe conference (DATE'11), IEEE, Grenoble, France, pp 563–566