Communicating Cooperatively Scheduled Processes

On the Unlikelihood of Implementing a Pure CSP Channel

KEVIN CHALMERS, Ravensbourne University London, United Kingdom
JAN BÆKGAARD PEDERSEN, University of Nevada Las Vegas, United States of America

This paper presents the modelling of the ProcessJ cooperative runtime environment for the implementation of a CSP-inspired communication channel. We used the CSP tool FDR to verify the correctness of the ProcessJ runtime and primitives, demonstrating how we have overcome a limitation in the existing ProcessJ runtime to improve behaviour. However, our work has demonstrated a limitation when trying to claim a cooperatively-scheduled channel implementation meets the abstract specification of a CSP channel. Our conclusion is that without sufficient hardware to execute all processes at once, a channel implementation cannot fully meet its specification when considering the execution environment in which the channel must operate. We are assured that the ProcessJ channel works correctly, such as other implementations including JCSP and CSO, but we are not assured that modelling can use a pure CSP channel in place of a ProcesssJ channel or other implementation when undertaking modelling due to unintended prioritisation occurring when not enough resources are available for full concurrency. Our work has demonstrated the need to consider the execution environment as it may cause behavioural issues not detected when only modelling a system in the abstract.

CCS Concepts: • **Theory of computation** → **Program verification**; • **Software and its engineering** → **Runtime environments**.

Additional Key Words and Phrases: CSP, process-orientation, ProcessJ, formal verification, cooperative scheduler, verified runtime

ACM Reference Format:
Kevin Chalmers and Jan Bækgaard Pedersen. 2025. Communicating Cooperatively Scheduled Processes: On the Unlikelihood of Implementing a Pure CSP Channel. 1, 1 (October 2025), 45 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Ensuring correct concurrent behaviour poses a significant challenge. Consider a synchronising channel that passes messages between processes. The conventional method of verifying a channel implementation often overlooks the hardware and execution context, such as schedulers, in which a concurrent system operates. This comes from an implicit assumption of the availability of resources or scheduling—eventually a waiting process will be serviced. In Hoare's Communicating Sequential Processes (CSP) [Hoa78, Hoa85, Ros98, Ros10], when an event is ready, it can happen, assuming the external environment will be ready and available for code execution when the process is ready. This is an oversimplification of how code is executed on a real computer. The execution resources/environment **must** be taken into account thus also included in the formal verification of a channel-based runtime.

Authors' Contact Information: Kevin Chalmers, Ravensbourne University London, School of Computing, Architecture, and Emerging Technologies, London, United Kingdom, k.chalmers@rave.ac.uk; Jan Bækgaard Pedersen, University of Nevada Las Vegas, Department of Computer Science, Las Vegas, NV, United States of America, matt.pedersen@unlv.edu.

ACM XXXX-XXXX/2025/10-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This paper extends our previously presented work [PC23]. In that paper, we followed the standard approach to verification using CSP—we modelled our synchronisation primitives for our runtime and language (ProcessJ) and demonstrated that they met their specifications. However, part of our goal was to understand what would happen when the cooperative scheduler of ProcessJ was made multithreaded. We needed to understand the execution environment that our primitives existed within. We came to the conclusion that not understanding and modelling the execution environment was a significant limitation in channel-based runtime verification, potentially leading to undiscovered issues. Indeed, the ProcessJ scheduler had issues—although stemming from a simpler implementation of the original scheduling system. To solve the issues, we were left with two options:

(1) Change the implementation of the synchronisation primitives to work in the execution environment.
(2) Change the execution environment so it worked correctly with the synchronisation primitives.

In this paper we focus on the second option as we were confident our synchronisation primitives were correct (we have verified them using the standard approach; see Section 2.3), but importantly we can update the ProcessJ scheduler to change the execution environment, an option not always open to implementers of concurrent runtimes.

1.1 ProcessJ

ProcessJ is a process-oriented programming language being developed at the University of Nevada Las Vegas in collaboration with the University of Roehampton. ProcessJ is a language that supports concurrency through parallel processes and synchronous non-shared channel communication. The syntax of ProcessJ closely resembles that of Java with extra keywords added to support process-orientation. The ProcessJ compiler produces Java source code which subsequently is compiled with the Java compiler; the resulting class files are then instrumented to support cooperative scheduling. The runtime and scheduler are written in Java, and the whole system ultimately runs on the JVM.

ProcessJ uses *processes* to specify the behaviour of its concurrently-executing components. A process is a series of steps that are undertaken sequentially. These steps can include the execution of other processes, possibly concurrently. Here, it is important to realise that a ProcessJ process is **not** an operating system process or thread. Though the code is executed by the CPU, the (operating system) thread that runs the ProcessJ code is what we later refer to as a *scheduler*. A ProcessJ process is an object on the JVM heap that contains the state of the process and has a *run*() method that contains the code to be run. The JVM thread that runs the code (the scheduler) simply calls this *run*() method every time it needs to run the ProcessJ process. This leads to jumping back to where the ProcessJ process yielded the last time it was run. The execution continues until a *yield* point is reached, at which time the *run*() method terminates and the scheduler is free to call the *run*() method of another ProcessJ process. All local state that would normally be on the operating system stack is now stored inside the object as fields.

To communicate between processes, ProcessJ provides synchronous CSP-like channels. Both writer/sender and receiver/reader must be present for a message exchange to happen. If the writer/sender arrives first, it must wait for the receiver/reader and vice-versa.

This style of concurrency requires three fundamental functions to specify system behaviour:

- The ability to execute two or more processes *concurrently* (*in parallel*).
- The ability to communicate between processes via a *channel*.
- The ability to *select* between two or more channel communications to define behaviour (*choice*).

Below is an example of a small ProcessJ program that runs two processes concurrently using the **par** keyword[1]. The two processes communicate an integer value 42 (passed in as a parameter) across a channel using the keywords **read** and **write**:

```
void reader(chan<int>.read in) {
  int v;
  v = in.read();              // read a value from the channel
  println(v);
}

void writer(chan<int>.write out, int val) {
  out.write(val);             // write the value val on the channel
}

void main(string args[]) {
  chan<int> c;                // declare a channel carrying ints
  par {                       // run in parallel the following two processes:
    reader(c.read);           // pass reading end of channel c to reader proc
    writer(c.write, 42);      // pass writing end of channel c and the value 42 to writer proc
  }
}
```

The ProcessJ type **chan**<*int*> denotes a *channel type* that carries integer values. Furthermore, **chan**<*int*>.**read** and **chan**<*int*>.**write** denote the *channel end types* of a channel carrying integer values. **read** refers to the reading end of such a channel. **write** refers to the writing end.

The ProcessJ scheduler is a cooperative scheduler that operates with a single run queue of processes. In its simplest form, it looks like this:

```
Queue<Process> runQueue;
...
// enqueue 1 or more processes to run ...
while (!runQueue.isEmpty()) {
  Process p = runQueue.dequeue();
  if (p.ready())
    p.run();
  if (!p.terminated())
    runQueue.enqueue(p);
}
```

Note, what we refer to as a *scheduler* (as shown above) is perhaps better understood as a runner. This is because there is no scheduling policy enforced, but processes are simply removed in order from the queue and run by having their *run*() method invoked. There may be multiple runners (schedulers) all sharing the same process queue.

In cooperative scheduling, active processes must *yield* control rather than have it *preemptively* taken by a scheduler. The ProcessJ scheduler is a thread that runs application code. It consists of a single loop that repeatedly dequeues a process, and if the process is ready, runs it. Once the process yields control back to the scheduler, or if the process is not ready to run, it is checked for termination, and if not terminated the process is placed at the back of the run queue. An important

[1]The process with the **par** block will itself block until all processes in the block have terminated.

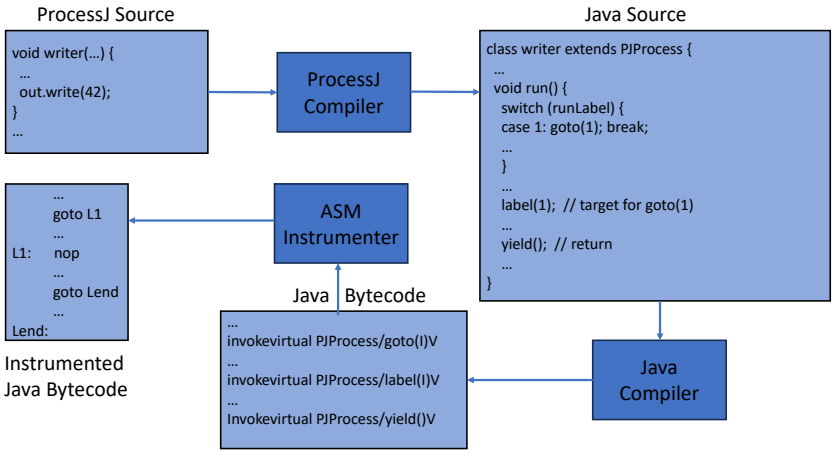

Fig. 1. The ProcessJ Toolchain.

observation about the way this single queue system works is that the run queue contains both ready and not ready processes. This is the scheduler we used in [PC23].

1.1.1 The ProcessJ Tool Chain. We touch upon the scheduling part of ProcessJ Processes in Section 2.2 and show the code generated for the ProcessJ example shown earlier in this section, but it is worth understanding the ProcessJ tool chain.

Figure 1 illustrates how ProcessJ source code becomes runnable Java bytecode:

- We start with a snippet of ProcessJ source code.
- The ProcessJ source is read as input by the ProcessJ compiler. The code generation module of the ProcessJ compiler produces Java source code. This Java code contains a number of "dummy" methods (*goto*, *label*, and *yield*) all declared in the class *PJProcess*. Their implementation is just an empty block. That is, if they are ever called nothing would happen, but the Java program correctly compiles to bytecode.
- Next, the Java compiler is invoked and it produces Java bytecode. Note, the calls to the dummy methods are still in this code.
- Finally, we replace the dummy method calls with appropriate bytecode:
  - *label(x)* becomes a `nop` instruction, but the address of this instruction is noted.
  - *goto(x)* becomes `goto Lx` where `Lx` is the the noted address of the `nop` instruction from the previous item.
  - *yield()* becomes `goto Lend`, where `Lend` is the address of a `nop` instruction at the very end of the code.
- The resulting Java bytecode file is now compatible with a cooperative scheduler that simply calls the *run*() method of the object representing the process. The process has code (the switch statement) at the top to correctly jump to where execution left off the last time it was run, and yield points have been added (by the code generation) to the appropriate places in the code.

More information about the code generation as well as the instrumentation can be found in [Cis18].

1.2 Limitations of the Cooperative Scheduler

Let us consider what consequences we may have once we extend our system to run multiple schedulers concurrently. First, observe that with a single scheduler, if no processes in the run queue are ready to run, a deadlock has occurred. Progress is impossible as there are no ready-to-run processes to drive the system forward by changing the *ready* flags of any of the not-ready processes in the run queue. This is a deadlock from the perspective of the application being run, as no processes will ever again become ready to run. From a verification point of view, this is exhibited as a livelock: non-ready processes keep getting removed from and put back onto the run queue. However, if there is at least one process ready to run, it will eventually move to the front of the queue and be executed, potentially altering the readiness status of other processes.

When multiple schedulers are involved, there is a scenario where a scheduler might keep removing non-ready processes from the shared queue endlessly. For instance, in a scenario where a reader and a writer are in continuous communication, it is conceivable (and can be shown in a simulated system trace) that one of the schedulers will manage both the reader and the writer alternately, while the other scheduler consistently deals with dequeuing and enqueuing non-ready processes. A possible system trace illustrating this situation is as follows:

$$\langle \mathit{dequeue}.P1, \mathit{write}, \mathit{yield}.P1, \mathit{enqueue}.P1, \mathit{dequeue}.P2, \mathit{dequeue}.P1, \mathit{enqueue}.P1, \mathit{dequeue}.P1, \ldots \rangle$$

where the identifiers $P1$ and $P2$ represent the process identifiers to be executed. Until $P2$ makes progress (it has only been dequeued), dequeue and enqueue of $P1$ will continue as $P1$ is not ready. To better illustrate this, let us split this trace across the two active scheduler threads:

$$\begin{array}{lccccccc} T1 = & \mathit{dequeue}.P1 & \mathit{write} & \mathit{yield}.P1 & \mathit{enqueue}.P1 & \mathit{dequeue}.P2 & \ldots & \ldots \\ T2 = & \ldots & \ldots & \ldots & \ldots & \mathit{dequeue}.P1 & \mathit{enqueue}.P1 & \mathit{dequeue}.P1 \end{array} \quad \ldots \quad \ldots$$

The issue becomes more apparent in a scenario where there are more schedulers than processes in the system. In such a case, a likely sequence of events could involve a continuous cycle of dequeueing and enqueueing operations. This straightforward setup of a cooperative scheduler was demonstrated in our earlier work [PC23], leading us to the following finding: "Changing how processes are added and removed from the queue, and managing their state, should enable us to remove the spinning (livelock), but careful consideration is required to ensure that verification can still occur."

In this paper we make these changes to the scheduler and the runtime system; we then perform the same verifications (i.e., using the same formal CSP models as the specification) to determine if we indeed managed to solve the problems discovered in [PC23]. The changes we propose in this paper include the following:

- Remove non-ready processes from the run queue. They do not need to be held in the queue as they are now held in the channel on which they are blocked.
- Make the run queue blocking (i.e., the dequeue operation will be blocking if the queue is empty).
- Add a *running* field to the process meta-data (similar to the *ready* field).
- Add a shared scheduling interface (scheduler manager).

The details about the implementation of this new scheduler are presented in Section 4. In [PC23] we started with an implementation of the runtime system of ProcessJ and then proceeded to create CSP models of the implementation. These models were then verified against specifications (from Welch and Martin [WM00]). In this paper we have no ProcessJ implementation, but start with a CSP model of the proposed design, and verify against the same Welch and Martin specification as we used previously. Once we know the design is correct, we can turn to changing the Java implementation of the scheduler and the runtime of ProcessJ.

In [PW18] a *Concurrency and Verification Workflow* diagram and approach are presented showing the typical workflow between the problem domain and a verified working implementation. In particular, the duality of a formal specification and an actual implementation is pointed out. In [PC23] we started with an implementation and developed the formal model to determine if the implementation was correct. In this paper we work in the opposite direction: here, we start with a formal model and once it has been verified, the implementation can commence.

1.3 Contributions

This research makes three key contributions. Firstly, this work has developed a verified cooperatively scheduled runtime for our process-oriented language, ProcessJ. This completes a verification journey that has always demonstrated that our channel implementation was correct, but we have now demonstrated that correctness when run in a multiprocessor environment. There are limitations to our verification due to the resources made available to the runtime, but this in itself is a further contribution.

Our second key contribution is the identification of limitations in CSP-inspired channel implementations. Although our context is within cooperative scheduling—where we have formally modelled the limitation—we also provide empirical evidence and discussion of this limitation for the preemptively scheduled JCSP library. Our argument is that without enough resources (i.e., hardware processors) to enable all active processes to make progress in parallel, a channel concurrency primitive mapping to CSP is still a challenge. The limitation comes from unintended prioritisation at the end of a channel communication, which can lead to potential starvation issues that would propagate through a system.

The final key contribution is our models working with the execution environment that our concurrency runtime operates within. Without our consideration of the execution environment, we would not have discovered flaws in the current implementation of the ProcessJ scheduling system which would have occurred when transitioning to a multithreaded version. We believe our approach to modelling the execution environment is unique and enables us to question implementations in operation. As we have verified our implementation (with limitations) in this context, we are more assured of its correct execution.

1.4 Definitions

To avoid confusion when reading the paper, we provide this definition section that explains the meaning attributed to a number of commonly used concepts that may be slightly different in the context of this paper:

- A **process**: In ProcessJ, a process is an entity that can be executed by a scheduler/runner (see the next bullet point). For ProcessJ, a process and all its state is an object that contains the state in fields and the code to be executed as a single method called *run*(). A process is a series of sequential steps. An individual step can be a concurrent execution of multiple other processes.
- **Yielding**: When a process decides to yield, its *run*() method is terminated. Yielding means giving control back to the scheduler, which will find a different process/object that is ready to run and call its *run*() method. As all state is kept in fields of the object there is no need to save any stack state. A process's *run*() method contains code that will jump to the latest yield point when called again. This is done through instrumentation of the bytecode.
- A **scheduler**: The typical definition of a scheduler is a module in the operating system that is in charge of selecting the next job that is to be run by the hardware. In ProcessJ (and in this paper) a scheduler is more of a *runner*; that is, a scheduler will interact with the

run queue and more importantly, it is the scheduler that **runs** the code. Thus, if we want multiple execution units, we add more schedulers. ProcessJ has no other scheduling policy than a FIFO queue of ready processes.

- A **thread** is typically defined as a sub-execution unit of a process. For almost all purposes, whether we refer to a thread or a process (from an operating system point-of-view) there is no difference. The technical differences do not matter—all that matters is that they are both independent execution units.

1.5 Breakdown of the Paper

In Section 2 we consider cooperative scheduling in general, and we discuss cooperative scheduling in the ProcessJ runtime in particular. In addition, we describe the verification results that we have achieved so far. Section 3 introduces CSP and FDR. In Section 4 we present the CSP for the new scheduler and the changes made to the runtime system. In Section 5 we present the specifications for both channel and alternation that we will be using in the verification phase of the paper. In Sections 6 and 7 we combine the new scheduler with the channel and choice implementations from [PC23] and redo the verification. In Section 8 we consider a number of issues concerned with scheduling. Section 9 summarises the results, and, finally, Section 10 wraps up with future work and possible extensions.

The CSP models presented in this paper can be found at https://github.com/kevin-chalmers/processj-csp/tree/main/non-spinning-scheduler.

2 Background and Related Work

In this section we start with some background information about cooperative scheduling and then consider the results from [PC23].

2.1 Cooperative Scheduling and Multitasking

To support multi-processing, a runtime typically takes one of two scheduling approaches:

- *preemptive scheduling*, where processes are allocated processor time-slices managed centrally (e.g., via the operating system).
- *cooperative scheduling*, where processes decide when to give up (*yield*) the processor.

If all processes have the same priority, preemptive scheduling guarantees a ready process will eventually be allocated processor time. Cooperative scheduling does not guarantee processor time as a process may use resources indefinitely—it is for the process to give up the processor. When it does give up the processor, a method is required to determine the next process to service. Typically, and in the case of ProcessJ, a queue of processes is maintained.

On a single processor system, a simple approach to such cooperative scheduling can be used—the next (ready) process is continuously dequeued from the queue and executed until the queue is empty. In a multiprocessor system, consideration of cross-core synchronisation is required. Often, as in the case of ProcessJ, we rely on preemptive locking mechanisms provided by the operating system to ensure correct concurrent behaviour. We therefore have to account for the cooperative scheduling (via yield points) within the preemptive scheduling (via locking) when designing a system. Thus, for ProcessJ runtime verification, we **must** incorporate the implementation of the cooperative scheduling environment.

Cooperative scheduling offers some advantages over preemptive scheduling. With cooperative scheduling, as we use in ProcessJ, processes are stackless (remember, a ProcessJ process is not a thread, but an object with data that is maintained to enable scheduling). Process local data is stored as object data. Therefore, a process is incredibly lightweight. Previous ProcessJ publications [SP16]

have demonstrated hundreds of millions of processes in operation. This compares to preemptive thread-based libraries, such as JCSP, managing only a few thousand processes. The limitation in such libraries is often due to large stack sizes (often greater than a megabyte) that each thread utilises as a stack. ProcessJ can support at least four orders of magnitude greater system scale.

The lightweight processes of ProcessJ also promote fast context switching [SP16]. As no preemptive thread swapping is required, context switches, and thus channel communication time, is fast. A thread provided by the operating system, such as in standard Java and used by JCSP, must save a register record and load a new one onto the CPU with each context switch. If this switch is cross-core, the impact is greater due to the stack swapping between caches.

Cooperative scheduling currently has limited popularity due to the programmer typically having to manage scheduling. Operating systems favour preemptive scheduling (although cooperative scheduling was dominant prior to modern 32-bit operating systems). Threading libraries in programming languages typically use operating system provided primitives. There are some exceptions (e.g., the Win32 fiber library), but preemptive scheduling is the *de facto* approach to threading. No mainstream language directly supports cooperative scheduling as part of its standard library.

We believe that hiding cooperative scheduling (via communication), as in ProcessJ, alleviates some of the challenges in managing such scheduling. Indeed, until Go 1.10, the scheduler was a mix of cooperative and preemptive. The current Go scheduler is preemptive, but Go is a compiled language that runs directly on hardware. As the aim of ProcessJ is to run within the JVM for portability, we are limited in how fast and lightweight processes are supported. occam-π also featured a cooperative scheduler [RSB09] which was likewise hidden from the programmer.

In contrast, cooperative *multitasking* has a growing popularity, primarily from the *async/await* pattern. Coroutines [MI09] have gained popularity in languages ranging from JavaScript and Python to C++ and Rust. Indeed, this has led to the discussion of new concurrency paradigms such as structured concurrency [CY23] which have influenced thinking in Kotlin and Swift. However, coroutines are not scheduled as in the case of a cooperative runtime as used in ProcessJ.

The ProcessJ scheduler is fairly unique. Other lightweight concurrency languages, such as Go and Erlang, typically use preemptive scheduling [CR22, CT09]. Indeed, the only similar runtime approach—hiding cooperation in synchronisation—we have found is in Stackless Python [Tis00].

2.2 Cooperative Scheduling in ProcessJ

In cooperative scheduling, a process is responsible for yielding control of computational resources. In the Java code generated from a ProcessJ program, this is achieved at various points by inserting *yield* points in connection with synchronisation (e.g., communication, choice, and other synchronisation primitives like barriers and timers). Yielding is achieved by a jump to the end of the run method code. These jumps are created during an instrumentation phase (see Section 1.1.1) of the class files after the Java compiler has run. In addition, code is inserted at the start of all *run*() methods to correctly jump back to where the process last yielded when it is run again. A yield also sets the next run label—the location where the execution resumes. It may also set the process not ready by setting the *ready* field of the process to false.

A ProcessJ procedure is compiled to a Java class. The code of the ProcessJ procedure is produced in the *run*() method of the class. In order for all the locals and parameters of a ProcessJ procedure to survive between invocations, they are turned into fields in the generated class. Thus, a ProcessJ process is stackless to the point that when a process yields there is no need for maintaining a stack as all the data needed by a process is kept as fields in an object on the heap.

The result of the compilation of the ProcessJ *reader* procedure from the introduction is the following Java class (which shows the general shape of the code produced for a ProcessJ procedure):

```
class reader extends PJProcess {
  // params and locals are turned into fields here
  protected PJOne2OneChannel in;
  protected int v;

  // constructor to set the fields representing the params.
  public reader(PJOne2OneChannel in) {
    this.in = in;
  }

  public void run() {
    // generated code to jump to the right place in the code
    switch (this.runLabel) {
    case 0: break;
    case 1: resume(1); break;          // resume(1) becomes goto L1
    case 2: resume(2); break;          // resume(2) becomes goto L2
    default: break;
    }

    // translated ProcessJ code with yields
    if (!in.isReadyToRead(this)) {     // if a channel is not ready to be read
      this.runLabel = 1;               // resume at label(1) (L1) next time.
      this.yield();                    // no writer is ready yet, so yield
    }
    label(1);                          // L1 is the address of this invocation
    v = in.read(this);                 // perform the actual read
    this.runLabel = 2;                 // resume at label(2) (L2) next time.
    this.yield();                      // courtesy yield

    label(2);                          // L2 is the address of this invocation

    // Here goes the code for the println call.
  }
}
```

Note, the *isReadyToRead*() method sets the process's *ready* field to false, if the channel does not have a value ready to communicate (i.e., if there is no writer present). Also, the fields *runLabel* and *ready* are found in the super class *PJProcess*.

The *label*(x) method invocations are removed but their addresses are recorded and associated with the number x—indicated by the labels L_i in the code above. A *resume*(y) method invocation is replaced by a goto instruction that jumps to the address associated with the corresponding *label*(y) invocation.

2.3 Related Work

In order to appreciate the approach we have taken in this paper to verify a *scheduled* system, it is important to understand the *standard* approach to verification of channel-based systems using CSP and FDR. In the standard approach, no runtime system, execution system, hardware etc.

is modelled. When modelling such systems with CSP, the assumption is that when an event is ready that resources will eventually become available. However, this assumption also includes the situation where the resource is available at exactly the point where the event has become ready. This paper in effect is controlling for this condition of resource availability. As we have a model of our scheduler, we can explicitly see the impact when resources are not available at a given time, testing the assumption of eventual resource availability under scheduling conditions.

The standard assumption means that events in the formal model of such a system can possibly happen whenever the system is ready to perform the event. No attention is paid to external resources or the execution environment—the verification is in the abstract rather than in a concrete environment. However, we argue that such an approach does not model what actually happens when a channel-based system is brought into service. In essence, the standard way of modelling such systems assumes that a ready event will remain available until scheduled. However, this also assumes the extreme case—that an event may immediately occur, implying that resources are available to execute that particular event when it immediately becomes available. Consider a process that is at the end of a queue of ready processes. It must wait to be scheduled—its events will not be immediately available. By modelling the execution environment as we do in this and previous papers, we are much closer to the behaviour of a real running implementation of a channel. In previous work, as the extreme case (no scheduling considerations taken) was not precluded, then certain behaviours were met by implementations that enabled meeting of pure channel specifications. Our work in ProcessJ is to consider the limitations when precluding the extreme case through the use of scheduling to better understand the behaviour of channel primitives in implementation.

Mutual-exclusion is long-solved (e.g., see [Lam87]). Model checking concurrent algorithms often begins with the assumption that multi-processor behaviour is typically managed via a preemptive scheduler. Preemptive scheduling is not a panacea for ensuring events happen in a fair manner or even that they will happen in a way enabling all potential behaviours (for example, see [Reg02]). Atomic operations introduce new challenges for mutual exclusion, but again, the preemptive scheduler enables certain assumptions (e.g., see [MS95]). The assumption of preemptive scheduling has been successfully applied to model check synchronous channel communication [WM00, Low19] in JCSP [WBM+10] and Communicating Scala Objects (CSO) [Suf08]. JCSP and CSO both rely on preemptive scheduling provided via the JVM (which in reality is normally the operating system). The ProcessJ scheduler more closely emulates the occam-π multi-processor cooperative scheduler defined by Ritson et. al. [RSB09].

In all the research we have encountered in the literature where the authors set out to prove correctness of a channel primitive, we find little consideration of the *execution environment*'s behaviours. For example, a known issue in Java concurrent code is spurious wakeups [Ora]. A spurious wakeup happens when a thread waiting is woken up (by the operating system) but the condition that sent the thread to sleep is still true. That is, the thread was woken up but the condition has not changed. We will not go into why spurious wakeups happen, but note that the standard trick to solve this problem is replacing the if statement that sent the thread to sleep with a while statement that checks the same condition:

```
  if ( condition )
      wait();
becomes
  while ( condition )
      wait();
```

However, from a modelling perspective, the code will be divergent as—unlikely as it may be—the while loop may loop forever. Spurious wakeups feature both in the core Linux pThread library as well as in Windows threads.

JCSP now has code to mitigate the issue of spurious wakeups, but this has never been formally modelled. Therefore, the JCSP verification is limited in its scope. No code runs itself, it is always executed within an execution environment: threads, processes, schedulers, etc.

2.4 Verification So Far

In [PC19] we took the first step on our verification journey. We started out approaching verification of channel communication for ProcessJ much in the same way as Welch and Martin [WM00]. We succeeded in verifying correctness of channel communication. Like Welch and Martin, we applied the *standard approach* of formal verification: we translated our implementation into CSP and then used the specification for channel communication developed by Welch and Martin.

In [PC23] we added formal verification of choice using a slightly optimised version of a specification, again, from Welch and Martin [WM00], and obtained the same results. However, more profoundly, we realised that none of the formal verification exercises that have been published so far took into account the execution environment in which processes run. That is, the processes themselves were modelled but not the environment in which they execute. This environment could include runtime systems, schedulers, and hardware. Since ProcessJ is cooperatively scheduled, modelling the interaction and behaviour for the scheduler becomes very important. It *is the scheduler* that runs a process's code. If we do not take into account the behaviour of the scheduler(s) and their interaction with the processes, then how could we ever know for sure that the verification is correct? We redid our analyses, but with the scheduler's implementation taken into account. These results were promising, but we did not quite manage to replicate the results when considering the channel communication or the choice **without** the scheduler. The major problem was divergence in the scheduler. This divergence was caused by the run queue containing processes that were not ready to be run. This, in turn, could cause one of more schedulers to continuously engage in enqueue and dequeue events when a not-ready process was removed from a queue that contained no processes that were ready to run (other schedulers would be off running code independently).

The solution was to develop a better scheduler (modify the execution environment) to one that does not cause divergence due to the not-ready processes in the run queue. This paper is the description of such an approach and the results on the verification of the channel communication as well as choice using CSP specification models developed in [PC23] but with new implementation models.

3 Communicating Sequential Processes

Communicating Sequential Processes (CSP) [Hoa78, Hoa85, Ros98, Ros10] is a process algebra that allows the specification of a concurrent system via *processes* and *events*. A process is an abstract component defined by the events in which it wishes to engage.

Events are *atomic*, *synchronous*, and *instantaneous*. An event cannot be divided, and it causes all synchronising processes to wait until the event occurs. The occurrence of the event is immediate when executed by the system. The CSP model defines which events processes are willing to synchronise upon at different stages of their execution.

3.1 Simple Processes

The simplest processes are STOP, which engages in no events and will not terminate, and SKIP, which can engage in the *termination event* ✓ and then terminate.

3.2 Prefixing

Events are added to a process using the *prefix* operator ($\rightarrow$). For example, the process $P = x \rightarrow$ STOP engages in event x, then stops. A process definition must end by executing another process definition; the general form is *Process = event* $\rightarrow$ *Process'*. Processes can be recursive (e.g., $P = x \rightarrow P$).

3.3 Choice

CSP has *choice* operators. The three most frequently used choice types are *external (or deterministic) choice*, *internal (or non-deterministic) choice*, and *prefix choice*. A process can exhibit alternate behaviour when engaging in a choice.

Given two processes P and Q, the definition $P \,\Box\, Q$ is a process that will behave as either P or Q, depending on the first event to be offered by the environment; it is called an *external choice*. For example, the process:

$P = (a \rightarrow Q) \,\Box\, (b \rightarrow R)$

is willing to accept a, then behave as Q, or accept b, then behave as R. If both a and b are available, then the system may choose either.

An internal choice is represented by $\sqcap$. A process $P \sqcap Q$ can behave as either P or Q without considering the external environment. The choice is internal to the process.

Both external choice and internal choice can operate across a set of events. If $E = \{e_1, \ldots, e_n\}$ is a set of events, then we can write $\Box\, a \leftarrow E \bullet a \rightarrow P$, which expands to $e_1 \rightarrow P \,\Box\, e_2 \rightarrow P \,\Box \cdots \Box\, e_n \rightarrow P$, and $\sqcap\, a \leftarrow E \bullet a \rightarrow P$, which expands to $e_1 \rightarrow P \sqcap e_2 \rightarrow P \sqcap \cdots \sqcap e_n \rightarrow P$ (where a does not appear as an event in P).

With prefix choice we can define an event and a parameter. If we define a set of events as $\{c.v \mid v \in \mathit{Values}\}$, we can consider c as a channel willing to communicate a value v. We can then define input and output operations (? and !, respectively) to allow communication and binding of variables. This is simply a shorthand due to the following identities:

$c!v \rightarrow P \equiv c.v \rightarrow P$

$c?x \rightarrow P \equiv \Box\, x \leftarrow \mathit{Values} \bullet c.x \rightarrow P$

where *Values* is a set.

In addition, CSP supports *if* statements. *if* follows the standard functional model with each branch having to end in another process definition. It is therefore common to write:

if (b) then
  P
else
  Q

The semantics of $c?v \rightarrow$ (if $(v == x)$ then P else Q) in CSP is equivalent to $(c.x \rightarrow P) \,\Box\, (\Box\, v \leftarrow X \backslash \{x\} \bullet c.v \rightarrow Q)$[2].

3.3.1 Guards. It is possible to control whether an alternation branch is enabled or disabled (i.e., whether the branch will even be considered). This is done by placing a Boolean expression (a guard) and a & before the branch. For example:

e_1 & $c?x \rightarrow \ldots \;\Box\; e_2$ & $d?x \rightarrow \ldots$

If the Boolean expression e_1 is true and the Boolean expression e_2 is false, only the $c?x$ branch will be considered. Similar, if only e_2 is true, only $d?x$ will be considered; only if both e_1 and e_2 are true will both branches be considered in the alternation.

[2]Assuming that v does not appear in P.

3.4 Process Composition

Processes can be combined via *parallel* and *sequential* composition. There are three forms of parallel composition.

$P[|\ A\ |]Q$ defines two processes with a *synchronisation set A*. For the events in A, they can only occur if offered by both P and Q at the same time (i.e., synchronously). Any event not in A will not synchronise between P and Q. For example, $(a \rightarrow b \rightarrow P)[|\ \{b\}\ |](b \rightarrow c \rightarrow Q)$ has two processes synchronising on b. Thus, a must happen, before both processes execute b and then c is performed. If the synchronisation set also contained c, then the right-hand side would not be able to perform this event unless the left-hand side also offered it; that is, P must offer c.

$P[A\ ||\ B]Q$ defines two processes that can only perform events in their respective *alphabets*; P with alphabet A and Q with alphabet B. P and Q must also synchronise on any events in the set $A \cap B$. For example, $(a \rightarrow b \rightarrow P)[\{a, b\}\ ||\ \{b, c\}](b \rightarrow c \rightarrow Q)$ has two processes synchronising on the intersection alphabet $\{b\}$. Thus, a must happen, before both processes execute b and then c is performed. If the alphabet on the left-hand side did not contain a, then system would deadlock as a could not be performed.

Two processes can also *interleave*: $P\ |||\ Q$. This means that P and Q execute in parallel but *do not* synchronise on shared events. Returning to our previous example, $(a \rightarrow b \rightarrow P)\ |||\ (b \rightarrow c \rightarrow Q)$ may accept a or b first. The event b is only ever performed by one process at a time.

Sequential composition is denoted as $P\ ;\ Q$. This means after P has terminated, Q is performed. Sequential composition operates on processes, not events. Therefore, it allows the definition of behaviour as a sequential composition of discrete process definitions.

3.5 Traces and Hiding

The set of all externally observed event sequences of a process is referred to as the *trace set*. For example, for a process $P = a \rightarrow P$ the shortest trace observable is the empty trace $\langle\rangle$. If and when P engages on the event a for the first time we can observe this external event and we now have the trace $\langle a\rangle$. The second time P engages on a we can observe the trace $\langle a, a\rangle$, and we say that the traces of P are $traces(P) = \{\langle\rangle, \langle a\rangle, \langle a, a\rangle, \ldots\}$. For a process $Q = a \rightarrow b \rightarrow Q$ has a trace-set $\{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle a, b, a\rangle, \ldots\}$.

Observable behaviour of a process can be concealed via the hiding operator \. Hidden events are replaced by τ and are ignored when comparing observable traces. For example, $(a \rightarrow b \rightarrow a \rightarrow \text{SKIP})\backslash\{a\}$ has traces $\{\langle\rangle, \langle b\rangle, \langle b, \checkmark\rangle\}$ as τs are not observable, and $\checkmark$ represents termination. Similarly, $traces(P \setminus \{a\}) = \{\langle\rangle\}$ and $traces(Q \setminus \{a\}) = \{\langle\rangle, \langle b\rangle, \langle b, b\rangle, \ldots\}$.

3.6 Models

CSP defines three models of behaviour that can be used to analyse systems: the *traces* model, the *failures* model, and the *failures/divergence* model. Each model subsequently analyses further behaviours of a system to establish refinement.

3.6.1 The Traces Model. The **traces model** comes from the externally observed behaviour of a system. If the set of traces of a system *implementation* Q is a subset of the set of traces of a system *specification* P (i.e., $traces(Q) \subseteq traces(P)$), we state that $P \sqsubseteq_T Q$ (Q *trace refines* P).

A different way of thinking about this is as follows: in a refinement test *Specification* $\sqsubseteq_T$ *Implementation*, *Specification* can be thought of as a specification that defines what traces are allowable; therefore, the refinement test asks whether *Implementation* has only allowable traces.

Hiding events allows us to analyse the external behaviour of a process, thus allowing assertions such as $P \sqsubseteq_T Q \wedge Q \sqsubseteq_T P$, where P is a simple specification and Q a complex implementation. An implementation may contain events that are not part of the specification. These are events

required to model the implementation of the system rather than just a specification of behaviour. We therefore hide internal events of an implementation to enable refinement checking.

3.6.2 The Stable Failures Model. The **stable failures model** [Ros98, Ros10] deals with events that a process may refuse to engage in after a sequence of events occur. A stable state is one where a process P cannot make internal progress (including no τ events) and must engage externally. A refusal is a set of events a process cannot engage with when in a stable state.

The stable failures model overcomes the limitation of comparing traces. For example, $(P = a \rightarrow P) \sqsubseteq_T ((P = a \rightarrow P) \sqcap \text{STOP}))$, the right-hand definition may non-deterministically refuse to accept any events. A *failure* is a pair (s, X) where s is a trace of a process P, and X is the set of events that can be refused after P has performed the trace s. Stating that $P \sqsubseteq_F Q$ means that whenever Q refuses to perform a set of events, P can do likewise. More formally, $P \sqsubseteq_F Q \Leftrightarrow \mathit{failures}(Q) \subseteq \mathit{failures}(P)$. Therefore, a specification in the stable failures model defines which failures an implementation is allowed to have.

3.6.3 The Failures-Divergences Model. The final model is the **failures-divergences model**. Divergences deal with potential livelock scenarios where a process can continually perform internal events (denoted by τ) and not progress in its externally observed behaviour. For example, consider the following two processes:

$P = a \rightarrow \text{STOP}$

$Q = (a \rightarrow \text{STOP}) \sqcap \text{DIV}$

where DIV is a process that immediately diverges. Although $P \sqsubseteq_T Q$ and $P \sqsubseteq_F Q$ (and vice-versa), Q can continuously not accept a and just perform τ. The final refinement check $\sqsubseteq_{FD}$ allows processes to be compared in this circumstance. Formally, we consider both the failures and the divergences of a process P as a pair: $(\mathit{failures}_\perp(P), \mathit{divergences}(P))$ and now define refinement in the failures/divergence model as follows: $P \sqsubseteq_{FD} Q \Leftrightarrow \mathit{failures}_\perp(Q) \subseteq \mathit{failures}_\perp(P) \wedge \mathit{divergence}(Q) \subseteq \mathit{divergences}(P)$. $\mathit{failures}_\perp(P)$ is defined as $\mathit{failures}(P) \cup \{(s, X) \mid s \in \mathit{divergences}(P)\}$. That is, the set of traces leading to a divergent set are added to the existing set of stable failures to create an extended failures set.

It is worth noting that, if both the specification and implementation are divergence free, then analysis only needs to demonstrate equivalence in the stable failures model. Indeed, for our verification purposes, both the specification and implementation are indeed divergence free. See Section 5.

3.7 FDR

FDR [GRABR14] is a refinement checker that can check various assertions about CSP processes written in CSP_M (a machine readable version of CSP). FDR is used to analyse CSP models of systems and to test them against specifications (also written in CSP). FDR allows processes to be refinement checked via the three models shown in the previous section.

FDR can also check a system for determinism and divergence freedom. In the stable failures model, FDR "considers a process P to be deterministic providing no witness to non-determinism exists, where a witness consists of a trace *tr* and an event a such that P can both accept and refuse a after *tr*." [FDR]. In the failures-divergences model, the process must also be divergent free.

3.8 Modelling State

All the ProcessJ runtime classes have member state variables which are accessible (read and write) by other classes. It is therefore necessary to model such shared state variables in CSP. In this section we present CSP that can be used to model general state which can be *set* and *read*. A state variable can be modelled in CSP via a process. The process maintains the current value of the state variable, providing events to either *load* the current value of the variable, or *store* a new value of the variable.

We can define a generic process to model a state variable (*VARIABLE*) with alphabet $\alpha VARIABLE$ as follows:

datatype *Operations* = *load* | *store*

VARIABLE(*var*, *val*) =
  var.load!val → *VARIABLE*(*var*, *val*) − *load* the value
  □
  var.store?val → *VARIABLE*(*var*, *val*) − *store* the value

$\alpha VARIABLE(var, T) = \{ var.o.v \mid o \leftarrow Operations, v \leftarrow T \}$

VARIABLE takes two parameters: *var* is the channel used to communicate between the variable and the environment. *val* is the current value of the variable. In the alphabet $\alpha VARIABLE$, *val* has type T, which is provided for the given variable by the system specifier.

For example, each ProcessJ process has a Boolean *ready* flag (stored as a field in an object), and we can define a channel for communicating with a variable representing such a flag as follows:

channel *ready* : *Processes.Operations.Bool*

The channel is called *ready* and communicates a triple: a process identifier (*pid*), an operation (*load* or *store*), and a Boolean value (*true* or *false*). We then define a *VARIABLE* process for each p in *Processes*.

4 Modelling a Non-Spinning Scheduler

In our previous work [PC23], we modelled what would happen when moving from a single-threaded scheduler to a multi-threaded scheduler, determining the problems that this would introduce while ensuring the system itself did not deadlock. The key issue faced was how the process queue was managed. In our previous work, processes were immediately put back into the process queue and the scheduling processes would check each process in the queue, looking for a ready one. This introduced divergence—the scheduling processes would loop continuously searching for a ready process. To solve this problem, we add an additional flag to the process meta-data—*running*—that allows distinguishing between processes that are running and processes that are ready via the *ready* flag. We also introduce scheduling and descheduling algorithms to ensure correct behaviour.

Furthermore, in our previous work, our models used a fixed schedule of process identifiers used to initialise the run queue. The new model lets each process schedule itself by sending its process identifier to the scheduling manager, which will add it to the run queue.

4.1 Scheduler Architecture

We are now ready to develop the CSP for the new *implementation*. We start with the CSP for the new scheduler. The overall scheduler architecture can be seen in Figure 2. We will describe each of these components in turn over the following subsections. In Figure 2 we use the following alphabets and synchronisation sets:

- $\alpha RUNQUEUE$ is the natural alphabet of the *QUEUE* process (see Section 4.1.1).
- $\alpha PROCESSES$ is the natural alphabet of the *VARIABLE* processes used for the the processes meta-data (see Section 4.1.3).
- $\alpha SCHEDULING$ is the difference of the union of the natural alphabets of the *SCHEDULER* processes and the *SCHEDULE_MANAGER* (see Section 4.1.2) and the natural alphabet of the *QUEUE*.

[3]Note, the *PROCESSES* part represents the *meta-data* for the processes needed by the scheduler, not the actual application processes. They must be run in parallel to the scheduler system.

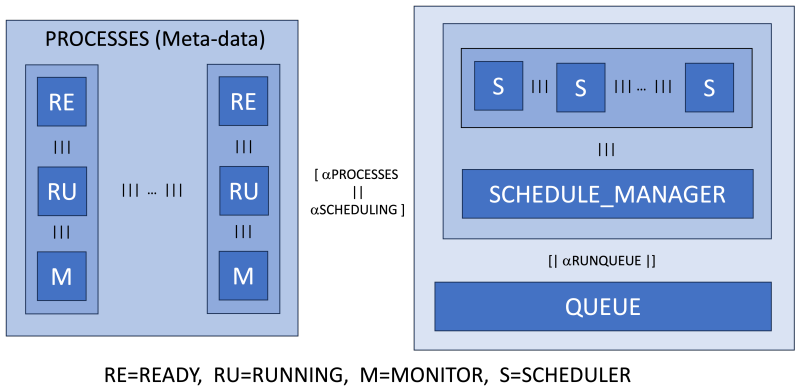

Fig. 2. Scheduler Architecture Diagram (*N_SCHEDULER_SYSTEM*).[3]

4.1.1 Queue. The scheduler requires a queue of ready to run processes. This can be modelled in CSP as follows:

$QUEUE(enqueue, dequeue, q, CAP) =$
  $\text{length}(q) < CAP \;\&\; enqueue?v \rightarrow QUEUE(enqueue, dequeue, q^\frown\langle v\rangle, CAP)$
  $\square$
  $\text{length}(q) > 0 \;\&\; dequeue!\text{head}(q) \rightarrow QUEUE(enqueue, dequeue, \text{tail}(q), CAP)$

The queue supports enqueuing and dequeuing. *CAP* is the maximum capacity of the queue, required for verification. Note the use of *guarded conditions* on each of the choice branches. If the queue is not full it will offer an *enqueue* event, and if it is not empty it will offer a *dequeue* event.

4.1.2 Scheduler(s) and Schedule Manager. In Figure 2, schedulers are labelled *S*. A scheduler is responsible for running a process by dequeuing a *pid* (Process Identifier) from the run queue, sending a *run* event to the relevant process, and waiting for a *yield* event to occur before recursing. Schedulers interleave with a new schedule manager process which accepts *schedule* events—a request to add a *pid* to the run queue—then enqueuing the *pid* onto the queue before recursing.[4] These two processes can be modelled in CSP as follows:

$SCHEDULER =$
  $rqdequeue?p \rightarrow$ – dequeue a process
  $run.p \rightarrow$ – run it
  $yield.p \rightarrow$ – wait for yield
  $SCHEDULER$ – recurse

$SCHEDULE\_MANAGER =$
  $schedule?pid \rightarrow$ – wait for a schedule event
  $rqenqueue!pid \rightarrow$ – place the process in the run queue
  $SCHEDULE\_MANAGER$ – recurse

where *rqenqueue* and *rqdequeue* are channels connected to the queue for enqueuing and dequeueing processes.

4.1.3 Process. In our previous work, a process was not defined as a separate entity. The only information about a process that a scheduler needed was its current ready state (represented by a *ready* flag). Our new scheduling algorithms require two additional pieces of meta-data:

[4]Note, only processes that are ready are ever scheduled.

- A *running* flag to determine if a process is currently being executed by a scheduler.
- A monitor to control access to the process data (*ready* flag and *running* flag) to avoid race conditions during the scheduling and descheduling stages.

PROCESS is therefore an interleaving of its two variables (*RE* for *ready* and *RU* for *running*) and the monitor (*M*) grouped together in the *PROCESSES* process in Figure 2. The CSP for the meta-data of a single process is as follows:

$$
\begin{array}{l}
PROCESS(p) = \\
\quad VARIABLE(ready.p, \text{true}) \qquad\qquad \text{– } ready = \textbf{true} \\
\quad ||| \\
\quad VARIABLE(running.p, \text{false}) \qquad\qquad \text{– } running = \textbf{false} \\
\quad ||| \\
\quad MONITOR(claim\_process.p, release\_process.p)
\end{array}
$$

A collection of *PROCESS* processes are defined as follows:

$$PROCESSES = |||\ p : Processes \bullet PROCESS(p)$$

where *Processes* is the set of *pid*s. *PROCESSES* is now all the meta-data the scheduler requires for all the processes in the system (but not the actual application processes).

4.1.4 Monitor. A monitor is used to control access to shared data—in this instance the channel and process objects. Our previous work only required a monitor for channels. As we now require a monitor for each process, we have made our monitor model more generic. The CSP for the *MONITOR* is as follows:

$$
\begin{array}{l}
MONITOR(claim, release) = \\
\quad claim?pid \rightarrow \qquad\qquad \text{– wait for a claim} \\
\quad release.pid \rightarrow \qquad\qquad \text{– wait for a release on the same pid} \\
\quad MONITOR(claim, release) \qquad\qquad \text{– recurse}
\end{array}
$$

4.1.5 Scheduler System. With everything in place, we can define an *N_SCHEDULER_SYSTEM* process representing our new scheduling system to run in parallel with whatever processes we wish to run. The scheduler side of this process is simply the queue (*QUEUE*) run in parallel with *N* schedulers (*SCHEDULER*) and a single *SCHEDULE_MANAGER*. The schedulers and the schedule manager are interleaved as they do not interact:

$$
\begin{array}{l}
N\_SCHEDULER\_SYSTEM(N) = \\
\quad (\\
\quad\quad (\\
\quad\quad\quad QUEUE(rqenqueue, rqdequeue, <>, \text{card}(Processes)) \\
\quad\quad\quad [|\ \alpha RUNQUEUE\ |] \\
\quad\quad\quad ((|||\ n : \{1..N\} \bullet SCHEDULER)\ |||\ SCHEDULE\_MANAGER) \\
\quad\quad) \\
\quad\quad [\alpha SCHEDULING\ ||\ \alpha PROCESSES] \\
\quad\quad PROCESSES \\
\quad) \setminus \alpha RUNQUEUE
\end{array}
$$

The various alphabets on which the processes synchronise can be found in the CSP code available online.

4.2 Scheduler Interaction

To work with the new scheduler, we require new algorithms for scheduling, descheduling, and yielding. These algorithms are used by executing processes and are not directly part of the scheduler architecture. Rather, when an executing process requires to schedule another process, deschedule itself, or yield to the scheduler, it uses these helper definitions to do so.

4.2.1 Schedule. *SCHEDULE* is a procedure that manages the interaction when an attempt is made to add a process to the run queue. In our previous work, to schedule a process it was sufficient to set its *ready* field to true. Our new model requires a process to determine if it must set a process ready and if so, whether to add it to the run queue. The *SCHEDULE* procedure will handle the scheduling of a process. There are three possible states a process can be in when it is scheduled:

(1) The process is ready, and therefore already exists in the run queue; therefore, do nothing.
(2) The process is not ready, but is still running (i.e., it is scheduled); therefore, set the process ready but do nothing else.
(3) The process is not ready, and is not running; therefore, set the process ready, and schedule it via the schedule manager.

The CSP code for the *SCHEDULE* procedure is:

```
SCHEDULE(me, pid) =
  claim_process.pid.me →                    – lock the process object
  ready.pid.load?r →                        – load the ready field
  if(r) then                                – if (the process is ready) then
    release_process.pid.me → SKIP           –     release the lock
  else                                      – else
    ready.pid.store!true →                  –     set the process ready
    running.pid.load?r2 →                   –     get the running status
    if(r2) then                             –     if (the process is running) then
      release_process.pid.me → SKIP         –         release the lock
    else                                    –     else
      schedule.pid →                        –         schedule the process
      release_process.pid.me → SKIP         –         release the lock
```

where *me* is the process identifier of the process calling *SCHEDULE* and *pid* the process identifier of the process being scheduled. *claim_process.pid.me* locks the process object (indicated by *pid*) to ensure only one schedule or deschedule procedure is executing at a time. We then check the *ready* flag, and if it is true, we release the lock and SKIP (State 1). If the *ready* flag is false, we set it to true (State 2 and State 3), and check the *running* flag. If the *running* flag is true, we release the process and SKIP (State 2). If not running, we add the process to the run queue via the *schedule* event (to the schedule manager), release the process and SKIP (State 3).

4.2.2 Deschedule. The *DESCHEDULE* procedure executes prior to a process yielding control back to the scheduler process. It must therefore set the *running* flag to false. There are two possibilities at this point:

(1) The process is ready—either because it is courtesy yielding or another process has set it ready prior to descheduling.
(2) The process is not ready.

The CSP code for *DESCHEDULE* is as follows:

```
DESCHEDULE(pid) =
  claim_process.pid.pid →                   – lock the process
  running.pid.store.false →                 – set the running field to false
  ready.pid.load?r →                        – get the ready field
  if(r) then                                – if (the process is ready) then
    schedule.pid →                          –     schedule the process
    release_process.pid.pid → SKIP          –     release the lock
  else                                      – else
    release_process.pid.pid → SKIP          –     release the lock
```

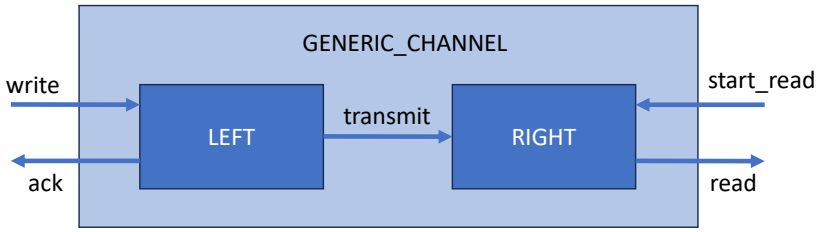

Fig. 3. The *GENERIC_CHANNEL* specification.

Note, only a process can deschedule itself, therefore *pid* is now the process identifier of both the caller and the process being descheduled. After claiming the process and setting *running* to false, we check the *ready* flag. If *ready* is true, we schedule the process again, release the process lock, and SKIP. Otherwise, we only release the process lock and skip.

4.2.3 Yield. Yielding occurs when a process releases the scheduler. That is, when a process is ready to give back control to the scheduler that was running it. Before a process yields, it must call *DESCHEDULE* and, after it is resumed, it must set itself 'as running' (set *running* to true; this is *running.pid.set*!true in the code below). In between the *DESCHEDULE* call and the setting itself ready again, a process engages in the actual *yield* event, and then waits for another scheduler to engage it in a *run* event. The CSP code for *YIELD* is as follows:

```
YIELD(pid) =
  DESCHEDULE(pid);                          – deschedule the process
  (
    yield.pid →                             – yield
    run.pid →                               – run the process again
    running.pid.store!true → SKIP           – set the running field to true
  )
```

5 Specifications

To verify our new scheduler implementation, we return to our previous specifications of channel and choice. Our goal is to demonstrate that the ProcessJ channel and choice mechanisms behave as specified when executed in coordination with the scheduler system. Two specifications are required—a *generic channel* and a *generic alternation.*

5.1 A Generic Channel Specification

For channels, we use the specification presented in [WM00], because interleaving read and write processes with a simple transmit channel does not provide a specification behaviour to verify against. Welch and Martin's specification has also been proven to be equivalent (in the *failures-divergence model*) to a CSP channel (See page 289 of [WM00]). Figure 3 shows the *GENERIC_CHANNEL* specification that we are about to build. First, we define channels to represent a channel object interface:

```
channel read, write : Channels.Processes.Values
channel start_read, ack : Channels.Processes
transmit : Channels.Values
```

The *LEFT* (writing) process accepts a message on the *write* channel, transmits the message to the reader, and performs an *ack* event. This is the equivalent of calling a *write* method on the

channel, with *ack* similar to a *return*. The *RIGHT* (reading) process engages in an event on the *start_read* channel before receiving a message via *transmit*. The transmitted message is passed to the environment on the *read* channel. This is equivalent to calling a *read* method on the channel, with *read* returning the read value to the caller.

$$
\begin{array}{lll}
LEFT(pid, chan) = & write.chan.pid?mess \rightarrow & \text{– take a message} \\
& transmit.chan!mess \rightarrow & \text{– send it} \\
& ack.chan.pid \rightarrow & \text{– acknowledge the send} \\
& LEFT(pid, chan) & \text{– recurse} \\
 & & \\
RIGHT(pid, chan) = & start\_read.chan.pid \rightarrow & \text{– Start the read} \\
& transmit.chan?mess \rightarrow & \text{– received the message} \\
& read.chan.pid!mess \rightarrow & \text{– deliver the message} \\
& RIGHT(pid, chan) & \text{– recurse}
\end{array}
$$

We define a *GENERIC_CHANNEL* by executing *LEFT* and *RIGHT* in parallel:

$$
\begin{array}{l}
GENERIC\_CHANNEL(pid, pid', chan) = \\
\quad (LEFT(pid, chan)[\alpha LEFT(pid, chan) \;||\; \alpha RIGHT(pid', chan)]RIGHT(pid', chan)) \\
\quad \setminus \{transmit.chan.v \mid v \leftarrow Values\}
\end{array}
$$

Note that *write* and *ack* events represent the call and return from a write operation (a channel write); similarly, the *start_read* and *read* events represent the call and return from a reading operation (a channel read). Between these pairs the processes synchronise on *transmit*, representing that the communication actually happened.

We *hide* the *transmit* events as they are internal to the process and not part of the external specification behaviour. The *GENERIC_CHANNEL* process has been verified to be *divergence free*.

5.1.1 Algebraic Proof. The generic channel is identical to Welch and Martin's [WM00]. They also provide an algebraic proof to demonstrate their specification is equivalent to a regular CSP channel. As we use the same specification, we are likewise attempting to prove for ProcessJ. Our previous paper [PC23] describes this equivalence in more detail.

5.2 A Generic Alternation Specification

A simple example of **alternation** or alt in ProcessJ where we reuse the *writer* procedure from the introduction could look like this:

```
void selector(chan<int>.read c1, chan<int>.read c2) {
  int v;
  alt {
    v = c1.read() : { System.out.println(v); }
    v = c2.read() : { System.out.println(v); }
  }
}

void main(string args[ ]) {
  chan<int> c1, c2;
  par {
    writer(c1.write, 42);
    writer(c2.write, 43);
    selector(c1.read, c2.read);
  }
}
```

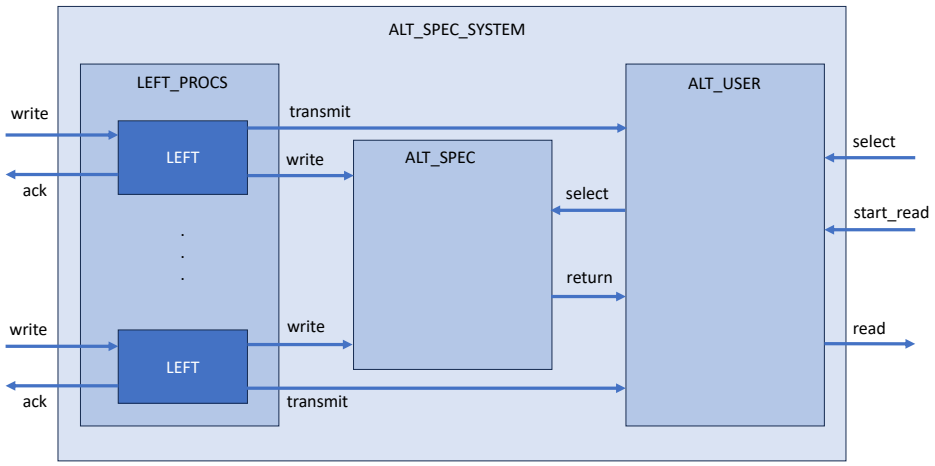

Fig. 4. The *ALT_SPEC_SYSTEM* specification.

An alternation (or *alt* for short) choses one of its *ready* branches and executes it along with the statement that follows. A channel-read (as used here) is ready if there is a committed writer on the channel. The output of running this code is either 42 or 43. In addition, the code will deadlock as only one of the writers will get to go. Only alternation of channel reading on non-shared channels is supported at present in ProcessJ (apart from timer timeouts and skip guard). For this analysis, each branch is of the form *v = c.read(): statement*. Though in ProcessJ, *skip: statement* (skip guard) and *t.timeout(v): statement* (timeout guard) are also allowed, but not considered in this paper.

Figure 4 illustrates the *ALT_SPEC_SYSTEM* specification we are about to construct. The specification uses the same channels as *GENERIC_CHANNEL* (e.g., *write*, *ack*, etc.) and also introduces a *select* event (used to signify the start of a choice being made), and an internal *return* channel to provide the channel identifier picked during the choice process.

To verify correctness of the ProcessJ *alt* (choice), we again turn to Welch and Martin [WM00]. However, contrary to the specification we used in [PC23], this time we have expanded the specification slightly. In particular, we have generalised it such that it works for any number of writers. There are three parts to this specification. The first part we need is an external choice that picks a writer. Generally, for n channels and n writing processes, we need to generate the following external choice:

$write.C1.W1 \rightarrow ALT\_SPEC(\ldots)$
$\Box$
$write.C2.W2 \rightarrow ALT\_SPEC(\ldots)$
$\Box$
$\ldots$
$\Box$
$write.Cn.Wn \rightarrow ALT\_SPEC(\ldots)$

We need generate this external choice 'on the fly', as n is not fixed. We can make this simple using a *zip* style function (e.g., from Haskell) which we can apply to such a builder function:

$zip(<>, <>) = <>$
$zip(< x >^\frown xs, < y >^\frown ys) = < (x, y) >^\frown zip(xs, ys)$

We also need two channels: one to kick off the *alt* and one to obtain the returned channel chosen. They are as follows:

channel *select* : *Processes*
channel *return* : *Channels*

We can generate the first part, which is an external choice taking messages (i.e., writes) as follows:

$$\Box\ (c, p) : pairs \bullet write.c.p?mess \rightarrow ALT\_SPEC(selector, pairs, union(writing, \{c\}), waiting)$$

The second requirement (containing the second and third part) for an *alt* is to service the selecting process. There are two choices: either the select has not started and the *alt* must offer the select event to begin the process; or the select has started and the *alt* must offer the currently ready channels. These ready channels are kept in the parameter *writing*. In an external choice of writing processes, we offer up a *return* event:

$$\Box\ c : writing \bullet return.c \rightarrow ALT\_SPEC(selector, pairs, \mathsf{diff}(writing, \{c\}), \mathsf{false})$$

where any writing process c is added to the *writing* set. *waiting* is a flag that will capture whether or not the selection process has started. Putting these two choices together with an event on the *select* channel indicating that the *alt* has begun, we get the following code for the *ALT_SPEC* process:

$$\begin{array}{l}
ALT_SPEC(selector, pairs, writing, waiting) = \\
\quad (\Box\ (c, p) : pairs \bullet write.c.p?mess \rightarrow ALT_SPEC(selector, pairs, \mathsf{union}(writing, \{c\}), waiting)) \\
\quad \Box \\
\quad \mathsf{if}\ waiting\ \mathsf{then} \\
\quad\quad (\Box\ c : writing \bullet return.c \rightarrow ALT_SPEC(selector, pairs, \mathsf{diff}(writing, \{c\}), \mathsf{false})) \\
\quad \mathsf{else} \\
\quad\quad select.selector \rightarrow ALT_SPEC(selector, pairs, writing, \mathsf{true})
\end{array}$$

Note, an event on *select* sets the *waiting* flag to true, indicating that the *alt* has started. Additionally, we need a process to use the *alt* and perform the actual read; we call this process *ALT_USER*, and it is defined as follows:

$$\begin{array}{ll}
ALT_USER(pid) = & \\
\quad select.pid \rightarrow & \text{– kick off the alt} \\
\quad return?idx \rightarrow & \text{– get a returned index of a ready channel back} \\
\quad start_read.idx.pid \rightarrow & \text{– Initiate the read} \\
\quad transmit.idx?mess \rightarrow & \text{– transmit the message internally} \\
\quad read.idx.pid!mess \rightarrow & \text{– do the actual read from the chosen channel} \\
\quad ALT_USER(pid) & \text{– recurse}
\end{array}$$

Before we can wire up the complete specification, we need a few helper functions. They are defined as follows:

$$PRIVATE\_ALT\_SPEC\_SYSTEM(pid) = \{transmit.c.v, return.c \mid c \leftarrow Channels, v \leftarrow Values\}$$

$$\begin{array}{l}
LEFT_PROCS(pairs) = \\
\quad ||| (c, p) : pairs \bullet LEFT(p, c)
\end{array}$$

where *LEFT_PROCS* creates n interleaved *LEFT* processes (the writers). *PRIVATE_ALT_SPEC_SYSTEM* contains the events that the alt specification will not engage with externally. We can now finally wire up the *ALT_SPEC_SYSTEM* specification as follows:

$$\begin{array}{l}
ALT_SPEC_SYSTEM(selector, chans, writing_procs) = \\
\quad (\\
\quad\quad (\\
\quad\quad\quad LEFT_PROCS(\mathsf{set}(zip(chans, writing_procs))) \\
\quad\quad\quad [\alpha LEFT_PROCS(writing_procs, chans) \ ||\ \alpha ALT_SPEC(selector)] \\
\quad\quad\quad ALT_SPEC(selector, \mathsf{set}(zip(chans, writing_procs))), \{\}, \mathsf{false}) \\
\quad\quad) \\
\quad\quad [\mathsf{union}(\alpha LEFT_PROCS(writing_procs, chans), \alpha ALT_SPEC(selector))\ ||\ \alpha ALT_USER(selector)]
\end{array}$$

ALT_USER(*selector*)
) \ union(*PRIVATE_ALT_SPEC_SYSTEM*(*selector*), {| *transmit* |})

where *chans* represents the channels, *writing_procs* represent the writers and *selector* is the selecting process. Initially, all flags and *waiting* are false, signifying that no writers are ready and the reader (the selector) is not waiting. When a writer becomes ready, the appropriate *ready* flag is set to true. The remaining events not hidden are equivalent to the events that appear outside the box in Figure 4.

The writing processes are *LEFT* processes from Section 5.1 build using *LEFT_PROCS*. *ALT_USER*, replaces the *RIGHT* process in regular channel communication. *transmit* and *return* events are hidden by defining *PRIVATE_ALT_SPEC_SYSTEM*. FDR verifies that the specification of the generic *alt* is deadlock-free both in the *failures* and in the *failures-divergence* model. Similar to the channel specification, we have also verified that the *ALT_SPEC_SYSTEM* is divergence free. The alt is also nondeterministic which occurs if multiple writing events are ready.

6 Channel Verification

In the previous two sections we presented:

- the CSP specifications for a the scheduler architecture: queue; the scheduler and the scheduler manager; a monitor; a process's meta-data (the part of a process required for the scheduler) and the scheduler system.
- the code a process uses to interact with the scheduling system: schedule, deschedule, and yield.
- the specifications for correct behaviour of a channel and alternation.

In this section and the following one, we provide the CSP model of the proposed ProcessJ translated code to meet the specification. In this section, the CSP for a channel communication system (a writer and a reader along with a channel) is presented. These processes running concurrently with the scheduler system allow verification of the channel implementation against the defined specification. In this section, we also highlight changes from our previous work [PC23] through underlined CSP statements.

An example of a channel communication in ProcessJ can be found in the introduction section of this paper. Recall that channel communication is one-to-one and synchronous.

A channel consists of a single monitor and three variables (fields): the reference (pid) of the writer process, the reference of the reader process, and the reference to the data.

First we define some constants and channels required for communicating with the *VARIABLE* processes and the *MONITOR* used in the *CHANNEL* process:

```
datatype Nullable_Processes = NULL | P1 | P2          – set of ProcessJ processes.
subtype Processes = P1 | P2
datatype Channels = C1                                – set of ProcessJ channels.
datatype Values = A | B                               – data values.

channel data : Channels.Operations.Values             – channels
channel writer, reader : Channels.Operations.Nullable_Processes
channel channel_claim, channel_release : Channels.Processes
```

The *reader.chan* and *writer.chan* represent references to the reader and the writer of the channel. The *data.chan* is the field representing the value being communicated across the channel. The monitor process uses the *channel_claim.chan* and *channel_release.chan* channels for claiming and releasing a lock on the channel.

A channel is the three *VARIABLE* processes and a single *MONITOR* process interleaved (along with the code for *READ* and *WRITE*.):

```
CHANNEL(chan) =
  VARIABLE(writer.chan, NULL)                    – writer = null
  |||
  VARIABLE(reader.chan, NULL)                    – writer = null
  |||
  VARIABLE(data.chan, A)                         – data = A
  |||
  MONITOR(channel_claim.chan, channel_release.chan)
```

To verify our channel implementation working with our scheduler, we need to build a system consisting of a single channel (as defined above) and a reader and a writer. We will consider the reader and the writer in the following section.

6.1 Channel Communication

What does the CSP that corresponds to a ProcessJ channel-write statement (e.g, *out*.**write**(42)) look like? Here, we incorporate it into a process called *RESTICTED_PROCESS_WRITER*[5].

6.1.1 A Writing Process. A writing process behaves as follows:

- It must be scheduled (i.e., it needs to be ready to run and placed in the run queue).
- It then must be run by the scheduler.
- The write commences (and loops):
  - A message (via *write*) is taken from the environment.
  - The channel is claimed to ensure exclusive access by the writer.
  - The data field is written to with the message.
  - The writer field is written to with the process identifier (pid) of the writing process.
  - The writing process sets itself not ready.
  - If the reading process is present (the *reader* field is not null), the reading process is scheduled.
  - The channel is released from exclusive access.
  - The writing process yields.
  - Once the writer is rescheduled, an *ack* event is raised to the environment.

Here is the associated CSP code:

```
RESTRICTED_PROCESS_WRITER(pid, chan) =
  schedule!pid →                                 – schedule the process
  run.pid →                                      – wait to be run
  running.pid.set!true →                         – this.running = true
  RESTRICTED_PROCESS_WRITER'(pid, chan)

RESTRICTED_PROCESS_WRITER'(pid, chan) =
  write.chan.pid?message →                       – take message from the environment
  channel_claim.chan!pid →                       – claim the channel
  WRITE(pid, chan, message);                     – perform the write
  channel_release.chan.pid →                     – release the channel
  YIELD(pid);                                    – yield
  ack.chan.pid →                                 – acknowledge to the environment
  RESTRICTED_PROCESS_WRITER'(pid, chan)          – recurse
```

[5]Restricted because it is controlled by the scheduler as opposed to an unrestricted process that is not controlled by a scheduler.

```
WRITE(pid, chan, item) =
  data.chan.store!item →                  – set the data field of the channel
  writer.chan.store!pid →                 – register the writer
  ready.pid.store!false →                 – set the process not ready to run
  reader.chan.load?v →                    – get the reader field
  if (v != NULL) then                     – if (reader ≠ null) then
    SCHEDULE(pid, v)                      –     schedule the reader
  else                                    – else
    SKIP                                  –     do nothing
```

The differences between the new version above and the version found in [PC23] are as follows (the lines are underlined in the code above):

- *schedule!pid*: the first time a process runs, it must schedule itself. When a process is created, its *ready* field is set to true and its *running* field to false. In the previous version, a fixed schedule was provided to the system.
- *running.pid.store!true*: when a process runs (either the first time or after resuming from a yield) it must set its *running* field to true. Note that this happens also in the *YIELD* procedure. The *running* flag is a new addition.
- *SCHEDULE(pid, v)*: rather than simply setting the reader process's *ready* field to true, we use the new *SCHEDULE* procedure to schedule the reader again. This is due to the new scheduling algorithm in use.

The yielding of the writer is necessary to ensure that the communication is indeed synchronous. It is worth noting that a one-to-one channel is always ready for a writer to write on when it gets to the write statement. If there were data still on the channel, the reader would not yet have read it, and the writer would have been not-ready to run, but if the writer is ready to run, the data must have been read because the writer could only have been scheduled by the reader as the channel is one-to-one and synchronous.

6.1.2 A Reading Process. The reading process is more complicated than the writer as there is no guarantee that a reader can continue directly to reading a message from the channel, as the writer may not have written any data yet. It behaves as follows:

- The reading process is scheduled.
- The reading process is run.
- The running flag is set to true.
- The read commences and loops:
  - A *start_read* event is accepted from the environment indicating the read has begun.
  - The channel is locked for exclusive access.
  - If the writer is not present (the *writer field is null*):
    - The *reader* field is updated with the reading process ID.
    - The reading process is set not ready.
    - The channel is unlocked for exclusive access.
    - The reading process yields.
  - Else, if the writer is present (the *writer* field is not null):
    - The channel is unlocked for exclusive access.
  - The channel is locked for exclusive access (necessary as the reading process might have yielded prior to this point).

- The writing process is scheduled[6].
- The reader and writer fields are set to null.
- The data is retrieved from the channel.
- The channel is unlocked for exclusive access.
- The reading process yields (a courtesy yield).
- The data is returned to the caller (via the *read* event).

The actual CSP looks like this:

```
RESTRICTED_PROCESS_READER(pid, chan) =
    schedule.pid →                                  – schedule the process
    run.pid →                                       – wait to be run
    running.pid.set!true →                          – this.running = true
    RESTRICTED_PROCESS_READER'(pid, chan)

RESTRICTED_PROCESS_READER'(pid, chan) =
    start_read.chan.pid →                           – external event to start the read
    channel_claim.chan!pid →                        – claim the channel
    writer.chan.load?p →                            – get the writer field
    (
        if (p == NULL) then                         – if (writer == null) then
            reader.chan.store.pid →                 –    reader = this
            ready.pid.store!false →                 –    this.ready = false
            channel_release.chan.pid →              –    release the monitor
            YIELD(pid)                              –    yield
        else                                        – else
            channel_release.chan.pid →              –    release the channel
            SKIP
    );
    channel_claim.chan!pid →                        – claim the channel
    READ(pid, chan);                                – see READ below
    data.chan.load?message →                        – message = data
    channel_release.chan.pid →                      – release the monitor
    (
        YIELD(pid);                                 – yield
        read.chan.pid!message →                     – deliver the message to the environment
        RESTRICTED_PROCESS_READER'(pid, chan)       – recurse to start over
    )

READ(pid, chan) =
    writer.chan.load?w →                            – get the writer field.
    if (w == NULL) then
        DIV                                         – never happens.
    else
        SCHEDULE(pid, w);                           – schedule the writer
        (
            writer.chan.store!NULL →                – writer = null
            reader.chan.store!NULL →                – reader = null
            SKIP
        )
```

[6]Note that this writer cannot write a new value to the channel because the channel is still locked by the reader.

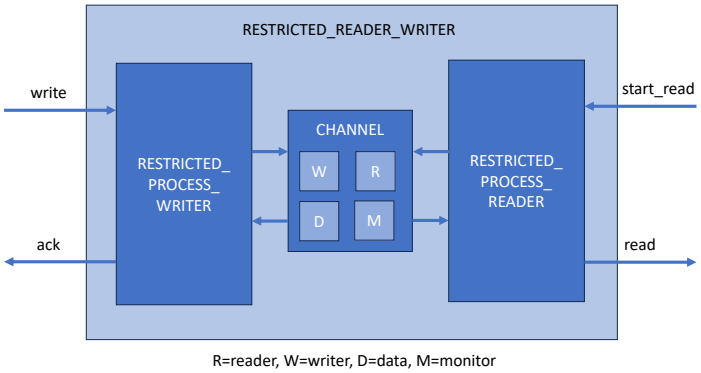

Fig. 5. A writer and a reader along with a channel.

We now have both the writing and the reading side of our channel communication system (the left and right parts of Figure 5). We can now wire them up with with a channel and run them with a scheduler.

6.1.3 Scheduled Reader/Writer System. We can now create a process that runs the reader and the writer interleaved and both in parallel with a channel; see Figure 5. That CSP looks as follows:

$$
\begin{array}{l}
RESTRICTED\_READER\_WRITER(writer\_proc, reader\_proc, chan) = \\
\quad (\\
\quad\quad (\\
\quad\quad\quad RESTRICTED\_PROCESS\_WRITER(writer\_proc, chan) \qquad \text{– a writing process} \\
\quad\quad\quad ||| \\
\quad\quad\quad RESTRICTED\_PROCESS\_READER(reader\_proc, chan) \qquad \text{– a reading process} \\
\quad\quad) \\
\quad\quad [|\ \alpha CHANNELS\ |] \\
\quad\quad CHANNELS \qquad \text{– a channel} \\
\quad) \setminus \alpha CHANNELS
\end{array}
$$

where *CHANNELS* is defined as follows:

$$CHANNELS = |||\ c : Channels \bullet CHANNEL(c)$$

which, because *Channels* is just $\{C1\}$, evaluates to a single channel, namely, $CHANNEL(C1)$.

All we now have to do is run this process in parallel with the *N_SCHEDULER_SYSTEM* from Section 4 to have a complete CSP specification of the *implementation* of a channel communication in ProcessJ. Here, n is the number of schedulers that we want the system to run with.

$$
\begin{array}{l}
RESTRICTED\_PJ\_CHAN\_SYSTEM(reader\_proc, writer\_proc, chan, n) = \\
\quad (\\
\quad\quad RESTRICTED\_READER\_WRITER(reader\_proc, writer\_proc, chan) \\
\quad\quad [|\ \alpha N\_SCHEDULER\_SYSTEM\ |] \\
\quad\quad N\_SCHEDULER\_SYSTEM(n) \\
\quad) \setminus \alpha N\_SCHEDULER\_SYSTEM
\end{array}
$$

We can now perform the verification using FDR by asserting deadlock freedom, livelock freedom, determinism, and the following refinement checks:

$$GENERIC\_CHANNEL(P1, P2, C1) \sqsubseteq_M RESTRICTED\_PJ\_CHAN\_SYSTEM(P1, P2, C1, n)$$
$$RESTRICTED\_PJ\_CHAN\_SYSTEM(P1, P2, C1, n) \sqsubseteq_M GENERIC\_CHANNEL(P1, P2, C1)$$

Table 1. Results of FDR Verification for Channels Under New Scheduling System

	Restricted (Number of Schedulers)								Unrestricted
	1		2		3		4		
	Old	New	Old	New	Old	New	Old	New	
Deadlock free	FD	FD	F	FD	F	FD	F	FD	FD
Livelock free	FD	FD	✗	FD	✗	FD	✗	FD	FD
Spec ⊑ Impl	T	T	F	FD	F	FD	F	FD	FD
Impl ⊑ Spec	✗	✗	T	FD	T	FD	T	FD	FD

Spec = GENERIC_CHANNEL, Impl = RESTRICTED_PJ_CHAN_SYSTEM.

where *M* is either *T* (trace refinement), *F* (failures refinement), or *FD* (failures-divergence refinement). We use n=1, 2, 3, 4, and $n \geq 2$ is enough schedulers for each process to get one.

6.2 Verification Results for ProcessJ Channel Implementation Under New Scheduling System

In Table 1 the right-most column labelled **Unrestricted**[7] is the verification results we obtained in [PC23] for channel communication *without* a scheduler. That is, the results we obtained by applying the *standard* approach to verification where no execution environment or hardware was taking into account. This column represents our verification goal. If we can achieve the same results with the implementations in this paper, we will have demonstrated that the channel behaves as specified within our execution environment. As Table 1 shows, there is limited success based on the number of running schedulers available. The implementations that include a scheduler are referred to as **Restricted**. In Table 1, the results from [PC23] are listed in the **Old** columns, and the new results from this paper are listed in the **New** columns.

With the newly developed scheduler system, we can now repeat all the CSP assertions developed in [PC23] to verify if we indeed have managed to achieve success in the *failures-divergence* model, the strictest of the three FDR models[8]. The entries which differ from the old to the new results have been highlighted with a grey box.

As we can see, most of the new entries are indeed FD, which means that the checks succeeded in the *failures-divergence* semantic model and yielded the same as the unrestricted verification (i.e., the 'standard' verification approach without taking the scheduler into account). However, a number of entries are not.

- 1 Scheduler:
  - *Spec* ⊑ *Impl*: It is clear that the implementation's traces are indeed a (true) subset of the traces allowed by the specification. With a single scheduler the order in which processes appear in the run queue will make some of the traces allowed by the specification impossible. The specification requires both the *start_read* and *write* events are available. A single scheduler means only one of these two events will be available, non-deterministically based on the initial queue. The specification also requires *read* and *ack* are available at the same time. However, with a single scheduler, the order of events will become sequential — ⟨*ack*, *write*, *read*⟩. Note that the next *write* will occur before *read* due to the yielding and scheduling within the system. Channel behaviour

[7]Unrestricted because there was no scheduler to restrict the execution.

[8]We do realise that since both the specification and the implementation are divergence-free, verifying in the *stable failures* model is technically sufficient.

then becomes a known fixed sequence of events. Failures refinement is not possible as the refusal sets will differ: there is only one scheduler, so the process that can run is limited to the next ready process in the run queue.

– *Impl* ⊑ *Spec*: With a single scheduler and an ordering imposed by the order in which the processes are placed in the run queue, the implementation will never be able to mimic the specification completely. If the reader is in the queue before the writer, the reader will be run first. Similarly, if the writer is before the reader in the queue, it will be run first. The specification does not make that distinction. It should be obvious that the implementation's set of traces is indeed a true subset of that of the specification—something FDR verifies as well.[9]

With enough schedulers, which, in this case is just two, all results will be the same as the unrestricted version. As more processes are added to the system, further schedulers would be required to meet the specification. For example, with a three channel system featuring six processes, if five or fewer schedulers are available then likewise the implementation will only refine within the traces model of the specification, not the failures model.

7 Choice Verification

With channel communication handled, we now turn to choice (also known as *alternation*, or simply *alt*).

For choice we have two writers ($W1$ and $W2$) and a single reader (R) that performs an alternation between the two writers. Thus, we also need two channels ($C1$ and $C2$). Therefore, we have the following slightly changed initial definitions:

datatype *Nullable_Processes* = *NULL* | *R* | *W*1 | *W*2 – set of ProcessJ processes
subtype *Processes* = *R* | *W*1 | *W*2
subtype *Writers* = *W*1 | *W*2

datatype *Channels* = *NONE* | *C*1 | *C*2 – set of channels (*NONE* is '*NULL*')
subtype *Active_Channels* = *C*1 | *C*2

datatype *Values* = *A* | *B* – data values

This time, we get two channels as follows:

CHANNELS = ||| *c* : *Active_Channels* • *CHANNEL*(*c*)

The selector/reader needs two additional *VARIABLE* processes for variables called *i* and *selected*. We define them as follows:

ALT_VARIABLES = *VARIABLE*(*i*, *NONE*) ||| *VARIABLE*(*selected*, *NONE*)

where *NONE* serves as the *null* value for channels. We have added a *NONE* value as during the enable/disable process of the select, we might have no channel selected.

We now can add channels for communicating with the variables:

channel *i, selected* : *Operations.Channels*

7.1 The Writing Processes

The implementation of the writing end of the two channels part-taking in the choice is exactly the same as we saw in Section 6.1.1. The writing processes are not aware that the reading ends of the channels on which they communicate are used in a choice operation in the reading process.

[9] In general, if there are not enough schedulers for each process to have its own, the implementation will *never* more than trace-refine the specification as some traces in the implementation are simply impossible. Naturally as long as the system does not deadlock (which it never does) we are fine.

7.2 The Reading Choice Process

The alt operator chooses between a number of ready channel communications (here just two) by picking one of them based on a rule set. Here this rule set is simply picking the first one that is ready (by the order in which they appear in the program)—this is called a *prioritized choice* or sometimes a *pri alt*. This choice is performed by an enable/disable sequence which first traverses all the channel communications from top to bottom and determines the top-most ready one (enable), then traverses them backwards and updates the top-most ready one if one further up is found (disable). The enable sequence maintains a variable *i* (holding a channel) and the disable sequence produces the channel to be read in a variable called *selected* (stored in the variable *c* in the CSP below). See [PC23] for more details about the working of the *ENABLE* and *DISABLE* procedures; also note that in this paper both procedures are now more general, recursive versions, that work with any number of branches like the rest of the *alt* system. The steps of the reading choice are:

- The selecting process is scheduled.
- The selecting process is run.
- The running flag is set to true.
- The select commences and loops:
  - The ready flag is set to false.
  - The select operation is started.
  - The branch enable sequence is executed.
  - The selecting process yields—it may be ready to run if a ready branch was found, or it might now wait until one of the channels signals it is ready.
  - The branch disable sequence is executed.
  - The selected channel is retrieved.
  - The read operation is started on the selected channel.
  - The selected channel is locked for exclusive access.
  - The writing process is scheduled.
  - The reader and writer fields are set to null.
  - The data is retrieved from the channel.
  - The channel is unlocked for exclusive access.
  - The selecting process yields (a courtesy yield).
  - The data is returned to the caller (via the *read* event).

The equivalent CSP is:

```
RESTRICTED_PROCESS_SELECTOR(pid) =
  schedule!pid →                          – schedule the process
  run.pid →                               – wait to be run
  running.pid.set!true →                  – this.running = true
  RESTRICTED_PROCESS_SELECTOR'(pid)

RESTRICTED_PROCESS_SELECTOR'(pid) =
  ready.pid.store!false →                 – this.ready = false
  select.pid →                            – external event to start the alt
  ENABLE(pid);                            – execute the enable sequence
  YIELD(pid);                             – yield
  DISABLE(pid);                           – execute the disable sequence
  selected.load?c →                       – read the selected variable
  (
    start_read.c.pid →                    – start the read on the selected channel
    channel_claim.c.pid →                 – claim the channel
```

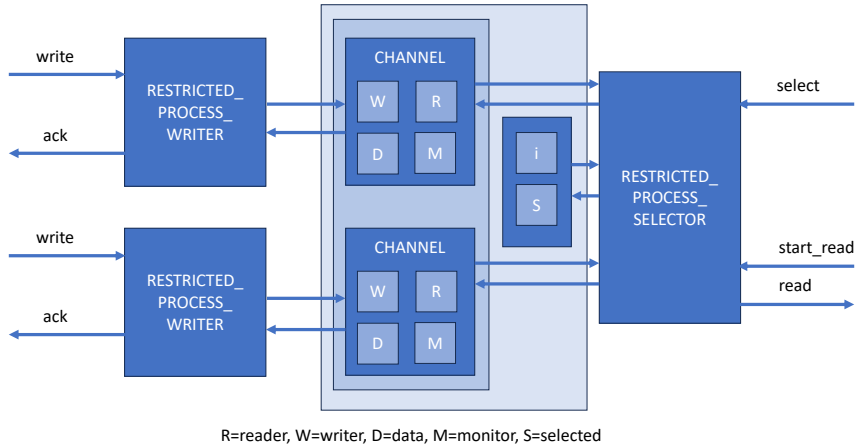

Fig. 6. Two writers and a selector/reader along with two channels and the extra alt variables.

$READ(pid, c)$; – perform the actual read
$data.c.load?message \rightarrow$ – *message* = *data*
$channel\_release.c!pid \rightarrow$ – release the channel
$YIELD(pid)$; – courtesy yield
$read.c.pid!message \rightarrow$ – deliver the message to the environment
$RESTRICTED\_PROCESS\_SELECTOR'(pid)$ – recurse to start over
)

where *READ* is as defined in Section 6.1.2. Note, both *ENABLE* and *DISABLE* may set the reading process's ready field back to *true* if a ready channel is encountered. We now can wire two writers and a single selector together in a system along with the channels and the required variables (*i* and *selected* in the *ALT_VARIABLES* process) and get the following CSP process (which is depicted in Figure 6).

That CSP code looks like this:

$RESTRICTED\_SELECTOR\_WRITERS(writing\_procs, selector, chans) =$
(
$(CHANNELS \;|||\; ALT\_VARIABLES)$
$[|\ \text{union}(\alpha CHANNELS, \alpha ALT\_VARIABLES)\ |]$
$ALT\_NETWORK(writing\_procs, selector, chans)$
) \ union($\alpha CHANNELS, \alpha ALT\_VARIABLES$)

where *ALT_NETWORK* consists of a single *RESTRICTED_PROCESS_SELECTOR* and the appropriate number of *RESTRICTED_PROCESS_WRITER* processes:

$ALT\_NETWORK(writing\_procs, selector, chans) =$
$RESTRICTED\_PROCESS\_SELECTOR(selector)$
$|||$
$PROCESS\_WRITER\_NETWORK(writing\_procs, chans)$

$PROCESS\_WRITER\_NETWORK(<proc>\hat{}\,writing\_procs, <chan>\hat{}\,chans) =$
$RESTRICTED\_PROCESS\_WRITER(proc, chan)$
$|||$
$PROCESS\_WRITER\_NETWORK(writing\_procs, chans)$

Table 2. Results of FDR Verification for Choice Under New Scheduling System

	Restricted (Number of Schedulers)								Unrestricted
	1		2		3		4		
	Old	New	Old	New	Old	New	Old	New	
Deadlock free	FD	FD	F	FD	F	FD	F	FD	FD
Livelock free	FD	FD	✗	FD	✗	FD	✗	FD	FD
Spec ⊑ Impl	T	T	T	T	F	FD	F	FD	FD
Impl ⊑ Spec	✗	✗	✗	✗	T	FD	T	FD	FD

Spec = ALT_SPEC_SYSTEM, Impl = RESTRICTED_PJ_ALT_SYSTEM

where *PROCESS_WRITE_NETWORK* builds an interleaving of the correct number of *RESTRICTED_PROCESS_WRITER* processes. Finally, just like with the channel communication, we run in parallel with a scheduling system and obtain the final process, *RESTRICTED_PJ_ALT_SYSTEM*, which looks like this:

```
RESTRICTED_PJ_ALT_SYSTEM(writing_procs, selector, chans, n) =
  (
    RESTRICTED_SELECTOR_WRITERS(writing_procs, selector, chans)
    [| αN_SCHEDULER_SYSTEM |]
    N_SCHEDULER_SYSTEM(n)
  )\ αN_SCHEDULER_SYSTEM
```

This time, the *RESTRICTED_PJ_ALT_SYSTEM* can be checked for deadlock, livelock, and determinism, as well as verified against the *ALT_SPEC_SYSTEM* specification from Section 5.2:

$$ALT\_SPEC\_SYSTEM(R, <C1, C2>, <W1, W2>) \sqsubseteq_M RESTRICTED\_PJ\_ALT\_SYSTEM(<W1, W2>, R, <C1, C2>, n)$$

$$RESTRICTED\_PJ\_ALT\_SYSTEM(<W1, W2>, R, <C1, C2>, n) \sqsubseteq_M ALT\_SPEC\_SYSTEM(R, <C1, C2>, <W1, W2>)$$

Again, M is one of T, F, or FD. R is the reader/selector process, and $W1$ and $W2$ are the two writing processes, and $C1$ and $C2$ are the channels. We use $n = 1, 2, 3, 4$, and $n \geq 3$ is enough schedulers for each process to get one.

7.3 Verification Results for ProcessJ Choice Implementation Under New Scheduling System

In Table 2 we present the choice verification results using the same mapping between restricted, old and new results. We have again repeated all the same CSP assertions from [PC23] to verify choice in the *failures-divergence* model. As with channel, most of the new entries are FD. In the following section we shall explain the differences in the results.

7.3.1 *Analysis of Choice Verification Results.*

- 1 and 2 Schedulers:
  - Spec ⊑ Impl: The implementation traces are a subset of the specification. However, as with the single-scheduler channel, the system cannot undertake all possible event interleavings, meaning the implementation refusal set is different from the specification refusal set.
  - Impl ⊑ Spec: As the implementation does not contain all possible traces of the specification (due to the different refusal set), the specification traces are not a proper subset

of the implementation. The specification can allow progress of each writer and the selector concurrently, whereas only one process in the single scheduler system can continue at a time.

The limitations here are the same as with channels. With enough schedulers, which, in this case is three, all results will be the same as the unrestricted version. Without enough *SCHEDULER* processes, some traces can no longer happen due to the order in which processes sit in the run queue; if there are not enough schedulers to run them all, then the run queue order imposes certain restrictions on the traces that are possible. This will not happen if there are at least as many schedulers as there are processes.

7.4 Three-Branch Choice

We have also verified a three-branch *alt* using the same CSP as for the two-branch *alt*. The results are like those in Table 2 except for three schedulers the refinement check $Spec \sqsubseteq Impl$ which now only succeeds in the traces model and the refinement $Impl \sqsubseteq Spec$ now fails in all models. This is exactly as the two-branch *alt* behaves with two schedulers. The assertions pass for four schedulers. In other words, we have also shown that for four active processes (three writers and one selector) we require at minimum four scheduling processes to enable meeting the specification in the stable-failures model.

It should be noted that although we have demonstrated a two-branch alt and a three-branch alt verification, the analysis is not general enough for us to prove that the alt would work with any arbitrary number of branches with the right number of supporting scheduling processes. However, we believe the demonstration of two-branch and three-branch alt is enough for us to conclude that this is likely the case.

8 Analysis of Scheduling Behaviours

In this section, we use different analysis and explanation techniques to illustrate the unintended prioritisation (and thus increasing likelihood of traces not occurring) that can occur during channel communication. We first consider trace-limitations that can be caused by insufficient resources, before presenting empirical evidence of the prioritisation occurring within JCSP. Finally, we present an analysis of the interactions occurring during channel communication for both ProcessJ and JCSP to demonstrate the availability to continue at points of interaction.

8.1 The Trace-Limitations Caused by Insufficient Resources

We have already seen how a lack of resources (not enough schedulers) can impact the possible traces of a system. In this section we illustrate this issue with a small general example. In a single-threaded execution environment, consider the following simple CSP:

$A = a \rightarrow \mathsf{SKIP}$
$B = b \rightarrow \mathsf{SKIP}$
$AB = A \;|||\; B$

We can easily determine the set of traces (again ignoring the termination event ✓) of AB, and it is:

$$traces(AB) = \{\langle\rangle, \langle a\rangle, \langle b\rangle, \langle a, b\rangle, \langle b, a\rangle\}$$

After the empty trace $\langle\rangle$, FDR computes the set of events the process is willing to engage in as the set $\{a, b\}$—this set is called the *acceptance set*. Similarly, the *stable refusal set*—the set of events that the process can refuse to engage in—is the empty set $\{\}$. That is, either of A and B can progress (as A and B are interleaved). Now consider an execution context with a single scheduler and a single run queue. This run queue will contain the equivalent processes of A and B, say, P_A and P_B.

With two processes, there are two different ways such a run queue can look (denoted R_1 and R_2), and they are as follows:

$R_1 = [P_A, P_B]$ or $R_2 = [P_B, P_A]$

If the run queue has the form R_1, then AB effectively becomes $A\ ;\ B$ (ignoring the SKIPs). If R_2 is the run queue, then AB is $B\ ;\ A$. Then possible traces when using R_1 as the run queue is[10]:

$$traces((P_A \ ||| \ P_B)_{R_1}) = traces(A\ ;\ B) = \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\}$$

that is, the set of traces that can be produced by running the single scheduler with the particular run queue configuration found in R_1. Similarly, if we use the R_2 run queue configuration, we get:

$$traces((P_A \ ||| \ P_B)_{R_2}) = traces(B\ ;\ A) = \{\langle\rangle, \langle b\rangle, \langle b, a\rangle\}$$

It should be clear that for any run queue schedule R_i, the following equation will therefore hold:

$$traces((P_A \ ||| \ P_B)_{R_i}) \subseteq traces(AB)$$

while at the same time we have the following:

$$traces(AB) \nsubseteq traces((P_A \ ||| \ P_B)_{R_i})$$

If AB is a *specification*, and $(P_A \ ||| \ P_B)_{R_i}$ an *implementation*, then the implementation (when running under a single ProcessJ scheduler) will always *behave* and never produce a trace disallowed by the specification, because the scheduler is deterministic. However, there will be traces of the specification that a particular run queue configuration cannot produce (e.g., $\langle b, a\rangle$ cannot be produced by the R_1 configuration). This will be the reason for the trace refinement failing in this direction when running with just a single scheduler in ProcessJ.

8.2 Empirical Evidence of JCSP Prioritisation

We believe that preemptive implementations of a CSP-inspired runtime will likewise suffer from unintended priorities. Indeed, we have empirical evidence demonstrating that JCSP has such an unintended prioritisation. Consider the following system specification (presented here in CSP):

$PROC\_1 = a!0 \rightarrow b!1 \rightarrow PROC\_1$
$PROC\_2 = a?0 \rightarrow b!2 \rightarrow PROC\_2$
$RECEIVER = b?x \rightarrow b?y \rightarrow RECEIVER$
$SYSTEM = (PROC\_1\ [|\ \{a\}\ |]\ PROC\_2)\ [|\ \{b\}\ |]\ RECEIVER$

We have implemented this system in both JCSP and ProcessJ. The code is not presented here for simplicity as the CSP definition is enough to explain the general behaviour under consideration.

In this system, *PROC_1* and *PROC_2* first communicate via channel a (*PROC_1* writing and *PROC_2* reading). They then both communicate to *RECEIVER* via channel b. *PROC_1* and *PROC_2* do not synchronise on channel b, and so will proceed in some order. For *RECEIVER*, the values of x and y will either be 1 and 2 (*PROC_1* communicates on b first) or 2 and 1 (*PROC_2* communicates on b first).

In a fair system, with no hidden priorities, *RECEIVER* will have equal probability to see either option for the values of x and y. On an unfair system, the probability will favour one choice over another. We are fully aware that CSP does not consider probabilities in this manner, and probabilistic models exist. However, from an implementation point-of-view, it is important to recognise the outcomes of limited resources, which monitoring the likelihood of an event occurring provides empirical evidence of.

[10]We will use the a and b events from the CSP here as well

In JCSP, channel *b* is implemented as an *Any2OneChannel*. This channel has a shared writing end, protected by a monitor. With this implementation of the specified system, we have created a race condition between *PROC*_1 and *PROC*_2 to claim the shared channel end to use channel *b*.

When we implement this system in JCSP and run it on a single processor machine (such as a Raspberry Pi Zero) or otherwise constrain the available processors for the JVM to one, over 99.999% of the time, *RECEIVER* will observe 2 then 1 on channel *b*, demonstrating priority to the reading process, *PROC*_2, during channel communication. Thus, we know that JCSP, when constrained of resources will demonstrate unexpected priority in its channel communication based on the fact that reading leaves the communication first. In ProcessJ, the results we see are writing leaving the communication first, giving priority to *PROC*_1.

The system (execution environment) has prioritised a particular trace. As no execution environment analysis was taken for JCSP, the existing verification of JCSP would not signal an issue that certain behaviours may not be possible under certain resource and scheduling conditions. It can be surmised that the JCSP channel does not fully behave as a CSP channel under constrained conditions, and a CSP channel cannot be naïvely used as a drop-in replacement for a JCSP channel between implementation and modelling.

We believe this outcome is likely across all standard (e.g., deterministically scheduled) preemptive runtimes implementing CSP channels when under constrained resources. One side of the communication will complete first, leading to unexpected priorities. More work is required to build a suitable execution environment for a preemptively scheduled runtime to fully test these beliefs. In CSP, all processes participating in a communication event continue together (the event is atomic to all participants). In an implementation, without hardware support, we find this unlikely as only certain participants will have hardware executing them to enable those processes to continue in parallel. A process within a deterministic preemptively scheduled runtime will not yield the hardware until its time slice has ended — prioritising certain parts of the system accordingly. In JCSP terms, this would be the reading end of a channel communication being prioritised.

8.3 Examination of Channel Interactions

Let us now examine the interaction between a reading and writing process during the communication handshake undertaken by a channel. Both ProcessJ and JCSP will be presented[11]. We consider each step of the interaction, illustrating when the writing process (WRITE) and the reading process (READ) are blocked or runnable. As these interactions take place during locked sections with notifications, the reordering of operations is limited and execution ordering (through the channel communication ping-pong) is known. We present both when the writing process arrives at the channel first and when the reading process arrives at the channel first. For ProcessJ, we present the interactions as if there is a single scheduling (runner) thread in operation due to the potential for more concurrency due to reduced locking in comparison to JCSP. In Table 3 and Table 4, ■ indicates when a process is blocked from execution, whereas ▷ indicates where a process is able to continue execution.

Table 3 presents the interactions during channel communication with ProcessJ. Note that for the situation where the WRITE process accesses the channel first, at step 11 WRITE is free to continue as it has been scheduled by READ (step 10) and will now wait trying to lock the channel. READ will eventually yield (step 15), meaning with only a single processor WRITE can continue without restriction, prioritising its behaviour. Similarly, when READ arrives first, WRITE will be free to continue at step 15, while READ will yield at step 19. With two or more runner threads, write and

[11]See Appendix B for the JCSP code-snippets.

Table 3. Singularly Scheduled ProcessJ Channel Communication

Write first.

Step	WRITE	READ
1	LOCK CHANNEL	■
2	SET HOLD	■
3	SET WRITER	■
4	SET WRITER NOT READY	■
5	UNLOCK CHANNEL	■
6	YIELD	▷
7	■	LOCK CHANNEL
8	■	UNLOCK CHANNEL
9	■	LOCK CHANNEL
10	■	SCHEDULE WRITER
11	▷	SET WRITER TO NULL
12	▷	SET READER TO NULL
13	▷	TEMP STORE HOLD
14	▷	UNLOCK CHANNEL
15	▷	YIELD
16	RETURN	▷
…	▷	RETURN TEMP STORE

Read first.

Step	WRITE	READ
1	■	LOCK CHANNEL
2	■	SET READER
3	■	SET READER NOT READY
4	■	UNLOCK CHANNEL
5	▷	YIELD
6	LOCK CHANNEL	■
7	SET HOLD	■
8	SET WRITER	■
9	SET WRITER NOT READY	■
10	SCHEDULE READER	■
11	UNLOCK CHANNEL	■
12	YIELD	▷
13	■	LOCK CHANNEL
14	■	SCHEDULE WRITER
15	▷	SET WRITER TO NULL
16	▷	SET READER TO NULL
17	▷	TEMP STORE HOLD
18	▷	UNLOCK CHANNEL
19	▷	YIELD
20	RETURN	▷
…	▷	RETURN TEMP STORE

Table 4. JCSP Channel Communication

Write first.

Step	WRITE	READ
1	LOCK CHANNEL	■
2	SET HOLD	■
3	SET EMPTY TO FALSE	■
4	WAIT (UNLOCK CHANNEL)	■
5	■	LOCK CHANNEL
6	■	SET EMPTY TO TRUE
7	■	NOTIFY
8	■	RETURN HOLD
9	■	UNLOCK CHANNEL
10	LOCK CHANNEL	▷
11	UNLOCK CHANNEL	▷
12	RETURN	▷

Read first.

Step	WRITE	READ
1	■	LOCK CHANNEL
2	■	SET EMPTY TO FALSE
3	■	WAIT (UNLOCK CHANNEL)
4	LOCK CHANNEL	■
5	SET EMPTY TO TRUE	■
6	NOTIFY	■
7	WAIT (UNLOCK CHANNEL)	■
8	■	LOCK CHANNEL
9	■	NOTIFY
10	■	RETURN HOLD
11	■	UNLOCK CHANNEL
12	LOCK CHANNEL	▷
13	UNLOCK CHANNEL	▷
14	RETURN	▷

read can make progress together, as the final yields by READ are a courtesy with the process still able to continue if resources are available.

Table 4 presents the interactions in JCSP when the channel communication occurs. Note that the opportunity for concurrency is reduced due to locking of the channel through synchronised methods, leading to WRITE being blocked from continuing until READ has completed (step 10 when write arrives first, and step 12 when read arrives first). READ is able to continue freely after it releases the lock.

With enough resources, it can be seen that both ProcessJ and JCSP have the opportunity for concurrent continuation. However, if resources are limited, then a more rigid ordering will be followed. For example, with ProcessJ we know that the writing process will always be prioritised leaving the channel. In JCSP, the reading process is prioritised. In JCSP, READ will notify WRITE that it can continue, but WRITE is blocked until READ unlocks the channel. As such, READ can continue freely until it is preempted. As READ was only recently provided resources (i.e., step 5 when WRITE comes first, step 8 when READ comes first), and only four operations occur (two for locks, one to notify), a preemptive scheduler would have to have aggressively short time slices to preempt READ at this point. As such, these priorities on write and read will occur under resource

constraints, and as such certain traces are not possible. This prioritisation will likewise propogate through a system, as illustrated in the empirical evidence presented in Section 8.2.

9 Discussion and Conclusion

Although our results demonstrate ProcessJ's channel and choice working correctly within certain operating conditions (or fully when not considering restricted operating conditions as demonstrated in our previous work [PC23], much like other papers on channel system modelling), this was not the ultimate goal of our work. Indeed, we had previously verified our primitives as correct outside their execution environment. Our primary aim was to ensure ProcessJ would work when multithreading was introduced to the scheduling system. Our previous paper [PC23] did not fully succeed, although, we were confident the divergence was not an issue for a system to make progress. In this paper, we have proposed a change to the ProcessJ scheduling runtime to remove divergence, and therefore verified we can model (and soon implement) an improved ProcessJ scheduling system. The work we have presented is on the verification of the scheduling environment that the channel and choice mechanisms must work within.

However, our work has demonstrated limitations with scheduling ProcessJ processes, which we believe will be evident in other channel-based implementations. Full specification verification is only possible if there are enough resources (i.e., processors) to run all active processes. This is demonstrated in our channel results (see Table 1), where full specification verification of a channel is only possible with two or more schedulers. Likewise, for choice, three or more schedulers are required for full system verification of a two-branch alt (with three active processes), and four schedulers for a three-branch alt (with four active processes). In both cases, the amount of hardware available **must** be equal to or greater than the number of active processes for full verification. We consider this a general problem—no real-world implementation of a channel-based runtime can meet the abstract specification of a channel unless there are enough processors available.

The problem we are presenting is illustrated in Figure 7. Welch and Martin [WM00] have algebraically proven their channel specification is the same as a pure CSP channel. We do not argue with this proof, and indeed use this specification ourselves. However, we do not believe a claim that a channel implementation fully meets Welch and Martin's channel specification is sound in all circumstances as demonstrated in our work. There is an additional level of indirection that has been introduced. The indirection occurs due to the claim that the channel implementation will refine the channel specification under failures-divergences, demonstrating a behavioural equivalence to a CSP channel. Welch and Martin argued that as their model of the JCSP channel implementation refined their channel specification, the JCSP channel could be replaced by a CSP channel when modelling. Thus, a system specification that used CSP channels could check a system implementation that used JCSP channels without modelling the full JCSP channel implementation. Only CSP channels were necessary.

However, if the channel implementation does not meet the specification in failures-divergences, then likewise any system implementation will not meet the system specification if only CSP channels are used to model actual channel implementations. As we have argued, meeting Welch and Martin's channel specification when considering the execution environment of the implementation is problematic. Thus, a broad claim of using a CSP channel to model an actual channel implementation to verify a system will not account for the unexpected prioritisation that occurs at the end of the channel implementation's ping-pong of synchronisation.

The additional level of indirection for verification has introduced potential unintended behavioural limitations. Problems will occur if a system is modelled and verified in CSP using the argument that the channel implementation can be replaced with a standard CSP channel. Our work demonstrates that this is only the case *if* enough processors are available to run all active processes.

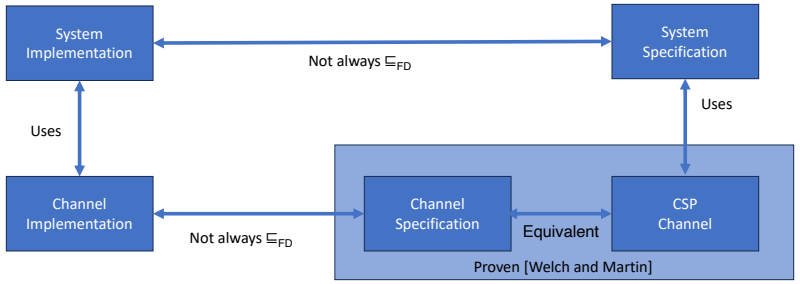

Fig. 7. Indirection of Verification with a Channel Implementation.

The same argument can be made for choice. This is a limitation of implementations without enough processors or the assumption that scheduling will provide all possible outcomes.

When taking a component-based approach to specification and verification (e.g., see [GRBC+17], Section 6), we can replace an implementation with its specification when the implementation has been verified correct within the failures-divergences model. Taking this approach with a channel implementation while not considering the execution environment is a concern. Claiming part of a system meets its specification through using a CSP channel in place of a channel implementation can lead to certain behaviours not being possible. For example, in ProcessJ, we know that writing is prioritised during a channel communication, and likewise JCSP prioritises reading. Without sufficient resources, this will mean that certain traces are not possible in the implementation in a live environment (and thus the failures-divergence verification does not hold in all circumstances). Such unintended priorities will lead to certain behaviours not occurring, thus creating unintended starvation in the system. This starvation can possibly lead to deadlock for parts of the system which are unserviced as some event traces are prioritised over the general set of all event traces.

With ProcessJ, we know explicitly the order that events will occur in a singularly scheduled system. Behaviour will be completely determined when the system is started—there will be only one trace of possible behaviour. A read-write handshake will always ensure the writing process will continue first (a fuller explanation is provided in our previous work [PC23]). Such assurance will mean any follow-on behaviour will be based on the writing process making progress before the reading process. Such behaviours will propagate through the system. As certain behaviours will thus be prioritised unintentionally, unexpected starvation can occur due to parts of the system not making fair progress as assumed in the specification.

We likewise believe such prioritisation will occur in a preemptively scheduled runtime. For example, in JCSP the reading process always exits the communication handshake first. Although preemptive scheduling means the writer will be allowed to make progress without having to wait for the reader to yield, the read process will have priority to progress immediately after the handshake. As the handshake involves a ping-pong of waits, notifies and locking, the reader leaves the handshake having only recently been allocated processor time, and is unlikely to be descheduled for the writing process. Likewise this priority will propagate through the system, and may lead to potential starvation scenarios. Preemptive scheduling is obviously less sensitive to such occurrences, and therefore the limitations of cooperative scheduling become clearer in a limited resource situation.

The potential issues we are highlighting do not lead directly to deadlock in a system. A system will still make progress, but only in a subset of any behaviour defined in a specification. Other implementations of CSP primitives have likewise been demonstrated to lack certain behaviours (e.g.,

see Section 8.4 of [Cha09] demonstrating the limitations of JCSP's multiway sync [WBM+10]). We have implementations that trace refine specifications, but do not provide equivalent behaviours in a manner to ensure confidence in using pure CSP modelling for a system that relies on channel-based communication.

Although our work has demonstrated a behaviour limitation in implementations of concurrency primitives, limiting the assurance that equivalent behaviour is possible between model and implementation, it does not mean such runtimes cannot be used. Our modelling demonstrates that the number of schedulers must equal the number of processes in a system to achieve full failures/divergence refinement in both directions. However, this will not be the case in a real system. A real system will only have a subset of its processes ready to run at any one time, and if all these processes can be serviced, we will reach specification equivalence.

On a single processor/threaded system, such as ProcessJ's current implementation, we will have limited behaviour to the point of potential singular trace behaviour. However, as the number of processors is increased, the more specification traces are possible. As the number of processors approaches the number of potentially ready processes, the specification and implementation trace sets will become closer to equivalent. When the number of processors equals the number of potentially ready processes, all system traces will be available. An interesting question we plan to explore is the potential relationship between the amount of resources available and the completeness of the implementation traces compared to the specification.

Given our discussion, we have demonstrated why we needed to create a suitable multithreaded scheduling system for ProcessJ. The current implementation is singular in its behaviour, thus negating many of the potential behaviours we may expect when designing concurrent systems. By introducing a multithreaded scheduler, we will enable a larger set of potential trace behaviours to occur. This work has clearly demonstrated this need which was not apparent in the standard verification approach of the ProcessJ concurrency primitives.

9.1 Conclusion

We now have verified the implementation of both channel communication and choice in ProcessJ and we have shown that with enough schedulers (≥ 2 for channel communication, ≥ 3 for a two-branch choice, and ≥ 4 for a three-branch choice), a cooperatively scheduled system (a *restricted* system) behaves **exactly** like an *unrestricted* non-scheduled one. We have *failures/divergence* results for all checks.

In the situation where the number of schedulers is less than the number of processes, we have shown that the system is deadlock free and will behave as specified. The lack of F/FD results in these cases have no impact on the correctness (i.e., the ability of the implementation to meet part of the specification behaviour) but are solely caused by execution behaviours imposed by the order of the processes in the run queue. It may slow down the execution compared to a system with an ample amount of schedulers as some processes need to cycle the run queue as they may be ahead of other processes whose prior execution is required in order to make progress. For example, with a single scheduler and a channel communication in which the reader is ahead of the writer in the run queue, the dequeuing sequence for a complete communication to happen will be: reader, writer, reader, writer, reader. For the same system but with the writer first in the queue, the dequeuing sequence is: writer, reader, writer, reader. As we can see, the first sequence has an additional dequeue of a reader at the beginning which eventually enqueued the reader to the back of the queue, and execution can progress as in the second example where the writer was first.

The execution environment is important. This has been demonstrated across two papers: in the first paper we did not get the results that we expected, but it was not because the implementation of the channel communication or choice was incorrect (we verified them using the standard approach),

but simply that the execution environment was not suited to support the expected behaviour. Therefore, in this paper, we made changes to the execution environment in order to bring the results of the verification of the channel communication and choice into line with what we expected from the standard approach. This strongly supports our assertion that verification should never be done without taking the execution environment into account.

This is also demonstrated by the results when there are not enough resources to achieve maximal concurrency. This results in the specification not being fully met. Whether or not this is a problem must be further investigated, but it may leave a concern that certain behaviours in the abstract system may not happen in the concrete system. When considering the concrete system interacting with other components of a larger system, we may have issues that verification against the abstract system may not identify. When creating concurrency primitives, such as a channel, and arguing behaviours are equivalent to a formal specification, significant limitations may result.

10 Future Work

The first step is to implement the results from this paper in the actual ProcessJ runtime/scheduler. The only challenging issue is the queue which must be blocking (without spinning) on the dequeue operation. However, the *java.util.concurrent* package contains a *LinkedBlockingQueue* class that serves exactly the purpose we need. See Appendix A for a prototype Java outline for how we plan to implement the new scheduling system in ProcessJ.

Clearly, one of the design goals of a concurrent system is speed and efficiency, not only correctness. However, a speedy system that is not correct is of no use to anyone. If we study our solution closely, we see that a number of locks are required to ensure safe and correct operation. There are locks on:

- An entire process; this is necessary to correctly handle setting the *ready* flag and placing processes in the run queue.
- An entire channel; this is necessary in order to avoid race conditions when reading and writing to the channel.

Inevitably, locking forces less concurrency, thus potentially making a system slower. The next step in developing the ProcessJ scheduler and runtime will therefore be to reduce the need and dependence on locking objects and data structures as far as possible. We believe that we can remove most of the locking we currently have by making use of the following two techniques:

- Atomic variables: atomic variables (fields) support compare-and-swap. That is, a compare-and-swap operation allows for comparing the variable (v) to a value (m) and, if the comparison succeeds, set v to a new value (n).
- Concurrent queue: a concurrent queue allows multiple processes to access it safely at the same time, but which also must be blocking when the queue is empty.

We believe that all the primitive fields (e.g., channel *data* store, process *ready* field etc.) can be correctly implemented using atomics.

Naturally, to support these claims, we need to develop CSP models for both atomic variables and a concurrent queue—this will be the next step in this story.

In addition, we also want to verify the correctness of shared channels in ProcessJ.

Our main argument is that to truly verify a concurrency runtime, the execution environment must be modelled. In the introduction, we stated two choices were possible when an issue in the model was discovered:

(1) Change the implementation of the synchronisation primitives to work better in the execution environment.
(2) Change the execution environment so it worked correctly with the synchronisation primitives.

To fully explore these issues, a preemptive scheduling approach in CSP would need to be developed, and then used to verify JCSP and other concurrency runtimes. We believe the spurious wakeup problem would lead to further consideration on how to ensure correctness of the JCSP runtime, which would likely mean updates to the synchronisation primitives as the environment would need to be changed. Or, it could mean the spurious wakeup problem could be solved using the execution environment model to discover a solution.

Acknowledgement

We would like to thank the reviewers for their helpful feedback, especially for the comments focusing on improving and optimizing the CSP.

References

[Cha09] Kevin Chalmers. *Investigating Communicating Sequential Processes for Java to Support Ubiquitous Computing*. PhD thesis, Edinburgh Napier University, 2009.

[Cis18] Benjamin Cisneros. ProcessJ: The JVMCSP Code Generator. Master's thesis, University of Nevada Las Vegas, 2018.

[CR22] Milind Chabbi and Murali Krishna Ramanathan. A Study of Real-World Data Races in Golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 474–489, New York, NY, USA, 2022. Association for Computing Machinery.

[CT09] Francesco Cesarini and Simon Thompson. *Erlang programming: a concurrent approach to software development*. O'Reilly Media, Inc., 2009.

[CY23] Yi-An Chen and Yi-Ping You. Structured concurrency: A review. In *Workshop Proceedings of the 51st International Conference on Parallel Processing*, ICPP Workshops '22, New York, NY, USA, 2023. Association for Computing Machinery.

[FDR] FDR 4.2.7 Documentation. https://cocotec.io/fdr/manual. Accessed: 2023-08-23.

[GRABR14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201, 2014.

[GRBC+17] Thomas Gibson-Robinson, Guy Broadfoot, Gustavo Carvalho, Philippa Hopcroft, Gavin Lowe, Sidney Nogueira, Colin O'Halloran, and Augusto Sampaio. *FDR: From Theory to Industrial Application*. Springer International Publishing, 2017.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, January 1987.

[Low19] Gavin Lowe. Discovering and correcting a deadlock in a channel implementation. *Formal Aspects of Computing*, 31(4):411–419, August 2019.

[MI09] Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2), February 2009.

[MS95] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Technical report, Rochester University, Department of Computer Science, 1995.

[Ora] Oracle. Oracle Help Center. https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/locks/Condition.html. Accessed: 2022-12-09.

[PC19] Jan B. Pedersen and Kevin Chalmers. Verifying Channel Communication Correctness for a Multi-core Cooperatively Scheduled Runtime Using CSP. In *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering*, FormaliSE '19, pages 65–74, Piscataway, NJ, USA, 2019. IEEE Press.

[PC23] Jan B. Pedersen and Kevin Chalmers. Towards Verifying Cooperatively-Scheduled Runtimes using CSP. *Formal Aspects of Computing*, 1(1), January 2023.

[PW18] Jan B. Pedersen and Peter H. Welch. The Symbiosis of Concurrency and Verification: Teaching and Case Studies. *Formal Aspects of Computing*, 30(2):239–277, March 2018.

[Reg02] J. Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 315–326, 2002.

[Ros98] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998. The text book teaching material can be found at http://www.comlab.ox.ac.uk/publications/books/concurrency/.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[RSB09] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore scheduling for lightweight communicating processes. In John Field and Vasco T. Vasconcelos, editors, *Coordination Models and Languages*, pages 163–183, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[SP16] Cable Shrestha and Jan B. Pedersen. JVMCSP—Approaching Billions of Processes on a Single-Core JVM. In Kevin Chalmers, Jan B. Pedersen, Jan F. Broenink, Brian Vinter, and Peter H. Welch, editors, *Proceedings of Communicating Process Architecture 2016*, Amsterdam, The Netherlands, August 2016. IOS Press.

[Suf08] Bernard Sufrin. Communicating Scala Objects. In *CPA*, volume 66, pages 35–54, 2008.

[Tis00] Christian Tismer. Continuations and stackless Python. In *Proceedings of the 8th international python conference*, volume 1, 2000.

[WBM+10] Peter Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Alting barriers: synchronisation with choice in Java using JCSP. *Concurrency and Computation: Practice and Experience*, 22(8):1049–1062, 2010.

[WM00] Peter H. Welch and Jeremy M. R. Martin. Formal Analysis of Concurrent Java Systems. In Peter H. Welch and André W. P. Bakkers, editors, *Communicating Process Architectures 2000*, volume 66, WoTUG-31, pages 275–301, Amsterdam, The Netherlands, sep 2000. IOS Press. ISBN: 978-1-58603-907-3.

A Java Code

In this section we present real Java code implemented using the verified CSP. The *Process* class is basically the *PJProcess* class from the ProcessJ runtime. Note, the use of monitors in the CSP code is mirrored by declaring methods *synchronized* and by using *synchronized* blocks. Additionally, the *LinkedBlockingQueue* is thread safe.

A.1 ScheduleManager

```
import java.util.concurrent.*;
import java.util.*;

class ScheduleManager {

    // This is the process queue - only processes that are ready are in this queue.
    static BlockingQueue <Process>  queue = new LinkedBlockingQueue <>();

    // This is a hash set of all the schedulers
    static HashSet <Thread>  schedulers = new HashSet <Thread>();

    // Number of processJ processes on the heap.
    static int processCount = 0;

    // This method adds a process to the queue.
    public static synchronized void schedule(Process p) {
        try {
            queue.put(p);
        } catch (Exception e) { }
    }

    // Increment the number of processes.
    public static synchronized void processCountInc() {
        processCount++;
    }

    // Decrement the number of processes.
    // If all processes have terminated, terminate all the schedulers
```

```
    public static synchronized void processCountDec() {
        processCount−;
        if (processCount == 0)
            new Thread() {
                public void run() {
                    ScheduleManager.shutDown();
                }
            }.start();
    }

    // Create a new Scheduler
    public static void newScheduler() {
        Scheduler s = new Scheduler();
        schedulers.add(s);
        s.start();
    }

    // Shut down all the active scheduler.
    public static void shutDown() {
        for (Thread t : schedulers)
            t.interrupt();
    }
}
```

A.2 Scheduler

```
class Scheduler extends Thread {
    public void run() {
        while (true) {
            Processp = null;
            try {
                // get a new process to run
                p = Manager.queue.take();
            } catch (InterruptedException ie) {
                return;
            }
            // run the process
            p.run();
        }
    }
}
```

A.3 Process (PjProcess)

```
class Process {
    // Ready field
    private boolean ready = true;
```

```
    // Running field
    private boolean running = false;

    // Method for scheduling a process
    public synchronized void schedule() {
      if (!ready) {
        ready = true;
        if (!running)
          ScheduleManager.schedule(this);
      }
    }

    // Method for descheduling a process
    public synchronized void deschedule() {
      running = false;
      if (ready)
        ScheduleManager.schedule(this);
    }

    // Constructor
    Process(int pid) {
      this.ready = true;
      ScheduleManager.processCountInc();
      ScheduleManager.schedule(this);
    }
  }
```

A.4 An Application Process

The class *Foo* below is a (somewhat abstract) example of a compiled ProcessJ procedure.

```
  class Foo extends Process {
    public void run() {
      // Processes always set themselves to running at the start and after each yield point.
      // Since every re-activation starts here setting running to true here is enough.
      synchronized (this) {
        running = true;
      }

      // This is the switch that jumps to the appropriate place in the application code below.
      switch(runLabel) {
        ...
      }

      // Application code
      ...

      // At every yield point:
      this.deschedule();
```

```
            // At termination:
            ScheduleManager.processCountDec();
            return;

        }
    }
```

B JCSP Read and Write Code

In this section we present the relevant JCSP code for reading and writing. The complete code can be found at:
github.com/JonKerridge/jcsp/blob/master/src/main/java/jcsp/lang/One2OneChannelImpl.java
First the code for writing:

```
    synchronized (rwMonitor) {
        hold = value;
        if (empty)
            empty = false;
        else {
            empty = true;
            rwMonitor.notify();
        }
        rwMonitor.wait();
    }
```

and now the code for reading:

```
    synchronized (rwMonitor) {
        if (empty) {
            empty = false;
            rwMonitor.wait();
        } else
            empty = true;
        rwMonitor.notify();
        return hold;
    }
```