Verifying Correctness of Shared Channels in a Cooperatively Scheduled Process-Oriented Language

Jan Pedersen[a,*], Kevin Chalmers[b]

[a]*Department of Computer Science, University of Nevada Las Vegas, 4505 South Maryland Parkway, 89154, Las Vegas, NV, USA*
[b]*School of Computing, Architecture, and Emerging Technologies, Ravensbourne University, 6 Penrose Way, SE10 0EW, London, UK*

Abstract

Correct concurrent behaviour is important in understanding how components will act within certain conditions. In this work. we analyse the behaviour of shared communicating channels within a coorporatively scheduled runtime. We use the refinement checking and modelling tool FDR to develop both specifications of how such shared channels should behave and models of the implementations of these channels in the cooperatively scheduled language ProcessJ. Our results demonstrate that although we can certainly implement the correct behaviour of such channels, the outcome is dependant on having adequate resources available to execute all processes involved. We conclude that modelling the runtime environment of concurrent components is necessary to ensure components behave as specified in the real world.

Keywords: CSP, process-orientation, processJ, formal verification, cooperatively scheduler, verified runtime

*Corresponding Author

1. Introduction

The correctness of software is an important issue. There are two main approaches to ensuring correctness: testing and formal verification. Almost all software is tested (more or less) rigorously until such time that the developer is convinced that most of the serious bugs have been eradicated. However, testing is not always enough. In [1] a simple piece of software for a made-up Martian rover was considered. The software could end up in a deadlocked configuration (because of an inversion error) but the authors calculated that "...we would need to wait towards 8 million seconds (over 90 days) to reach a 50% chance of seeing that deadlock triggered." This illustrates that using testing alone to ensure the correctness of software is simply not enough. The Mars rover system from above was formally verified using CSP [2, 3] (Communicating Sequential Processes) and the formal model checker FDR [4], and the inversion error was found immediately and FDR produced an execution trace explaining how the error could occur. This is just one example of how formal verification can find errors and bugs that testing alone may never encounter. Formally verified correct software becomes even more important when we consider runtime systems associated with programming languages: since a large number of programs will be built upon the language's runtime, it is vitally important that it (the runtime) is bug free and behaves according to its specification.

In this paper we utilise CSP, a process algebra defined using processes, channels, and choice. It supports process composition using a parallel and interleaving operators. Section 4 goes into more detail about CSP and the various semantic models that can be used to check refinement between processes. This paper models the implementation of shared channels in the ProcessJ runtime, demonstrating their correctness against a well-formed and proved specification. However, we also present limitations encountered due to the limitations of scheduling.

ProcessJ is a process-oriented programming language being developed at the University of Nevada Las Vegas and the University of Roehampton. ProcessJ has non-buffered synchronous channels for communication and is highly concurrent. It is cooperatively scheduled, running on the JVM. This means that the runtime system for ProcessJ is written in Java; that is, the code that handles scheduling and communication etc. is implemented in Java. It is extremely important that this code (we will collectively refer to as "the runtime") is correct—that is, it should never deadlock or livelock and it should behave according to a predetermined specification describ-

ing what acceptable behaviour is. In [5] we started out this verification journey to prove the behaviour for the ProcessJ runtime correct. In this initial approach we translated the ProcessJ runtime code for channel communication to CSP and proved it to behave correctly when comparing to a well-established specification. In [6] we added verification of choice and realised that the *standard approach* used for verification was insufficient—it fails to take the code for the runtime system (i.e., schedulers, runners etc.) into account. We showed that this is crucial to do as the behaviour of a *scheduled runtime* becomes limited in terms of possible behaviours. With this in mind we redid the verification of the existing scheduler of ProcessJ and realised that there were some issues with the implementation (live-lock in the scheduler). These issues were tackled in [7] and we showed that both one-to-one channel communication and alternation (choice) now work correctly with the the new scheduling system.

Now, with a well-behaved scheduling and execution system, it is time to attack the verification of the remaining parts of the runtime system. In this paper we consider shared channels; that is, channels with multiple readers and/or multiple writers.

In [5] and [6] we started with an actual implementation which we translated to CSP in order to perform the verification. In [7] we started with a specification and then implemented the verified scheduler in Java. In this paper we again switch back to translating existing code to CSP for verification (using the new verified scheduling system).

1.1. Motivation

A shared channel is one where multiple readers and/or writers may be engaged with a channel. The individual readers/writers do not synchronise amongst themselves and only synchronise with one partner (i.e., one reader and one writer synchronise) during communication. Such behaviour is allowed in CSP and an example is provided in Section 8. Due to implementation challenges when considering choice, the approach is to provide shared channels in CSP inspired implementations that differ from one-to-one channels. In ProcessJ, we do not rely on preemptive scheduling and thus have implemented a queuing mechanism for mutual exclusion on the shared channel ends. Our aim is to demonstrate that our implementation behaves as would be expected for such a channel.

In the standard approach to channel-based system verification, no runtime system, execution system, hardware etc. is modelled. Events in the formal model can happen whenever the system is ready to perform the

event. The verification is in the abstract rather than in a concrete environment. We argue that such an approach does not model what happens when a channel-based system is run in the real-world. 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 resources are available to execute the event when it 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. This assumption that resources are not scarce is not true. 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.

1.2. Contribution

The main contributions of this paper fall into three parts:

- We have verified that the ProcessJ implementation of shared channels using the existing cooperating scheduler in the ProcessJ runtime is correct. That is, they behaves as expected by virtue of passing refinement checks using a CSP translation of the implementation and a CSP generic model of a shared channel.
- We have developed an extended algebraic proof that the shared channel specification indeed does behave like a CSP shared channel.
- Finally, we have further demonstrated the relationship between available resources and active processes for meeting specifications. Again, we have shown that in order to obtain refinement between specification and model in both directions, the number of available schedulers[1] must be equal to or larger than the number of processes in the system.[2]

1.3. Channel Communication

In a process-oriented language like ProcessJ, communication between processes happens though non-buffered synchronous channels. The sending process performs a blocking write with a value on the channel by using the

[1]Note, a *scheduler* is really a *runner*; something that executes a ProcessJ process. We have in prior papers referred to these entities as schedulers, and carry on with that nomenclature in this paper as well.

[2]Or at least enough in the way that any ready process can be run immediately.

writing end. The reading process performs a blocking read on the *reading end*. When both the writer and the reader are ready, they synchronise and the value is exchanged. In [6, 7] we considered the correctness of one-to-one channels; that is, channels with exactly one sender and exactly one receiver. In this paper we introduce channels with shared channel ends to ProcessJ. This includes channels with a single writing end and multiple reading ends; multiple writing ends and a single reading end; and multiple writing ends and multiple reading ends.

A one-to-one channel carrying values of type *type* (named *c*) in ProcessJ is declared as follows:

chan<*type*> *c*;

Channels with shared ends are declared similarly but the keyword **shared** is used to denote the end which is shared:

shared write chan<*type*> *sharedWriteChan*;
shared read chan<*type*> *sharedReadChan*;
shared chan<*type*> *sharedWriteAndReadChan*;

reading and writing to a non-shared or shared channel end is similar; here are both reading and writing:

c.**read**()
c.**write**(*expr*)

When using a shared end of a channel, the ProcessJ compiler generates the correct protection to avoid concurrent access and other race conditions. In this paper we validate this compiler-generated runtime code along with the runtime scaffolding like access to queues of reader and writers all waiting to use a shared channel end. A more comprehensive example of a one-to-many (shared reading end) and a many-to-one (shared writing end) can be found in Figure 2. A simple one-to-one channel use can be found in Section 3

1.4. Breakdown of the Paper

In Section 2 we present a short discussion about scheduling as well as present related and prior work. Section 3 presents the ProcessJ programming language, and Section 4 gives a brief introduction to Communicating

Sequential Processes (CSP). In Section 5 we lay the groundwork for this paper by introducing a model for a scheduler and a process. We then continue on in Section 6 by illustrating the use of and translation in to Java of ProcessJ shared channels. This translation into Java is further translated into CSP in Section 7. A generic channel supporting multiple readers and writers is specified in CSP in Section 8 and a formal proof this specification is given in Section 10. Finally, Sections 9 and 11 contain the results and a discussion of them. We close with future work in Section 12.

The CSP code presented in this paper, along with all the tested assertions, can be found in the GitHub repository located at `http://github.com/mattunlv/process-shared-csp`.

2. Background Related Work

In this section we start with some background information about cooperative scheduling and then consider the results from [6].

2.1. Cooperative Scheduling and Multitasking

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

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

With no priorities, preemptive scheduling guarantees processes will be allocated processor time. Cooperative scheduling makes no such guarantee as a process may use resources indefinitely without yielding. In both cases, 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, we can use a simple approach to cooperative scheduling—a continuous dequeing of the next process to execute until the queue is empty. In a multiprocessor system, consideration of cross-core synchronisation is required. Often, and the case in 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.

There are advantages to cooperative scheduling. For example, in ProcessJ, processes are stackless; a process' local data is stored as object data. Therefore, a process is incredibly lightweight. Previous ProcessJ publications [8] have demonstrated hundreds of millions of processes in operation. This compares to preemptive thread-based libraries, such as JCSP, supporting only a few thousand processes. Large stack sizes (often greater than a megabyte) that each thread utilises increases memory requirements.

ProcessJ also has fast context switching due to its lightweight processes [8]. As no preemptive thread swapping is required, context switches, and thus channel communication time, is fast. A thread provided by the operating system will 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* has limited popularity due to the programmer typically having to manage scheduling. Operating systems favour preemptive scheduling. 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, will alleviate some of the challenges in managing such scheduling. Indeed, until version 1.10, the Go 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 [9] which was likewise hidden from the programmer.

In contrast, cooperative *multitasking* has a growing popularity, primarily from the *async/await* pattern. Coroutines [10] 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 [11] 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 [12, 13]. Indeed, the only similar runtime approach—hiding cooperation in synchronisation—we have found is in Stackless Python [14].

2.2. Related Work

Verification of runtimes has been undertaken for other systems like CSO [15] and JCSP [16] as well as the regular non-shared channels and choice for ProcessJ [6]. Similarly, other systems have been formally verified using CSP/FDR or other similar verification systems like SPIN (for example see [17]).

Mutual-exclusion is long-solved (e.g., see [18]). Model checking concurrent algorithms often begins with the assumption that multi-processor behaviour is typically managed via a preemptive scheduler. Atomic operations introduce new challenges for mutual exclusion, but again, the preemptive scheduler enables certain assumptions (e.g., see [19]). The assumption of preemptive scheduling has been successfully applied to model check synchronous channel communication [20, 21] in JCSP [16] and Communicating Scala Objects (CSO) [15]. JCSP and CSO both rely on preemptive scheduling provided via Java threads. The ProcessJ scheduler more closely emulates the occam-π multi-processor cooperative scheduler defined by Ritson et. al. [9].

However, as we emphasized in [6], all of those previous attempts have overlooked the *execution environment* (scheduler, hardware processors, etc.) when performing the verification. The verification that we performed in [6] was done not only in the standard way (not taking into account the execution environment), but also taking into account the cooperative scheduler. This is how we demonstrated the livelock in the original ProcessJ scheduler.

2.3. Our Work So Far

In [5] and [6] we performed verification of channel communication using one-to-one channels, first in the standard unrestricted (non-scheduled) manner, and then in a restricted manner, where the code of the runtime system and the scheduler was taken into account. In the unrestricted (non-scheduled version) we achieved the results that we expected. We then introduced a simple scheduler:

$Queue\langle Process\rangle\ runQueue;$
...

```
// 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);
}
```

When verifying our code taking this scheduler into account, the verification software determined that our code could livelock. This was caused through the run queue containing both ready and non-ready processes, and with more than one scheduler, it could be possible that one of more of the schedulers never executed any code but simply dequeued and re-enqueued non-ready processes forever. In [7] we fixed this issue by implementing an improved scheduler and removing the non-ready processes from the run queue. We could now perform the verification again and get the results we were aiming for. The aim for this paper is to succeed in the same manner with a scheduled system using channels with shared ends.

3. ProcessJ

ProcessJ [8] is a process-oriented programming language being developed at the University of Nevada Las Vegas. The syntax of ProcessJ resembles Java, with new constructs for synchronisation. It has CSP semantics similar to occam/occam-π [22, 23]. The ProcessJ compiler generates Java source files that are further compiled to produce class files, which in turn are rewritten using the ASM [24] bytecode rewriting tool to create cooperatively schedulable Java bytecode. Targeting the JVM ensures maximum portability, and tests have shown that a computer can handle upwards of half a billion processes on a single core [8].

ProcessJ supports four different channel types: one-to-one, one-to-many, many-to-one, and many-to-many. In this paper we tackle the correctness of the last three channel types.

Below is a simple ProcessJ example (with a single one-to-one channel for simplicity). The *main* method starts two processes (*reader* and *writer*) concurrently; *reader* is passed the reading end of a channel, and *writer* is passed the writing end of the same channel. The writing process sends the value 42 to the reading process, which in turn prints out the value. Once both these processes have terminated, the *main* process prints "Done".

```
public void reader(chan<int>.read in) {
    int x = in.read( ); // read a value from the in channel
    println("received:" + x); // print the received value
}

public void writer(chan<int>.write out) {
    out.write(42); // write the value 42 on the out channel
}

public void main(string args[ ]) {
    chan<int> c; // declare a channel
    par { // in parallel, run the following two processes:
        writer(c.write); // a writer w/ the writing end
        reader(c.read); // a reader w/ the reading end
    }
    println("Done");
}
```

3.1. Processes

The ProcessJ compiler outputs Java source code that links with a runtime systems. Every ProcessJ procedure becomes a subclass of the following ProcessJ runtime *PJProcess* class:

```
class PJProcess {
    int runLabel = 0;
    bool ready = true;
    bool running = false;

    public synchronized void setNotReady() {
        ready = false; // Set the process not-ready
    }

    public synchronized void setReady() {
        ready = true; // Set the process ready
    }

    public void run() { }
```

```
    // Dummy methods for ASM
    public void yield(int label) { }
    public void label(int label) { }
    public void resume(int label) { }
}
```

setNotReady() and *setReady*() set a process *not-ready-to-run* and *ready-to-run*, respectively. For example, the *reader* procedure would generated the following code:

```
public class reader extends PJProcess {
    private int x; // local ProcessJ variable x

    public void run() {
        switch(runLabel) {
        case 0: resume(0); break;
        ...
        }

        // code for x = in.read();
        // code for println("received:" + x);
    }
}
```

All local variables of *reader* are transformed into fields of the class; this way there is no need to maintain a stack of locals between invocations of the *run*() method. The process-flow of the ProcessJ compiler is as follows:

- The ProcessJ compiler reads a ProcessJ source file and produces, as output, a number of Java source files; one per ProcessJ procedure. This code contains invocations of three dummy methods: *yield*(), *resume*(), and *label*(), that act as placeholders for the later instrumentation phase.
- The generated Java source files are compiled with the Java compiler and class files are produced.
- The three dummy methods *yield*(), *resume*(), and *label*() are used and transformed in the generated code as follows:

- *resume*(i) is replaced by a '`goto` address of *label*(i)' when the bytecode rewriting is performed.
- *label*(i) is used to obtain the actual address of that particular location in the code. During bytecode rewriting it is replaced by a `nop` instruction once the address has been recorded.
- *yield*(i) is replaced by a jump to the end of the code after setting the *runLabel* to i. This way, when *run*() is called again, the switch at the start of the code will jump to the label associated with the current *runLabel*.

- The resulting instrumented bytecode is now compatible with the ProcessJ cooperative scheduler and can now be executed.

Each process has the following code at the beginning of its *run()* method:

```
switch (runLabel) {
    case 1: resume(1); break;
    case 2: resume(2); break;
    ...
    default: // runtime error.
}
```

Along with setting the *runLabel* for *yield*(i), this code implements the core of the cooperative scheduling from a process' point of view, namely, the ability to jump back to where the process last left off (called a *yield*()).

In Section 6 we look more closely at the Java code that gets generated for both channel reads and writes.

4. Communicating Sequential Processes

Communicating Sequential Processes (CSP) [2, 3, 25, 26] is a process algebra enabling concurrent system specification via *processes* and *events*. Processes are abstract components defined by the events they perform.

Events are *atomic*, *synchronous*, and *instantaneous*. That is, events are indivisible, and cause all engaged processes to wait until the event occurs, and when the event occurs it is immediate to all engaged processes. A CSP model defines the events processes are willing to synchronise upon at different stages of their execution.

The simplest CSP processes are STOP—which engages in no events and will not terminate—and SKIP—which engages in no events but will terminate.

Events are added to a process using the *prefix* operator ($\rightarrow$). For example, the process $P = x \rightarrow \mathsf{STOP}$ engages in event x, then stops. A process definition must end by executing another process definition; that is the general form is *Process* = *event* $\rightarrow$ *Process'*. Processes can be recursive (e.g., $P = x \rightarrow P$), with an anonymous form (e.g., $\mu r.x \rightarrow r$) we will use for simplicity in the proof of our shared channels in Section 10.

4.1. Choice

CSP has several *choice* operators to exhibit branching behaviour. The three most frequently used choice types are *external (or deterministic) choice*, *internal (or non-deterministic) choice*, and *prefix choice*.

Given two processes P and Q, the definition $P \,\Box\, Q$ (external choice) is a process that will behave as P or Q based on the first event offered by the environment. 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. The system can (non-deterministically) choose either when both a and b are available.

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

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

$$\Big\Box_{a \in E} a \rightarrow P \equiv e_1 \rightarrow P \,\Box\, e_2 \rightarrow P \,\Box\, \cdots \,\Box\, e_n \rightarrow P$$

$$\Big\sqcap_{a \in E} a \rightarrow P \equiv e_1 \rightarrow P \sqcap e_2 \rightarrow P \sqcap \cdots \sqcap e_n \rightarrow P$$

With prefix choice we can define an event and a parameter. If we define a set of events as $\{c.v \mid v \in Values\}$, 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:

$$c!v \rightarrow P \equiv c.v \rightarrow P$$
$$c?x \rightarrow P \equiv \Box_{x \in Values} c.x \rightarrow P$$

where *Values* is a finite set.

CSP supports a functional *if* statement with each branch ending in a process definition. It is therefore common to write:

$$\begin{array}{l} 1\mathsf{if}\ (b)\ \mathsf{then}\ P \\ \mathsf{else}\ SKIP \end{array}$$

The semantics of $c?v \rightarrow (\mathsf{if}\ (v == x)\ \mathsf{then}\ P\ \mathsf{else}\ Q)$ in CSP is equivalent to $(c.x \rightarrow P) \Box (\Box_{v \in X - \{x\}} c.v \rightarrow Q)$[3].

4.1.1. Pre-Guards

It is possible to control the availability of choice branches (i.e., whether the branch will be considered) by placing a Boolean expression and a & before the choice branch as a pre-guard. 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 expressions e_2 is false only the $c?x$ guard 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 guards be considered in the alternation.

4.2. Process Composition

Processes can be combined via *parallel* and *sequential* composition. A parallel composition is denoted as $P \mid\mid Q$. P and Q must now synchronise on a shared set of events. There are two forms of parallel composition.

- The generalised parallel $P \underset{A}{\mid\mid} Q$ defines two processes with a *synchronisation set* A.
- The alphabetised parallel $P\ {}_A||_B\ Q$ defines two processes that can only perform events in their respective *alphabets*; P with alphabet A and Q with alphabet B.

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

For the generalised parallel, both P and Q must offer any event in A at the same time for that event to occur. Events not in A will not cause synchronise between P and Q, although P and Q may still perform such events. For example,

$$(a \rightarrow b \rightarrow P) \underset{\{b\}}{||} (b \rightarrow c \rightarrow Q)$$

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 as the left-hand side would not offer it.

For the alphabetised parallel, P and Q must synchronise on any events in $A \cap B$. 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*: $P \ ||| \ Q$, which means that P and Q execute in parallel but *do not* synchronise on any 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. As with choice, we can interleave over a set, such as $\underset{a \in \{1 \ldots n\}}{|||} P$ to run n interleaving instances of process P.

Sequential composition is denoted as $P \ ; Q$. This means after P has terminated, Q is performed. Therefore, it allows the definition of behaviour as a sequential composition of discrete process definitions.

4.3. Traces and Hiding

The *trace set* is the set of all a process's externally observed sequences of events . For example, for a process $P = a \rightarrow P$ the shortest trace observable is the empty trace $\langle\rangle$. When P engages on the event a for the first time we can observe this external event and we now have a 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 process events can be concealed via the hiding operator $\backslash$. Hidden events are replaced by τ and are ignored when comparing observable traces. For example, $(a \rightarrow b \rightarrow a \rightarrow \mathsf{SKIP})\backslash\{a\}$ has traces $\{\langle\rangle, \langle b\rangle\}$ as τs are not observable. Similarly, $traces(P \setminus \{a\}) = \{\langle\rangle\}$ and $traces(Q \setminus \{a\}) = \{\langle\rangle, \langle b\rangle, \langle b, b\rangle, \ldots\}$.

4.4. Models

CSP defines three semantic models of behaviour to analyse systems: the *traces* model, the *failures* model, and the *failures/divergence* model. Each model extendeds analysis upon the previous model to establish stronger refinement checks.

4.4.1. The Traces Model

The **traces model** comes from the externally observed behaviour of a system. If the trace set of a system *implementation* Q is a subset of the trace set of a system *specification* P (i.e., $traces(Q) \subseteq traces(P)$), we state that $P \sqsubseteq_T Q$ (Q *trace refines* P). In a refinement test *Specification* $\sqsubseteq_T$ *Implementation*, *Specification* can be thought of as the allowable traces. The refinement test checks if an *Implementation* has only allowable traces.

Hiding events allows us to analyse the external behaviour of a process, thus allowing assertions such as $P \sqsubseteq_T Q \wedge Q \sqsubseteq_T P$. 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.

4.4.2. The Stable Failures Model

The **stable failures model** [25, 26] deals with events that a process may refuse to engage in after a trace. A stable state is one where a process P cannot make internal progress (i.e., via hidden events) and must engage externally. A refusal is an event a process cannot engage with when in a stable state.

The stable failures model overcomes the limitation of comparing traces. For example, $(P = a \rightarrow P) \sqsubseteq_T ((P = a \rightarrow P) \sqcap \mathsf{STOP}))$, although 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 does likewise. More formally, $P \sqsubseteq_F Q \Leftrightarrow failures(Q) \subseteq failures(P)$.

Therefore, a specification in the stable failures model defines which failures an implementation is allowed to have.

4.4.3. 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 and not progress in its externally observed behaviour. For example, consider the following two processes:

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

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 refinement $\sqsubseteq_{FD}$ allows such process comparisons. Formally, we consider both the failures and the divergences of a process P as a pair: $(failures_{\perp}(P), divergences(P))$ and now define refinement in the failures/divergence model as follows: $P \sqsubseteq_{FD} Q \Leftrightarrow failures(Q) \subseteq failures(P) \wedge divergence(Q) \subseteq divergences(P)$. $failures_{\perp}(P)$ is defined as $failures(P) \cup \{(s, X) \mid s \in divergences(P)\}$. That is, the set of traces leading to divergence are added to the existing set of stable failures to create an extended failures set.

Simplified Refinement Checking. 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, and therefore we only need check that our implementation meets the specification in the failures model.

4.5. FDR

FDR [4] is a refinement checker that can check various assertions about CSP processes written in $\mathrm{CSP_M}$ (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 also supports assertions for checking deadlock freedom, divergence freedom, and whether a process definition is deterministic.

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." [27]. In the failures-divergences model, the process must also be divergent free.

4.6. Modelling Global State in CSP

All the ProcessJ runtime classes have member state variables. CSP does not have a global state space, only a global event space. It is therefore necessary to model such state in CSP using processes and events. In this section we present CSP that can be used to model general state that 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:

$$
\begin{aligned}
&\mathsf{datatype}\ Operations = load \mid store \\
&VARIABLE(var, val) = \\
&\quad (var.load!val \rightarrow VARIABLE(var, val)) \\
&\quad \Box \\
&\quad (var.store?val \rightarrow VARIABLE(var, val)) \\
&\alpha VARIABLE(var, T) = \{var.o.v \mid o \leftarrow Operations, v \leftarrow T\}
\end{aligned}
$$

$VARIABLE$ takes two parameters: var is the channel used to communicate the variable with 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 *ready* and a *running* flag (stored as a field in a process), and we can define a channel for communicating these flags as:

$$
\begin{aligned}
&\mathsf{channel}\ ready : Processes.Operations.Bool \\
&\mathsf{channel}\ running : Processes.Operations.Bool
\end{aligned}
$$

5. Scheduling and Processes

Before we consider scheduling, we need to introduce the model of a ProcessJ *process*—or more precisely, the meta data associated with a process as well as the definition of a queue data structure in CSP.

5.1. Processes

A process, in the context of this paper, is code the communicates; that is, either a reader or a writer. We implement these in Section 7.1. For each process the scheduling system requires metadata. Each process requires a monitor for mutual exclusion, a flag for running state, and a flag for the ready state. The fields are represented using the $VARIABLE$ process. We define a monitor process as:

$$
\begin{aligned}
&MONITOR(claim, release) = \\
&\quad claim?pid \rightarrow \\
&\quad release.pid \rightarrow \\
&\quad MONITOR(claim, release)
\end{aligned}
$$

where *claim* and *release* are channels communicating the *identifier* of the process claiming exclusive access to the metadata. *claim* receives the identifier, and will only *release* with the same identifier, ensuring exclusive access. We now define a $PROCESS$ as:

$$
\begin{aligned}
&PROCESS(p) = \\
&\quad\quad VARIABLE(ready.p, true) \\
&\quad ||| \; VARIABLE(running.p, false) \\
&\quad ||| \; MONITOR(claim\_process.p, release\_process.p)
\end{aligned}
$$

Running is initialised to false and ready to true. As we have multiple processes, we use p to identify each process. We define $PROCESSES$ as the interleaving of all $PROCESS$ metadata object-processes:

$$PROCESSES = \underset{p \in Processes}{|||} PROCESS(p)$$

5.2. A Queue Process

The scheduling system uses a queue of processes (the run queue), and we also require queues of processes that are waiting to use a shared channel end. We can implement a simple queue in CSP as follows:

$$
\begin{aligned}
&QUEUE(enqueue, dequeue, size, q, CAP) = \\
&\quad\quad \#q < CAP \;\&\; enqueue?v \rightarrow QUEUE(enqueue, dequeue, q \hat{} < v >, CAP) \\
&\quad \Box \; \#q > 0 \;\&\; dequeue!\mathsf{head}(q) \rightarrow QUEUE(enqueue,\ dequeue,\ \mathsf{tail}(q),\ CAP) \\
&\quad \Box \;\; size!(\#q) \rightarrow QUEUE(push, pop, size, q, CAP)
\end{aligned}
$$

where $\#q$ denotes the length of the sequence q. The queue can *enqueue* processes if it is not full; it can dequeue processes if it is not empty, and it can report its size. CAP is the max size of the queue (this is needed to enable for FDR to verify the system—FDR cannot handle an unbounded queue). In an actual implementation we do not define a max capacity. We can now completely describe the scheduling system in the next section.

5.3. Scheduling

In [7] we fixed the problem with the divergence caused by not-ready processes in the run queue [6]. We introduced a schedule manager to control the insertion of ready processes into the run queue.

$SCHEDULER =$
$rqdequeue?p \rightarrow$ - 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

$$SCHEDULERS(N) = (((|||_{n \in \{1..N\}} SCHEDULER) \;|||\; SCHEDULE\_MANAGER) \;||_{\alpha RUNQUEUE}\; QUEUE(rqenque, rqdequeue, rqsize, <>, \#Processes)) \setminus \alpha RUNQUEUE$$

In the original version of the scheduler, scheduling a process only required setting the *ready* field to true. With the new scheduler, a *running* field is introduced, requiring more considered scheduling operations. To schedule a process use the following procedure:

$SCHEDULE(me, pid) =$
$claim\_process.pid.me \rightarrow$ - lock the process object
$ready.pid.load?r \rightarrow$ - get the *ready* field
if (r) then - **if** (the process is ready) **then**

```
        release_process.pid.me →        - release the lock
        SKIP                            - and do nothing
    else                                - else
        ready.pid.store!true →          - set the process ready
        running.pid.load?r2 →           - get the running status
        if (r2) then                    - if (the proc. is running) then
            release_process.pid.me →    -   release the lock
            SKIP                        -   and do nothing
        else                            - else
            schedule.pid →              -   schedule the process
            release_process.pid.me →    -   release the lock
            SKIP                        -   and do nothing
```

Processes are descheduled when they yield.

```
YIELD(pid) =
    DESCHEDULE(pid);                    - deschedule the process
    yield.pid →                         - yield to the scheduler
    run.pid →                           - wait to be scheduled
    running.pid.store!true → SKIP       - set running to true
```

Processes deschedule themselves, yield to the scheduler, and then wait for a scheduler to run them again. At this point, they set their running flag to true and continue from the point they yielded. Descheduling is defined as:

```
DESCHEDULE(pid) =
    claim_process.pid.pid →             - lock the process metadata
    running.pid.store.false →           - set running to false
    ready.pid.load?r →                  - check if still ready to run
    if (r) then                         - if (proc. ready to run) then
        schedule.pid →                  -   schedule the process
        release_process.pid.pid →       -   unlock the metadata
        SKIP
    else                                - else
        release_process.pid.pid →       -   unlock the metadata
        SKIP
```

The process metadata is locked, and the running flag set to false. A check is made on the ready flag, and if true it is scheduled (added to the run queue). Otherwise, nothing happens.

With all these processes in place, we can define a stand-alone scheduling process that works for any number of schedulers:

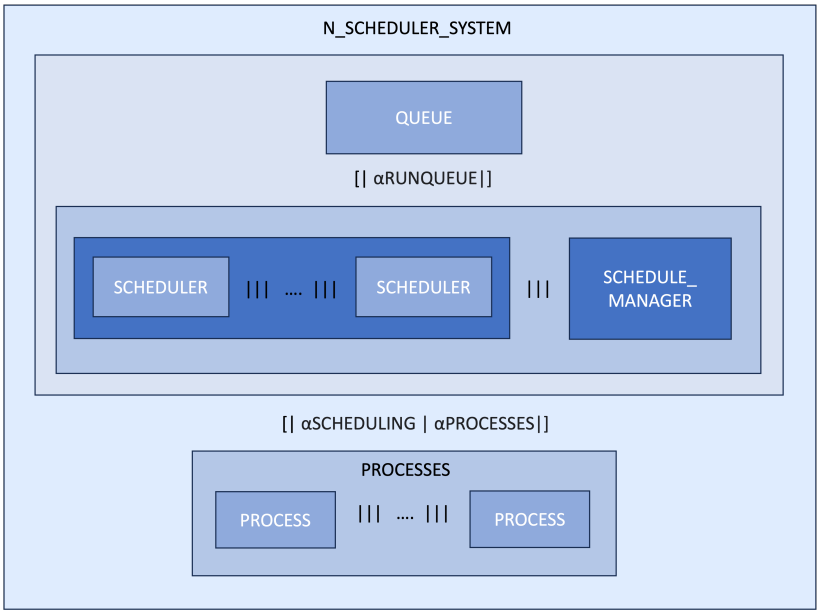

Figure 1: The $N\_SCHEDULER\_SYSTEM$ network.

$$N\_SCHEDULER\_SYSTEM(N) = \\ SCHEDULERS(N) \ {}_{\alpha SCHEDULING}||_{\alpha PROCESSES} \ PROCESSES$$

Figure 1 shows the $N\_SCHEDULER\_SYSTEM$ process.

We now have the underlying scheduling system in place, and in the next section we present a the implementation of a shared channel in ProcessJ and its translation to CSP.

6. Shared Channels in ProcessJ

In Section 8 we present the CSP for the **specifications** of the behaviour of a shared channel. In this section we will develop the CSP for the *implementation* of the shared channel. This includes a number of steps to produce a CSP model starting with ProcessJ code and considering the generated Java code:

1. We need to consider the translation of both a channel *read* expression and a channel *write* statement in ProcessJ to Java.
2. We need to translate this generated Java code into CSP.
3. We need CSP code for the ProcessJ runtime implementation of the shared channel (written in Java), this includes functionality to claim and unclaim channel ends amongst other things.

In the following section we start with an example of two ProcessJ programs that use shared channels; one with a shared reading end and one with a shared writing end.

6.1. ProcessJ Example of Shared Ends

Figure 2 shows two different ProcessJ programs. The left side illustrates the use of a channel with a **shared writing** end: we run two instances of f that share the writing end of a channels c concurrently with a single reader. The reader reads from the channel twice and prints out either `42 43` or `43 42` depending on which writer uses the channel first. The right side illustrates the use of a channel with a **shared reading** end: we run one writer (that writes the values `42` and then `43` on the channel) along with two readers. Each reader will read from the shared channel once and print their values. Depending on the scheduling, the result is similarly either `42 43` or `43 42`.

Channels can also have both shared reading and writing ends. A shared channel is **not** a CSP multi-way synchronisation, which requires all involved processes to engage on the same event. With shared channels, only one reader and one writer are engaged per channel communication.

In order to ensure that only one writer/reader will be accessing the channel, we require a mutual exclusion mechanism. In ProcessJ, a claim value and process queue is used. Next, we consider how to translate the processJ communication primitives into Java.

6.2. Translating write() to Java on a Shared Channel Write End

As the first step we consider the translation of *write* in ProcessJ to Java. A ProcessJ *write* statement on a *non-shared* channel end is:

chan$< \cdots >$ c;
$\vdots$
c.**write**(val)

This code translates to the following Java code:

$\overline{c}$.*write*(**this**, *val*);
this.*runLabel* = 2; // representing L_2
yield();
L_2:

Shared Writing End	Shared Reading End

void *f*(**shared chan**<**int**>.**write** *out*,
int *val*) {
out.**write**(*val*);
}
 	
void *g*(**chan**<**int**>.**read** *in*){
int *x*;
x = *in*.**read**();
println(x);
x = *in*.**read**();
println(x);
}
 	
void*main*(**string** *args*[ ]) {
shared write chan<**int**> *c*;
par {
f(*c*.**write**, 42);
f(*c*.**write**, 43);
g(*c*.**read**);
}
}
	void *f*(**chan**<**int**>.**write** *out*, **int** *val*) {
	out.**write**(*val*);
	out.**write**(*val+1*);
	}
	
	void *g*(**shared chan**<**int**>.**read** *in*) {
	int *x*;
	x = *in*.**read**();
	println(*x*);
	}
	
	void *main*(**string** *args*[ ]) {
	shared read chan<**int**> *c*;
	par {
	f(*c*.**write**, 42);
	g(*c*.**read**);
	g(*c*.**read**);
	}
	}

Figure 2: Example ProcessJ shared reading (one-to-many) and shared writing (many-to-one) channels.

where $\overline{c}$ is a Java object reference to a channel. If a channel end is shared, access to it must be protected. The simplest approach is to claim the channel before the code above gets executed and release it again after. In order to avoid busy waiting for processes that fail the claim, we equip a shared channel end with a queue of processes waiting to use the channel end. Therefore, if a process fails to claim, it gets placed at the end of the queue. It it worth noting that a channel with shared ends is no different in its basic construction than a non-shared channel except access to it is controlled with a claim value and a queue of processes awaiting access. If we repeat the ProcessJ code from above with a channel where the writing end is shared we have:

shared write chan$< \cdots >$ *c*;
⋮
c.write(*val*);

The generated Java code with a claim looks like this (code in outlined boxes comes from the non-shared version):

if (!$\overline{c}$.*claimWrite*(**this**)) {
  this.*runLabel* = 1; // representing L_1
  yield();
}
L_1:

$\overline{c}$.*write*(**this**, *val*);
this.*runLabel* = 2; // representing L_2
yield();
L_2:

$\overline{c}$.*unclaimWrite*();

where $\overline{c}$ is a Java reference to a channel object, and **this** is a reference to the process object (*PJProcess*). Before the write can commence on the shared channel end, the process must first make a claim on the writing end of the shared channel. This is done in the synchronized *claimWrite*() method. The claim succeeds if no other process holds a claim or if the process already has a claim on the channel end. If a different process already claimed the channel end, the process will be placed in a queue, set not-ready to run

and *false* will be returned. If the claim fails, the process yields, and when re-awoken (continuing at L_1), the write can proceed. The process was set ready to run by a different process that held the lock on the channel end. This happens in *unclaimWrite*(). After this, the process again yields in order for the communication to remain synchronous (the writing process gets re-started by the reader eventually) and continues at L_2, after which the channel end is unclaimed and ready to be claimed by another writer. When a process unclaims a channel end, the queue of waiting processes is popped (if not empty) and the removed process is set ready to run and registered as holding the claim to the channel end.

The channel end claims on a shared channel uses the field *writerclaim* for the shared write channels, and *readclaim* for the shared read channels. This field is set and read in synchronized methods only. Both the *claimWrite* and *unclaimWrite* are synchronous methods. This means that a different lock is taken out on the entire channel object while these two methods run. We will model this in the implementation as well. Let us look at the *PJMany2OneChannel* class:

```
public class PJMany2OneChannel<T> extends PJOne2OneChannel<T> {
    protected PJProcess writeclaim = null;

    protected Queue<PJProcess> writeQueue = new LinkedList<>();

    public synchronized boolean claimWrite(PJProcess p) {
        if (writeclaim == null || writeclaim == p) {
            writeclaim = p; // this channel's write is claimed by p.
            return true;
        }else {
            p.setNotReady(); // set p not ready to run.
            writeQueue.add(p); // add p to the queue of waiting writers.
        }
        return false;
    }

    public synchronized void unclaimWrite() {
        if (writeQueue.isEmpty()) {
            writeclaim = null; // no one wants the channel's writing end right now.
        } else {
            PJProcess p = writeQueue.remove(); // get the next waiting writer.
            writeclaim = p; // this channel's write end is claimed by p.
            schedule(p); // schedule p.
        }
    }
}
```

The field *writeclaim* holds a reference to the process that currently holds the lock on the channel or `null` when no claim exists. The *writeQueue* holds a list of other writers that wish to write to the channel.

The two methods *claimWrite*() and *unclaimWrite*() are used for claiming and unclaiming a channel. Let us consider them in greater detail in turn. A call to *claimWrite*() with a process object reference checks if the channel is currently claimed or otherwise. If true, *writeclaim* is updated to the process' reference and *true* is returned. If another process has a claim of the channel, the waiting process is set not-ready and placed at the back of the queue and *false* is returned.

unclaimWrite() will set the *writeclaim* field back to *null* if no other processes are waiting in the queue. If one is, the first will be removed from the queue and its reference is placed in *writeclaim*, **and** that process will be set ready. It should be clear now why the second check ($writeclaim == p$) is necessary in the *claimWrite*() method.

To model this in CSP we need:

- A variable *writeclaim* holding a process ID.
- A queue of process IDs.
- The code for *claimWrite*() and *unclaimWrite*()

in addition, we need the state and methods from the *PJOne2OneChannel* class, which are as follows:

- The *reader* and *writer* variables representing the reader and the writer of the channel.
- Code for *isReadyToRead*(), *read*() and *write*().

but also the data from the *PJChannel* class:

- The actual *data* on the channel.

For completeness, let us just list the code for *PJOne2OneChannel* and *PJChannel* here:

public *PJOne2OneChannel*<*T*> **extends** *PJChannel*<*T*> {
    PJProcess reader = *null*;
    PJProcess writer = *null*;

```
    boolean isReadyToRead(PJProcess p) {
        if (writer != null) {              // if (a writer is present) then
            return true;                   //     return true
        } else {                           // else
            reader = p;                    //     register p as reader
            reader.setNotReady();          //     set reader not ready
            return false;                  //     return false
        }
    }

    void write(PJProcess p, T item) {
        data = item;                       // set data on channel
        writer = p;                        // register the writer
        writer.setNotReady();              // set writer not ready
        if (reader != null)                // if (a reader is there) then
            schedule(this, p);             //     schedule it
    }

    T read(PJProcess p) {
        schedule(this, writer);            // schedule the writer
        writer = null;                     // clear writer field
        reader = null;                     // clear reader field
        return data;                       // return the data
    }
    ⋮
}
```

and the parts of *PJChannel* that are important:

```
public class PJChannel<T> {
    protected T data;
    ...
}
```

The CSP code for *read*() and *write*() was previously developed and used in [6]. Since we are considering a shared-write (many-to-one) channel, the read expression (*c*.**read**()) will be exactly as it was in [6], namely:

```
if (!c.isReadyToRead()) {
```

```
    this.runLabel = 1; // representing L1
    yield();
}
L1:
var = c̄.read();
this.runLabel = 2; // representing L2
yield();
L2:
```

Again, the label numbers 1 and 2 are arbitrarily chosen here. Before we develop the CSP, let us first consider shared read channels in the next section.

6.3. Translating read() to Java on a Shared Channel Read End

Like before, let us start with the read on a *non-shared* channel read end:

```
chan< ··· > c;
⋮
var = c.read()
```

We saw the code that this expression generates for a non-shared channel end at the end of the previous section. The approach is very similar to the shared writing end (where the channel is defined as **shared read chan**$< \cdots >$): we claim a lock before and release after, and we maintain a queue of readers that failed to obtain the lock[4]. This generated Java code looks like this:

```
if (!c̄.claimRead(this)) {
    this.runLabel = 1; // representing L1
    yield();
}
L1;
```

[4]The actual code for the *claimRead*() and *unclaimRead*() can be found in the GitHub repository along with all the CSP scripts used in this paper.

```
if (!c̄.isReadyToRead(this)) {
  this.runLabel = 2; // representing L2
  yield();
}
L2;

x = c̄.read(this);
this.runLabel = 3; // representing L3
```

$\bar{c}.unclaimRead()$;

```
yield();
L3;
```

Note, the placement of the *unclaimRead*() before the *yield*(). The yield is a courtesy yield[5] only and could be removed, so it is OK to unclaim the channel end before performing the yield.

The *PJOne2ManyChannel* class is very similar to the *PJMany2OneChannel* that we showed earlier; rather than *claimWrite*() and *unclaimWrite*() it has *claimRead*() and *unclaimRead*(). Rather than a *writeQueue* it has a *readQueue*, and instead of the *writeclaim* field it has a *readclaim* field. The implementation of the claim and unclaim method are exactly the same as for the many-to-one channel.

We know that we can reuse the *read*() and *write*() code from [6] as well as code generated for the writes to non-shared ends as reads from non-shared ends. For shared ends, we need to book-end the code it with the claiming and unclaiming of the channel end as just explained and illustrated. In the next section we present the needed CSP for the new variables/fields and the required queue.

7. CSP Model of the ProcessJ shared channels

The next step is to translate the generated Java code into CSP and then perform the verification against the specifications from Section 8. However, we start with the ProcessJ channel implementation in CSP.

[5]We do this to create some kind of fairness.

7.1. Channels in CSP

As we are implementing channels with shared ends (reading and writing), we define a *Processes* type based on the tests being undertaken. For example, for a one-to-many channel with one writer sending to two readers we have:

- all processes used by the model
datatype *Nullable_Processes* = *NULL* | *R1* | *R2* | *W*
- the processes that can be scheduled
subtype *Processes* = *R1* | *R2* | *W*
- the processes reading from the channel
subtype *Reading_Processes* = *R1* | *R2*
- the processes writing to the channel
subtype *Writing_Processes* = *W*
- the reading processes with null
subtype *Nullable_Reading_Processes* = *NULL* | *R1* | *R2*
- the writing processes with null
subtype *Nullable_Writing_Processes* = *NULL* | *W*

A channel communication in ProcessJ is not instantaneous as was described in Section 6.1. Both the writer and the reader on a channel start their respective interaction, wait for synchronisation, and then complete. We consider these two operations (reading and writing) to have a starting event and an ending event. Due to the reader and writer effectively interleaving except at the channel synchronisation point, their are four possible interactions that can occur on a channel as presented in Figure 3. We therefore must define these four events as channels like so:

channel *read, write*:*Channels.Processes.Values*
channel *start_read*, *ack*: *Channels.Processes*

A *read* and *write* communicate on a *channel*. We identify the *process* performing the action, and need to know the *value* communicated. We therefore must define the types $Channels$ and $Values$. Our system only has one channel. Two values are sufficient to demonstrate the correct message transmission of a channel.

datatype *Channels* = *C1*
datatype *Values* = *A* | *B*

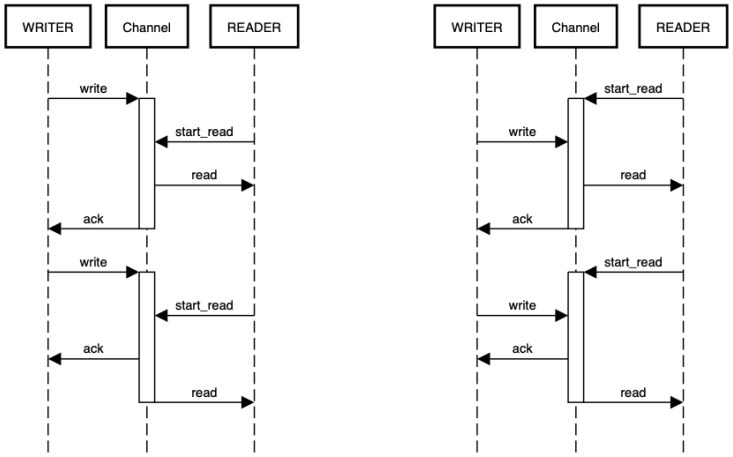

Figure 3: Four interactions of channels.

In Section 5 we introduced the *VARIABLE* and the *MONITOR* processes. These are needed in our channel implementation as well. From [6] we can almost reuse the *CHANNEL* process; all we need extra is a monitor. A channel has a single monitor as well as three fields: a reference to the reader, a reference to the writer, and the data being sent; we can use three *VARIABLE* processes to implement these fields:

channel *writer, reader : Channels.Operations.Nullable_Processes*
channel *data : Channels.Operations.Values*
channel *channel_claim, channel_release :Channels.Processes*

$$
\begin{aligned}
&CHANNEL(chan) = \\
&\quad VARIABLE(writer.chan, NULL) \\
&\quad ||| \ VARIABLE(reader.chan, NULL) \\
&\quad ||| \ VARIABLE(data.chan, A) \\
&\quad ||| \ MONITOR(channel\_claim.chan, channel\_release.chan)
\end{aligned}
$$

We also reuse the code for *read*() and *write*():

$$
\begin{aligned}
&READ(chan) = \\
&\quad writer.chan.load?w \rightarrow && \text{- get } writer. \\
&\quad \textsf{if } (w == NULL) \textsf{ then} \\
&\quad\quad \textsf{DIV} && \text{- this never happens} \\
&\quad \textsf{else} \\
&\quad\quad ready.w.store!true \rightarrow && \text{- } writer.setReady();
\end{aligned}
$$

```
    writer.chan.store!NULL →          - writer = null;
    reader.chan.store!NULL →          - reader = null;
    SKIP

WRITE(pid, chan, item) =
    data.chan.store!item →            - data = item;
    writer.chan.store!pid →           - writer = pid;
    ready.pid.store!false →           - writer.setNotReady();
    reader.chan.load?v →              - get reader
    if (v! = NULL) then               - if (reader != null) then
        ready.v.store!true → SKIP     -     schedule(this, reader);
    else
        SKIP
```

To interact with the *MONITOR* and *QUEUE* processes we define the following channels:

```
channel write_end_enqueue, write_end_dequeue : Channels.Writing_Processes
channel read_end_enqueue, read_end_dequeue : Channels.Reading_Processes
channel write_queue_size : Channels.{0..(card(Writing_Processes) - 1)}
channel read_queue_size : Channels.{0..(card(Reading_Processes) - 1)}
channel readclaim : Channels.Operations.Nullable_Reading_Processes
channel writeclaim : Channels.Operations.Nullable_Writing_Processes
```

**_end_enqueue* and **_end_dequeue* are used to add or remove processes to the wait queue on a shared channel end. The **_queue_size* channels allow getting the current size of the wait queues. Finally, *readclaim* and *writeclaim* allow getting and setting the current process claiming the shared end. We create a many-to-one channel process by interleaving a regular channel, a queue of writers, and a variable for the field *writeclaim*:

```
MANY_TO_ONE_CHANNEL(c) =
        CHANNEL(c)
    ||| QUEUE(end_enqueue.c.write_end, end_dequeue.c.write_end,
                queue_size.c.write_end, < >)
    ||| VARIABLE(writeclaim.c, NULL)
```

Figure 4 illustrates the *MANY_TO_ONE* channel.

For the *ONE_TO_MANY_CHANNEL* process, the first two parameters of the *QUEUE* process simply changes to *end_enqueue.c.read_end, end_dequeue.c.read_end*, and the *VARIABLE*'s first parameter is now the

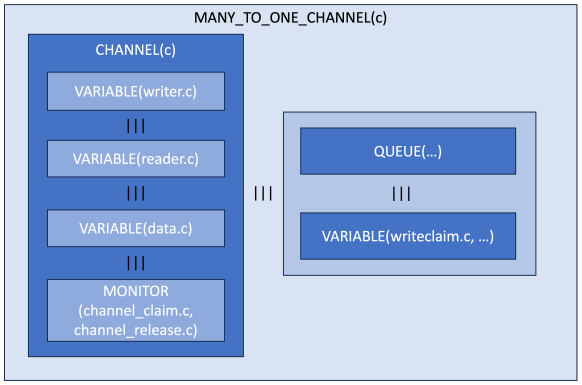

Figure 4: The *MANY_TO_ONE_CHANNEL* process.

readclaim channel instead. For the *MANY_TO_MANY_CHANNEL* process, we need two queues (one for readers and one for writers) and two variables (one for the *readclaim* and one for the *writeclaim* fields). The complete code is available on the GitHub page listed for this paper in the introduction.

7.2. Claiming and Unclaiming a Channel

We now implement the *CLAIM_WRITE* and *UNCLAIM_WRITE* as follows:

```
CLAIM_WRITE(pid, chan) =
    channel_claim.chan!pid →                - claim the channel
    writeclaim.chan.load?wc →               - read the writeclaim field
    if (wc == NULL or wc == pid) then       - if (writeclaim == null ||
                                                  writeclaim == p) then
        writeclaim.chan.store!pid →         -   writeclaim = p;
        channel_release.chan.pid →          -   release the channel
        SKIP
    else                                    - else
        ready.pid.store!false →             -   p.setNotReady();
        write_end_enqueue.chan.pid →        -   writeQueue.add(p);
        channel_release.chan.pid →          -   release the channel
        YIELD(pid)                          -   yield();

UNCLAIM_WRITE(pid, chan) =
    channel_claim.chan!pid →                - claim the channel
```

$queue\_size.chan.write\_end?s \rightarrow$ - $s = writeQueue.size()$;
if $(s == 0)$ then - **if** $(s == 0)$ **then**
  $writeclaim.chan.store!NULL \rightarrow$ - $writeclaim = null$;
  $channel\_release.chan.pid \rightarrow$ - release the channel
  SKIP
else - **else**
  $write\_end\_dequeue.chan?p \rightarrow$ - $p = writeQueue.pop()$;
  $writeclaim.chan.store!p \rightarrow$ - $writeclaim = p$;
  $SCHEDULE(pid, p)$; - $schedule(p)$;
  $channel\_release.chan.pid \rightarrow$ - release the channel
  SKIP

SCHEDULE schedules the process by adding it to the run queue and setting its *ready* field to *true* if false. The definitions for *CLAIM_READ* and *UNCLAIM_READ* are similar as expected.

We also define the CSP for the ProcessJ channel-write statement and channel-read expression; we call these *RESTRICTED_PROCESS_WRITER* for a non-shared writing end, *RESTRICTED_PROCESS_SHARED_WRITER* for a share writing end, *RESTRICTED_PROCESS_READER* for a non-shared reading end, and *RESTRICTED_PROCESS_SHARED_READER* for a shared reading end. Let us start with the *RESTRICTED_PROCESS_WRITER* and the *RESTRICTED_PROCESS_READER* from [7] is shown below. We indicate where the claims and unclaims will be placed.

$RESTRICTED\_PROCESS\_WRITER(pid, chan) =$
  $schedule!pid \rightarrow$ - schedule the process
  $run.pid \rightarrow$ - wait to be run
  $running.pid.store!\text{true} \rightarrow$ - **this**.$running$ = **true**;
  $RESTRICTED\_PROCESS\_WRITER'(pid, chan)$

  $RESTRICTED\_PROCESS\_WRITER'(pid, chan) =$
  $write.chan.pid?message \rightarrow$ - take message from the env.
  -- place for future *CLAIM_WRITE*
  $channel\_claim.chan!pid \rightarrow$ - claim the channel
  $WRITE(pid, chan, mesage)$; - perform the write
  $channel\_release.chan.pid \rightarrow$ - release the channel
  $YIELD(pid)$; - $yield()$
  -- place for future *UNCLAIM_WRITE*

```
    ack.chan.pid →                          - acknowledge to the env.
    RESTRICTED_PROCESS_WRITER'(pid, chan)

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

RESTRICTED_PROCESS_READER'(pid, chan) =
    start_read.chan.pid →                   - external event to start
    -- place for future CLAIM_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 channel
            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 channel
    (
        YIELD(pid);                         - yield();
        -- place for future UNCLAIM_READ
        read.chan.pid!message →             - deliver msg to the env.
        RESTRICTED_PROCESS_READER'(pid, chan)
    )
```

To produce the shared versions of these procedures, all we have to do is insert *CLAIM_WRITE/CLAIM_READ* and *UNCLAIM_WRITE/UNCLAIM_READ* as indicated above.

7.3. Building Shared Channels

Now we can follow much the same procedure as we did when we created the specifications: for the many-to-one channel, interleave a number of writers and a single reader and run these in parallel with a single many-to-one channel. This can be done as follows:

$$
\begin{aligned}
&RESTRICTED\_SHARED\_WRITER(writers,\ R,\ C) = \\
&\quad (|||_{p \in writers}\ RESTRICTED\_PROCESS\_SHARED\_WRITER(p,\ C)) \\
&\quad ||| \\
&\quad RESTRICTED\_PROCESS\_READER(R,\ C)
\end{aligned}
$$

$$
\begin{aligned}
&RESTRICTED\_READER\_SHARED\_WRITER(writers,\ R,\ C) = \\
&\quad (READER\_SHARED\_WRITER(writers,\ R,\ C) \\
&\qquad ||_{\alpha CHANNELS} \\
&\quad MANY\_TO\_ONE\_CHANNEL(C))\ \ \backslash \alpha CHANNELS
\end{aligned}
$$

If we were considering a classical unrestricted (unscheduled) system, *RESTRICTED_READER_SHARED_WRITER* would be the implementation we would utilize in a verification scenario (though without the scheduling events), but since we are indeed taking into account the execution environment, we need to attach a scheduling system as well. We do this by simply running in parallel *RESTRICTED_READER_SHARED_WRITER* and *N_SCHEDULER_SYSTEM*.

$$
\begin{aligned}
&RESTRICTED\_PJ\_MANY\_TO\_ONE\_CHAN\_SYSTEM(writers,\ R,\ C,\ N) = \\
&\ (RESTRICTED\_READER\_SHARED\_WRITER(writers,\ R,\ C) \\
&\qquad ||_{\alpha N\_SCHEDULER\_SYSTEM} \\
&\ N\_SCHEDULER\_SYSTEM(N))\ \ \backslash \alpha N\_SCHEDULER\_SYSTEM
\end{aligned}
$$

where n is the number of schedulers. The equivalent one-to-many and many-to-many systems are built in a similar manner. The actual code can be found in the GitHub repository. Figure 5 shows the process network of the *RESTRICTED_PJ_MANY_TO_ONE_CHAN_SYSTEM* process.

8. A Generic Shared Channel Specification

In CSP, a shared channel occurs when two or more processes interleave while they all are willing to communicate on a channel to another process. A simple example is:

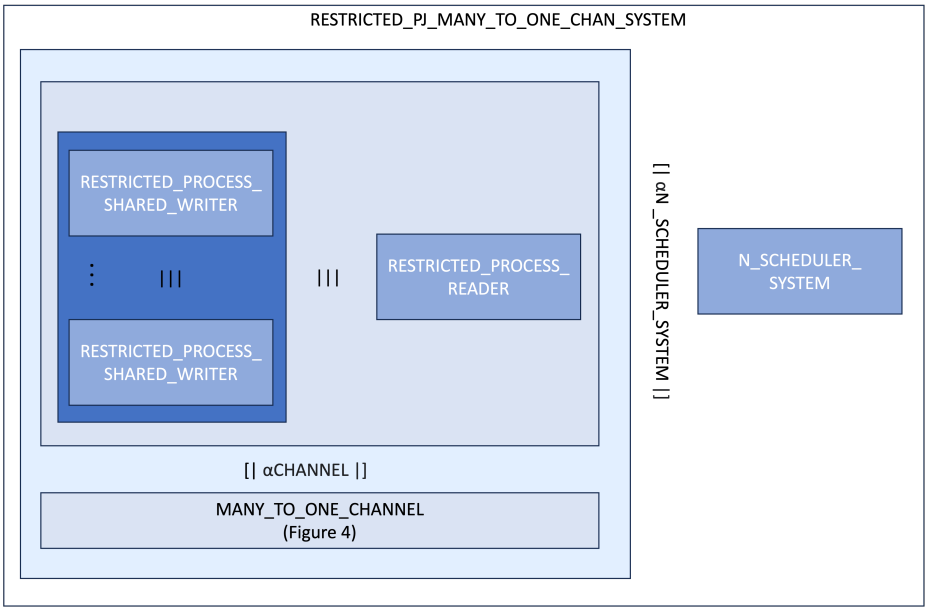

Figure 5: The *RESTRICTED_PJ_MANY_TO_ONE_CHAN_SYSTEM* process network.

$$P = a!x \rightarrow P$$
$$Q = a!y \rightarrow Q$$
$$R = a?z \rightarrow R$$
$$SYSTEM = (P \;|||\; Q) \;\underset{\{a.v \;|\; v \in Values\}}{||}\; R$$

The channel a synchronises P and R or Q and R, but the process pair P and Q will not synchronise on channel a. In CSP, the behaviour is equivalent to a nondeterministic choice between P and Q. For modelling, we must provide a specification that matches this behaviour.

In [6] we presented a specification of a non-shared channel which originally was used by Welch and Martin in [20] to prove correctness of the channels from the JCSP Java package. They introduced a *LEFT* and a *RIGHT* process; the *LEFT* process representing the writer and the *RIGHT* process representing the reader. Our goal is to model the four possible channel interactions presented in Figure 3.

channel $transmit$: $Channels.Values$

$$LEFT(pid, chan) =$$

$write.chan.pid?mess \rightarrow$ - take a message
$transmit.chan!mess \rightarrow$ - send it
$ack.chan.pid \rightarrow$ - acknowledge the send
$LEFT(pid, chan)$ - recurse

$RIGHT(pid, chan) =$
$start\_read.chan.pid \rightarrow$ - Start the read
$transmit.chan?mess \rightarrow$ - received the message
$read.chan.pid!mess \rightarrow$ - deliver the message
$RIGHT(pid, chan)$ - recurse

and a generic channel specification using these *LEFT* and *RIGHT* processes:

$$GENERIC\_CHANNEL(W, R, C) =$$
$$(LEFT(W, C) \ {}_{\alpha LEFT}||_{\alpha RIGHT} \ RIGHT(R, C)) \setminus \{| \ transmit \ |\}$$

Figure 6 shows this process network. $\alpha LEFT$ and $\alpha RIGHT$ are the *alphabets* on which the two processes engage; Their formal definitions are as follows:

$$\alpha LEFT(pid, chan) =$$
$$\{write.chan.pid.v, transmit.chan.v, ack.chan.pid \mid v \leftarrow Values\}$$
$$\alpha RIGHT(pid, chan) =$$
$$\{start\_read.chan.pid, transmit.chan.v, read.chan.pid.v \mid v \leftarrow Values\}$$

The *transmit* event is hidden[6] and the *LEFT* process engages with the environment on *write* and *ack* (the former representing the start of a *write* call on a channel and the latter representing the completion, and the *RIGHT* process on the events *start_read* and *read* (the former representing that start of a *read* call on a channel and the latter representing the completion and return of the read value).

Recall, ProcessJ has the following three different types of shared channels:

- A many-to-one channel (multiple writers to one reader).
- A one-to-many channel (one writer to multiple readers).

[6]We hide the complete set of events for *transmit*, denoted by the set $\{|transmit|\}$.

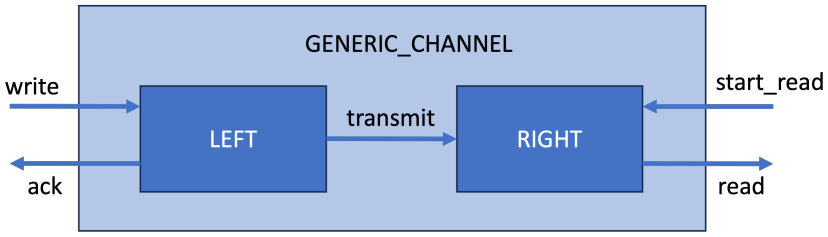

Figure 6: The composition of a *GENERIC_CHANNEL with a LEFT writer and a RIGHT reader.*

- A many-to-many channel (multiple writers to multiple readers).

We can specify all of these using the *LEFT* and the *RIGHT* processes described above. Let us start with the more general of the three, namely, the many-to-many (multiple writers and multiple readers); we call this one *N_TO_M_GENERIC_CHANNEL* and it looks like this:

$$
\begin{aligned}
&N\_TO\_M\_GENERIC\_CHANNEL(writers, readers, C) = \\
&\quad ((\quad |||_{p \in writers} \quad LEFT(p, C)) \\
&\qquad \Big|\Big|_{\{|transmit|\}} \\
&\quad (\quad |||_{p \in readers} \quad RIGHT(p, C))) \setminus \{|transmit|\}
\end{aligned}
$$

where *writers* is a list of n process identifiers for the writing processes (i.e., the *LEFT* processes), and *readers* is a list of m process identifiers for the reading processes (i.e., the *RIGHT* processes). The n writers and m readers all use the same *transmit* channel (the actual communication channel between left and right sides), and since all the readers are interleaved and all the writers are interleaved, only one writer and one reader will be paired up and communicate using the *transmit* channel. Thus, a single communication happens across the *transmit* channel, which is finally hidden from the environment. Figure 7 illustrates the *N_TO_M_GENERIC_CHANNEL* generic channel network. Note, each *write/ack* and each *start_read/read* pair of channels belong to different processes. Channel communication between a reader-writer pair still behaves as illustrated in Figure 3.

The one-to-many and the many-to-one channels can easily be specified using the *N_TO_M_GENERIC_CHANNEL*. Simply pass a list with a single writer to get a one-to-many channel, and to get a many-to-one channel, pass a list with a single reader.

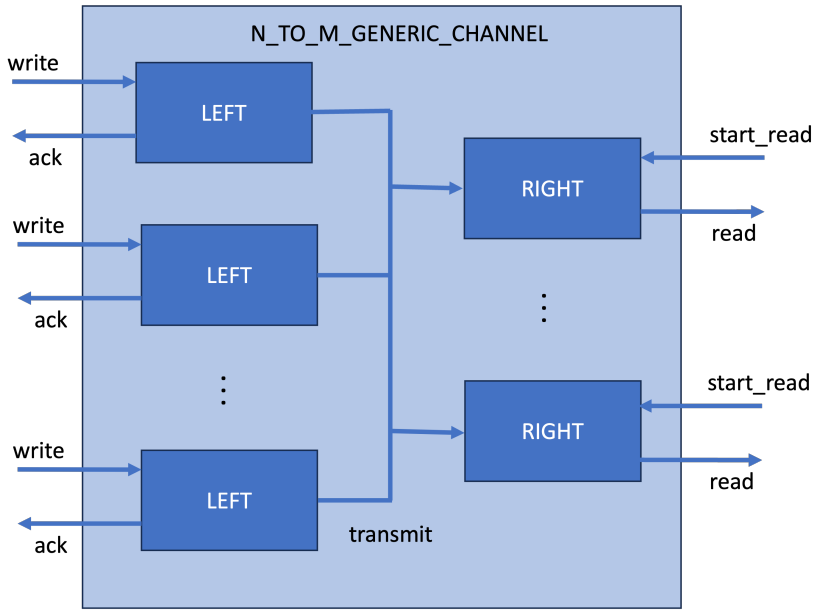

Figure 7: *N_TO_M_GENERIC_CHANNEL network.*

In the next section, we follow the procedure of Welch and Martin [20] by proving that our shared channel specification indeed behaves like a shared channel in CSP. Having accomplished this, we know that it is safe to use our specification of a shared channel in the subsequent CSP code.

9. Results

With both the CSP for the *specification* of a shared channel (Section 8) and the *implementation* (Section 7), we can now perform the actual verification using FDR.

We start by checking for divergence and deadlock freedom in both the specification and the implementation (with 1, 2, 3, and 4 schedulers); indeed both the specification and the implementation are divergence and deadlock free. Checking both the specification and the implementation for determinism fails—as it should. This means that we can limit our refinement checks to the *stable failures* model.

Table 1 shows the results of verification of the three different ProcessJ shared channel types.

Note, for the one-to-many we have one writer, and two or three readers—a total of three or four processes. Similarly, the many-to-one has two or three

Table 1: Results for the one-to-many channel. (F = Failures, T = Traces)

Spec $\sqsubseteq$ *Impl*	# Active Procs.	Number of schedulers **1**	**2**	**3**	**4**
$writers - readers$	One-to-many				
1-2	3	T	T	F	F
1-3	4	T	T	T	F
$writers - readers$	Many-to-one				
2-1	3	T	T	F	F
3-1	4	T	T	T	F
$writers - readers$	Many-to-many				
2-2	4	T	T	T	F

Spec $\sqsubseteq$ *Impl*	# Active Procs.	Number of schedulers **1**	**2**	**3**	**4**
$writers - readers$	One-to-many				
1-2	3	✗	✗	F	F
1-3	4	✗	✗	✗	F
$writers - readers$	Many-to-one				
2-1	3	✗	✗	F	F
3-1	4	✗	✗	✗	F
$writers - readers$	Many-to-many				
2-2	4	✗	✗	✗	F

writers and a single reader; again, three or four processes.

We see that when the number of schedulers is three for these two channel types, the results are positive for one writer and two readers as well as for two writers and one reader in the stable failures model in both directions. When it comes to three writers and a reader or one writer and three readers, the required number of schedulers to achieve refinement in the failures model is four.

For the many-to-many channel with two writers and two readers—a total of four processes, refinement in the failures model is not achieved until the number of schedulers equals at least four. In general, with n writers and m readers, a total of $n + m$ schedulers will be needed to obtain refinement in both directions in the stable failures model.

With the results from this paper, we now have verified the scheduled implementations of the processJ runtime for the following entities:

- One-to-one shared channels [6].
- Choice (alts) for up to 3 writers [7].
- One-to-many, many-to-one, and many-to-many channels [this paper].

In Section 11, we will look closer at the results that did not pass the refinement check, and explain why.

10. Formal proof of shared channel spec behaving like a shared CSP channel

We begin as Welch and Martin by considering ALT-free CSP programs. We will start with one that uses a many-to-one channel with no alternation. We define a $SCSP$ network as a special kind of CSP network $P_1 \,||\, P_2 \,||\, \ldots \,||\, P_n$ where P_i is defined as a $SCSPPROC$:

$$\begin{aligned} SCSPPROC = {} & SKIP \\ & \mid a!x \rightarrow SCSPPROC \\ & \mid a?x \rightarrow SCSPPROC \\ & \mid SCSPPROC \,\Box\, SCSPPROC \\ & \mid SCSPPROC \,\sqcap\, SCSPPROC \\ & \mid SCSPPROC \,|||\, SCSPPROC \end{aligned}$$

We have added $SCSPPROC \;|||\; SCSPPROC$ to Welch and Martin's original proof. For a many-to-one channel with two writers, we have the following specification:

$$MANY\_TO\_ONE\_CHANNEL(a) = \\ (((LEFT(a) \;|||\; LEFT(a)) \;||\; RIGHT(a))) \setminus \{a\}$$

where

$$LEFT(a) = write.a?x \rightarrow a!x \rightarrow ack.a \rightarrow LEFT(a) \\ RIGHT(a) = start\_read.a \rightarrow a?z \rightarrow read.a!z \rightarrow RIGHT(a)$$

We only need to prove the case for two *LEFT* processes as the proof simply expands for more *LEFT* processes as we will show. We now need to demonstrate that this channel does indeed behave as a shared CSP channel as presented in Section 8. We will examine an SCSCP network $V = P_1 \;||\; \ldots \;||\; P_n$ and algebraically transform it so the CSP channel is replaced with our *MANY_TO_ONE_CHANNEL*.

Our transformation replaces each channel communication with the equivalent *MANY_TO_ONE_CHANNEL*. We undertake the following steps:

1. We replace a CSP channel a in V with *MANY_TO_ONE_CHANNEL*.
2. We transform the process P_i to P_i' by replacing any $a!x \rightarrow Process$ by $write.a!x \rightarrow ack.a \rightarrow Process$.
3. We transform the process P_j to P_j' by replacing any $a?x \rightarrow Process$ by $start\_read.a \rightarrow read.a?x \rightarrow Process$.

We will show that the external behaviour of subnetwork $P_i||P_j$ is unchanged by this transformation. We need to prove that:

$$(P_i \;||\; P_j) \setminus \{a\} = \\ (P_i' \;||\; MANY\_TO\_ONE\_CHANNEL(a) \;||\; P_j') \\ \setminus \{write.a, ack.a, start\_read.a, read.a\}$$

Starting with the RHS, we replace *MANY_TO_ONE_CHANNEL* with two *LEFT* and one *RIGHT* processes, hiding the internal a.

$$(P_i' \;||\; MANY\_TO\_ONE\_CHANNEL(a) \;||\; P_j') \\ \setminus \{write.a, ack.a, start\_read.a, read.a\} = \\ (P_i' \;||\; (((LEFT(a) \;|||\; LEFT(a)) \;||\; RIGHT(a)) \setminus \{a\}) \;||\; P_j') \\ \setminus \{write.a, ack.a, start\_read.a, read.a\}$$

As P'_i and P'_j have had all occurrences of a we can move the internal hiding set to the outside of the definition.

$$(P'_i \;||\; (((LEFT(a) \;|||\; LEFT(a)) \;||\; RIGHT(a)) \setminus \{a\}) \;||\; P'_j) \setminus \{write.a, ack.a, start\_read.a, read.a\} =$$
$$(P'_i \;||\; ((LEFT(a) \;|||\; LEFT(a)) \;||\; RIGHT(a)) \;||\; P'_j) \setminus \{a, write.a, ack.a, start\_read.a, read.a\}$$

Also, as the *write* and *ack* pair and the *start_read* and *read* pair only exist in P'_i and P'_j respectively, we can rewrite the hiding as follows:

$$(P'_i \;||\; ((LEFT(a) \;|||\; LEFT(a)) \;||\; RIGHT(a)) \;||\; P'_j) \setminus \{a, write.a, ack.a, start\_read.a, read.a\} =$$
$$(((P'_i \;||\; (LEFT(a) \;|||\; LEFT(a))) \setminus \{write.a, ack.a\}) \;||\; ((RIGHT(a) \;||\; P'_j) \setminus \{start\_read.a, read.a\})) \setminus \{a\}$$

We therefore need to prove the following:

$$(P'_i \;||\; (LEFT(a) \;|||\; LEFT(a))) \setminus \{write.a, ack.a\} = P_i \text{ and} \quad (1)$$
$$(RIGHT(a) \;||\; P'_j) \setminus \{start\_read.a, read.a\} = P_j \quad (2)$$

P_i is the interleaving of two writing processes on the channel a, meaning $P_i = (\mu q.a!x \to q) \;|||\; (\mu r.a!x \to r)$. P_j is a single reading process, meaning $P_j = a?x \to P_j$.

For equation 1, we will distinguish between the two $LEFT(a)$ processes as *LEFT* and *LEFT'*:

$$LEFT(a) = write.a!x \to ack.x \to LEFT(a)$$
$$LEFT'(a) = write.a!y \to ack.y \to LEFT'(a)$$

With our restricted SCSP syntax, we know equation 1 is true as:

1. When transforming P_i to P'_i we replaced $a!x \to PROCESS$ with $write.a!x \to ack.a \to PROCESS$.
2. The parallel combination of $LEFT(a) \;|||\; LEFT'(a)$ and P'_i produces the process $(\mu q.write.a!x \to a!x \to ack.a \to q) \;|||\; (\mu r.write.a!y \to a!y \to ack.a \to r)$.
3. Hiding $\{write.a, ack.a\}$ of the process $(\mu q.write.a!x \to a!x \to ack.a \to q) \;|||\; (\mu r.write.a!y \to a!y \to ack.a \to r)$ gives us $(\mu q.a!x \to q) \;|||\; (\mu r.a!y \to r)$ which is the same definition of P_i.

For equation 2, we follow the same steps thus concluding that the transformation to replace a CSP channel with a *MANY_ TO_ ONE_ CHANNEL*(a) did not affect the external behaviour of the network. We end up with our original example:

$$((\mu q.a!x \rightarrow q) \;|||\; (\mu r.a!y \rightarrow r)) \;||\; (\mu p.a?z \rightarrow p)$$

As can be seen, it would be easy to increase the number of *LEFT* processes to any number as the transformation provides demonstrates these processes become an interleaving of writing processes.

A similar process can be undertaken to demonstrate the equivalence of a *ONE_ TO_ MANY_ CHANNEL* by introducing two *RIGHT* processes. *MANY_ TO_ MANY_ CHANNEL* also follows a similar process but with two *LEFT* and *RIGHT* processes. Note, in the CSP model the exclusive access to the *transmit* channel is ensured by reading and writing processes being *interleaved* and the *transmit* channel access being nondeterministic.

11. Discussion

In this paper we have proved that the modelled implementation of shared channels in ProcessJ will behave correctly if implemented according to this model:

- We get failures refinement (and there is never any divergence) if the number of schedulers is equal to or greater than the number of processes.
- If less, we only get trace-refinement because some traces are no longer possible. This does not mean that the system will deadlock or behave inappropriately, it just means that some traces are no longer realisable.

There are two questions that need answering, namely, why do we only get trace refinement in the $Spec \sqsubseteq Impl$ case and why does refinement checking fail for some cases in the $Impl \sqsubseteq Spec$ scenario.

Let us tackle the first question: the run queue that ready-to-run processes go into naturally imposes an order on processes. Remember, processes in the run queue are ready to run; all they need is a scheduler to run them. If we consider the simple case with just two processes, say, P_A and P_B in the run queue like this: $[P_A, P_B]$, and two schedulers, then it should be reasonably clear that these schedulers each can get a process to run. All

possible interleavings of events from running P_A and P_B concurrently are possible. However, if we only have a single scheduler available, only P_A will be executed and only events from this execution are observable. However, once P_A yields, the scheduler will run P_B and we can observe all its events. In reality, in the latter case the fact that both P_A and P_B could be run concurrently is of no consequences and they end up being run **in sequence** as there is only one scheduler to handle both processes, but only one at a time. P_A and P_B both have acceptance sets; that is, the set of events they are willing to engage on after a certain trace. If there are schedulers enough to run all ready processes, then the system's overall acceptance set is the union of the acceptance sets of every process in the system. However, if, as above, there are not enough schedulers, then any process that does not have a scheduler will not have its acceptance set included in the overall system's acceptance set. In a non-scheduled CSP system this is never a problem as events can happen when they are ready—they do not require any scheduler or runner code to execute them. This explains why failures refinement can never happen when there are not enough schedulers to run every ready process. At this point it is worth noting that even though we do get traces refinement, the serialisation that happens when not enough schedulers are available also reduces the number of possible traces. That is, a scheduled system with less schedulers than processes may result in some traces never being possible.

Now, once we realise that it becomes clear that when checking refinement in the other direction ($Impl \sqsubseteq Spec$) fails—even in the traces model—since there are now traces that the $Spec$ can perform that the $Impl$ never could, simply because of a lack of schedulers to maximise concurrency in the execution. CSP does not have any notion of "the order of processes being executed"; events happen instantaneously. However, that is not how things work in the real world. Code needs to be run and synchronising on events takes time.

12. Future Work

Surely speed and efficiency in a concurrent system is a design goal, but so is correctness. After all, a system that is incorrect is of no use to any one. Looking at the proposed implementation of shared channels presented in this paper, we note the frequent use of locks/monitors. We have locks on:

- Entire processes. This is currently necessary to correctly handle setting the *ready* and *running* flags and to correctly place processes in the run queue.
- Entire channels. This is necessary in order to avoid race conditions when communicating on the channel.

Locking data reduces concurrency, which will impact the speed with which the code runs. Therefore, we want to consider removing the use of locks and replacing them with atomic data structures. To investigate this potential improvement, we need to develop CSP models for atomic variables and a wait-free queue:

- Atomic variables for the fields which 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 the v to a new value (n).
- A wait-free-queue to replace the current queues in the runtime. A wait-free queue allows multiple processes to access it safely at the same time.

Given that the ProcessJ implementation of channels does not meet the specification without enough scheduling resources, we do not have a general, lightweight construct to build CSP models of ProcessJ systems. To model systems, we will require a simplified specification of how ProcessJ performs channel communication. This simplification can be tested against our ProcessJ implementation models to ensure they behave as desired. Such simplified models can then be used to build concurrent systems to analyse how ProcessJ will behave under different scheduling conditions for different scheduling problems. Of further interest is investigating the number of schedulers needed to ensure failures refinement in both directions. We have shown that if every process gets its own scheduler, then failures refinement holds in both directions, but is it possible that we can do with a number less that the number of processes in the system. This is an interesting topic to investigate and naturally, the answer will depend on the program being run.

References

[1] J. B. Pedersen, P. H. Welch, The symbiosis of concurrency and verification: Teaching and case studies, Form. Asp. Comput. 30 (2018)

239–277. URL: https://doi.org/10.1007/s00165-017-0447-x. doi:10.1007/s00165-017-0447-x.

[2] C. Hoare, Communicating Sequential Processes, CACM 21 (1978) 666–677.

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

[4] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A. Roscoe, FDR3 — A Modern Refinement Checker for CSP, in: E. Ábrahám, K. Havelund (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, volume 8413 of *Lecture Notes in Computer Science*, 2014, pp. 187–201.

[5] J. B. Pedersen, K. 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, IEEE Press, Piscataway, NJ, USA, 2019, pp. 65–74. URL: https://doi.org/10.1109/FormaliSE.2019.00016. doi:10.1109/FormaliSE.2019.00016.

[6] J. B. Pedersen, K. Chalmers, Toward Verifying Cooperatively Scheduled Runtimes Using CSP, Formal Aspects of Computing 35 (2023). URL: https://doi.org/10.1145/3605942. doi:10.1145/3605942.

[7] K. Chalmers, J. B. Pedersen, Communicating Cooperatively Scheduled Processes: On the Unlikelihood of Implementing a Pure CSP Channel (2024).

[8] C. Shrestha, J. Pedersen, JVMCSP—Approaching Billions of Processes on a Single-Core JVM, in: K. Chalmers, J. Pedersen, J. Broenink, B. Vinter, P. Welch (Eds.), Proceedings of Communicating Process Architecture 2016, IOS Press, Amsterdam, The Netherlands, 2016.

[9] C. G. Ritson, A. T. Sampson, F. R. M. Barnes, Multicore scheduling for lightweight communicating processes, in: J. Field, V. T. Vasconcelos (Eds.), Coordination Models and Languages, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 163–183.

[10] A. L. D. Moura, R. Ierusalimschy, Revisiting coroutines, ACM Trans. Program. Lang. Syst. 31 (2009). URL: https://doi.org/10.1145/1462166.1462167. doi:10.1145/1462166.1462167.

[11] Y.-A. Chen, Y.-P. You, Structured concurrency: A review, in: Workshop Proceedings of the 51st International Conference on Parallel Processing, ICPP Workshops '22, Association for Computing Machinery, New York, NY, USA, 2023. URL: https://doi.org/10.1145/3547276.3548519. doi:10.1145/3547276.3548519.

[12] M. Chabbi, M. K. 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, Association for Computing Machinery, New York, NY, USA, 2022, p. 474–489. URL: https://doi.org/10.1145/3519939.3523720. doi:10.1145/3519939.3523720.

[13] F. Cesarini, S. Thompson, Erlang programming: a concurrent approach to software development, O'Reilly Media, Inc., 2009.

[14] C. Tismer, Continuations and stackless python, in: Proceedings of the 8th international python conference, volume 1, 2000.

[15] B. Sufrin, Communicating scala objects., in: CPA, volume 66, 2008, pp. 35–54.

[16] P. Welch, N. Brown, J. Moores, K. Chalmers, B. Sputh, Alting barriers: synchronisation with choice in java using jcsp, Concurrency and Computation: Practice and Experience 22 (2010) 1049–1062. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1471. doi:10.1002/cpe.1471. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe

[17] L. NamvariTazehkand, A. Ebnenasir, A novel approach for specification and verification of symmetric distributed algorithms using spin, in: 2024 Third International Conference on Distributed Computing and High Performance Computing (DCHPC), IEEE, 2024, pp. 1–9.

[18] L. Lamport, A fast mutual exclusion algorithm, ACM Trans. Comput. Syst. 5 (1987) 1–11. URL: http://doi.acm.org/10.1145/7351.7352. doi:10.1145/7351.7352.

[19] M. M. Michael, M. L. Scott, Simple, fast, and practical non-blocking and blocking concurrent queue algorithms., Technical Report, ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE, 1995.

[20] P. Welch, J. Martin, Formal Analysis of Concurrent Java Systems, in: P. Welch, A. Bakkers (Eds.), Communicating Process Architectures 2000, volume 66, WoTUG-31, IOS Press, Amsterdam, The Netherlands, 2000, pp. 275–301. ISBN: 978-1-58603-907-3.

[21] G. Lowe, Discovering and correcting a deadlock in a channel implementation, Formal Aspects of Computing 31 (2019) 411–419. URL: https://doi.org/10.1007/s00165-019-00487-y. doi:10.1007/s00165-019-00487-y.

[22] SGS-THOMSON Microelectronics Limited, occam 2.1 Reference Manual, Prentice-Hall, 1995.

[23] A. Sampson, C. Ritson, M. Jadud, F. Barnes, P. Welch, occam-π Home Page, Systems Research Group, University of Kent, http://occam-pi.org/, 2010.

[24] E. Bruneton and R. Lenglet and T. Coupaye, ASM: a code manipulation tool to implement adaptable systems, in: Adaptable and extensible component systems, 2002.

[25] A. W. Roscoe, The theory and practice of concurrency, Prentice Hall, 1998. URL: http://www.cs.ox.ac.uk/people/bill.roscoe/publications/68b.pdf, the text book teaching material can be found at http://www.comlab.ox.ac.uk/publications/books/concurrency/.

[26] A. Roscoe, Understanding Concurrent Systems, Springer, 2010. URL: http://www.comlab.ox.ac.uk/ucs.

[27] FDR 4.2.7 Documentation, https://cocotec.io/fdr/manual, ???? Accessed: 2022-08-25.