https://doi.org/10.1007/s00165-017-0447-x

Formal Aspects of Computing (2018) 30: 239–277

Formal Aspects of Computing

The symbiosis of concurrency and verification: teaching and case studies

Jan B. Pedersen[1] and Peter H. Welch[2]

[1] Department of Computer Science, University of Nevada Las Vegas, Las Vegas, NV 89154-4019, USA
[2] School of Computing, University of Kent, Canterbury CT2 7NF, UK

Abstract. Concurrency is beginning to be accepted as a core knowledge area in the undergraduate CS curriculum—no longer isolated, for example, as a support mechanism in a module on operating systems or reserved as an *advanced* discipline for later study. Formal verification of system properties is often considered a *difficult* subject area, requiring significant mathematical knowledge and generally restricted to smaller systems employing sequential logic only. This paper presents materials, methods and experiences of teaching concurrency and verification as a unified subject, as early as possible in the curriculum, so that they become *fundamental* elements of our software engineering tool kit—to be used together every day as a matter of course. Concurrency and verification should live in symbiosis. Verification is essential for concurrent systems as testing becomes especially inadequate in the face of complex non-deterministic (and, therefore, hard to repeat) behaviours. Concurrency should *simplify* the expression of most scales and forms of computer system by reflecting the concurrency of the worlds in which they operate (and, therefore, have to model); simplified expression leads to simplified reasoning and, hence, verification. Our approach lets these skills be developed without requiring students to be trained in the underlying formal mathematics. Instead, we build on the work of those who have engineered that necessary mathematics into the concurrency models we use (CSP, π-calculus), the model checker (FDR) that lets us explore and verify those systems, and the programming languages/libraries (occam-π, Go, JCSP, ProcessJ) that let us design and build efficient executable systems within these models. This paper introduces a workflow methodology for the development and verification of concurrent systems; it also presents and reflects on two open-ended case studies, using this workflow, developed at the authors' two universities. Concerns analysed include safety *(don't do bad things)*, liveness *(do good things)* and low probability deadlock *(that testing fails to discover)*. The necessary technical background is given to make this paper self-contained and its work simple to reproduce and extend.

Keywords: Process-orientation, Concurrency, Deadlock, Event ordering, Liveness, Verification, Occam-π, CSP

Correspondence and offprint requests to: P. H. Welch. E-mail: p.h.welch@kent.ac.uk

1. Introduction

Concurrency and formal verification, if they are taught at all at undergraduate level, are taught as distinct subjects. Both are viewed as *difficult*. The novelty in what we offer in this paper is a way to teach both together in such as way that they reinforce each other and enable deeper understanding and application.

If these are taught early in the curriculum, they become second nature and provide a stronger foundation for education in computer engineering. The need for concurrent systems has snowballed over the last decade (not only in hardware but also in software) so that traditional sequential programming skills, mastered by the average programmer, are no longer sufficient. The need for verification is even more important for concurrent systems because of their additional complexity. We believe improving such skills will improve the way future complex systems are developed, tested, and deployed.

The ideas reported in this paper were developed as a collaboration between two institutions: the University of Kent at Canterbury in the United Kingdom (Welch) and the University of Nevada Las Vegas, in the United States (Pedersen).

At Kent, concurrency has been a course module at the undergraduate level since 1986. Mostly, it has taken place during the second year. For a brief time, we were allowed to teach a shorter version during the first year. CSP and formal verification was taught as a separate optional module, again in year two. Teaching them in a combined way, as discussed in this paper, commenced in 2010 in the second year. Students at this stage were familiar only with object-oriented programming through Java and with only a modest mathematical foundation. At UNLV concurrency and verification has been taught together as one part of a graduate course (for students already holding a B.Sc. or equivalent in computer science), which has been offered three times.

1.1. Concurrency is essential

One of the wrong turns taken by Computer Science, both academic and industrial, over the past 60 years has been the focus on *serial* forms of computing (be that 'structured', 'object oriented' or even, despite appearances, 'functional'). *Concurrency*, when taught or practiced, has been treated as an advanced topic—only to be approached once we have become comfortable with sequential programming and, only then, as a *last resort* (e.g. to reduce response latencies in a real-time application or to make efficient use of a parallel supercomputer). But we see concurrency as a fundamental mechanism of the universe, existing in all structures and at all levels of granularity. To be useful in this universe, any computer system has to model and reflect an appropriate level of abstraction. *For simplicity*, therefore, the system needs to be concurrent [Rob12], [Wel13a]—so that this modelling is obvious and correct. If you share this intuition, there are radical consequences for the ways we teach and practice computer science. In particular, we should teach sound concurrency ideas and practices at the same time as introducing sequential programming [ACM12].

Today, the commercial reality of multi-core processors means that concurrency issues can no longer be ducked if applications are going to be able to exploit more than an ever-diminishing fraction of their power. This is a mean, but very forceful, reason to take this subject seriously.

This paper presents some of the ways in which we have been teaching concurrency at the undergraduate and master's level. We focus on the model of concurrency known as Process Oriented Design [Wel00], [WP10], [Wel13a], which is based on the formal process algebras of Hoare's Communicating Sequential Processes (CSP) [Hoa85, Ros97, Ros10] and Milner's π-calculus [Mil99]. The materials and case studies in this paper are designed to illustrate the most important concepts of this approach to concurrency. It is our experience, from almost 30 years of developing this teaching, that students do not have difficulties in understanding and applying the ideas. On the contrary, fluency seems fairly easy to acquire (because the mechanisms correspond well with our experience of the way the world works and, so far as software engineering is concerned, enable compositional reasoning and no conflict with everything we expect from sequential programming). The reason we are teaching these concepts is because they are not generally taught and this seems wrong to us: they enable concurrency to become a standard part of the software engineering tool kit without which modern scalable complex systems cannot safely be built.

1.2. The need for verification

Multi-core architectures are now standard, with the number of cores per processor growing each year. Multi-processor networks are inescapable for super-computing problems and most forms of embedded computer platform. Programmers (and students) cannot avoid concurrent reasoning when dealing with these devices—avoidance leads to many bad things. Verification of this concurrent reasoning is mostly set aside (as it has generally been for sequential reasoning, we admit). A significant amount of professional development time and money is spent instead on testing software. However, testing and debugging concurrent programs is even more difficult than for sequential programs—common faults are intermittent and not reproducible on demand. If the concurrency pattern is beyond the embarrassingly parallel (i.e., the processes need to engage with each other) and we have made some mistakes in design or coding, testing may *never* see these faults and our system will eventually fail in service. So, we need to verify.

Therefore, just as we need tools (e.g. programming languages and integrated development environments) to produce executable systems, we need tools (e.g. model checkers) to produce verified systems. Language and model checker pairs need to live to the same concurrency model. All these tools need to be integrated, and taught and used together! We need theory and programming technology that turns this around and makes concurrency an elementary part of the every day tool kit of every software engineer. This is what we teach.

1.3. Research led teaching: our experiences

Since 1986, concurrency has been a major module in the Computer Science undergraduate (and taught Masters) curricula at the University of Kent. Initially prompted by the development of the INMOS transputer, our courses have evolved to provide a secure foundation for the design, implementation and analysis of concurrent and parallel systems across a wide field of application. We have, however, kept to our root principles of a concurrency model based on the formal algebras of CSP and the π-calculus. These ideas are not taught through the formal mathematics, but through structured diagrams, lots of programming, rigorous argument and fun with robotics (virtual and real). The courses are elective, advertised to and attracting only those students who enjoy and want to program – currently about half our cohort (approximately 100 students per year). We engage our students with our research in this field—for example, complex systems modelling and emergent behaviour (CoSMoS [SWT+07], To-Boldly-Go [WWSK12]), reactive embedded systems (RMoX [JBV03, Bar05]), programming language design (occam-π [WB05a, BW04, WP10, WB05b, WB08, WPB+11, WPBR11, WB11b, RW10, Sam08, SRJ+10, BWMW10]), concurrency libraries for mainstream languages (such as JCSP for Java [WBM+10, WBM+07, WB11a], C++CSP for C++ [BW03, Bro10a] or CHP for Haskell [Bro08, Bro10b]), efficient run-time kernels for multi-core processors (CCSP [RSB12]) and a portable interpreter to fit on small memory platforms (the Transterpreter [JJ04]).

Recently, we have explored teaching verification in a course at the University of Nevada, Las Vegas. Some of the material presented in this paper is the result of a live exercise during lecture time. This examines a 3 process system synchronising over 11 channels and a single barrier. The students are asked to describe the opening behaviour of this system, at first reasoning informally using their intuitive understanding of the semantics of process synchronisation. Their conclusions are verified using a formal model-checker. Deeper questions are then raised about various safety (*"do no wrong"*) and liveness (*"do right"*) requirements[1] that the system has to satisfy throughout its operation. The interesting, and sometimes dark, arts of model-checking are introduced to answer these questions—but in a way that requires no deep mathematical skills and is very close to programming (in which they have skills). Another exercise at the University of Kent concerns a safety-critical system that seems well-programmed and passes days[2] of soak-testing. Yet a potential for deadlock is immediately exposed by the model-checker. Finding the cause, fixing it and *verifying the fix* considerably enhances understanding, demonstrates the importance of program verification and shows that the latter can and should be done as part of normal programming activity. Presenting the (simple) technical background to these exercises and reporting on their success is the main theme of this paper.

[1] Safety and liveness are considered in the sense given by CSP—see the opening paragraphs of Sect. 5.5.4—not in the sense of dependable systems.

[2] It would actually pass *years*.

1.4. Paper organization

In Sect. 2, we describe a methodology for developing verified applications using the concurrency model of process oriented design. Section 3 briefly introduces *process orientation* and the concepts needed for the rest of the paper. Section 4 shows a language binding for process-oriented systems, which allows us to write executable code.[3] Sections 5 and 6 present the two case studies that provide the foundation for this paper, and finally, Sect. 7 concludes and reflects on the lessons learned. A smaller case study (of a conundrum that surprised Ben-Ari [BA10]) is given in an Appendix. Where relevant, we will introduce each section or subsection with a box of the form:

> **Intentions:** What did we want the students to learn?
> **Methodology:** What part of the methodology from Sect. 2 does this support and how?
> **Questions:** What we ask the students.

Not all three items are relevant to all sections; those which are irrelevant are left out.

1.5. On-line resources

All the (occam-π) executable sources and the (CSP) formal models discussed in this paper are available on-line at the supporting website [PW15]: http://SantaClausProblem.net/verification. These include full sources for additional soak-testing for the second case study (Sect. 6) and the small study in the Appendix.

2. Concurrency and verification methodology

Figure 1 illustrates the workflow between the problem domain and seven areas of work in the development of verified systems utilising concurrency and process-oriented design.

1. **Problem domain**. This is the task of developing a description of the problem in terms of its environment and not in terms of computing.
2. **Process oriented design**. This is the modelling of the problem environment as a structured network of communicating processes, with each process responsible for managing the state and behaviour of the individual entities in the system.
3. **Assertion design**. Assertion design is concerned with determining (a) constraints to which the system must adhere and (b) positive behaviours that the system must supply. Assertions are often designed iteratively along with the process-oriented design. They may need revision later if subsequent verification fails or unexpected behaviour appears.
4. **Executable Model**. The executable model is the actual program source code.
5. **Formal Model**. This is a description of system behaviour in a formal process algebra. We use CSP, which can express a rich set of behaviours (e.g. liveness and safety issues) and is supported by a powerful model checker (FDR). The formal and executable models are two sides of the same coin. Both need to be done, but the order does not matter. In fact, this is normally an iterative process.
6. **System**. The system is the execution of the code in a test or actual environment.
7. **Verification**. This is the task of verifying that the formal model satisfies the assertions, and is done though a combination of formal model checking and deductive reasoning. If any part of the verification fails, this is either because of an error in the formal model (i.e., the intended behaviour has not been captured), an incorrect assertion about the intended behaviour has been made or the process-oriented design is incorrect—checked usually in that order.

[3] Sections 3 and 4 are minor revisions of sections 2 and 3 of an earlier work [WP10]: *"Santa Claus: Formal Analysis of a Process-Oriented Solution"*, in ACM Trans. Program. Lang. Syst., {32, 4, Article 14, (April 2010)} ©ACM, 2010. http://doi.acm.org/10.1145/1734206.1734211. They are included here for the benefit of readers unfamiliar with these technologies.

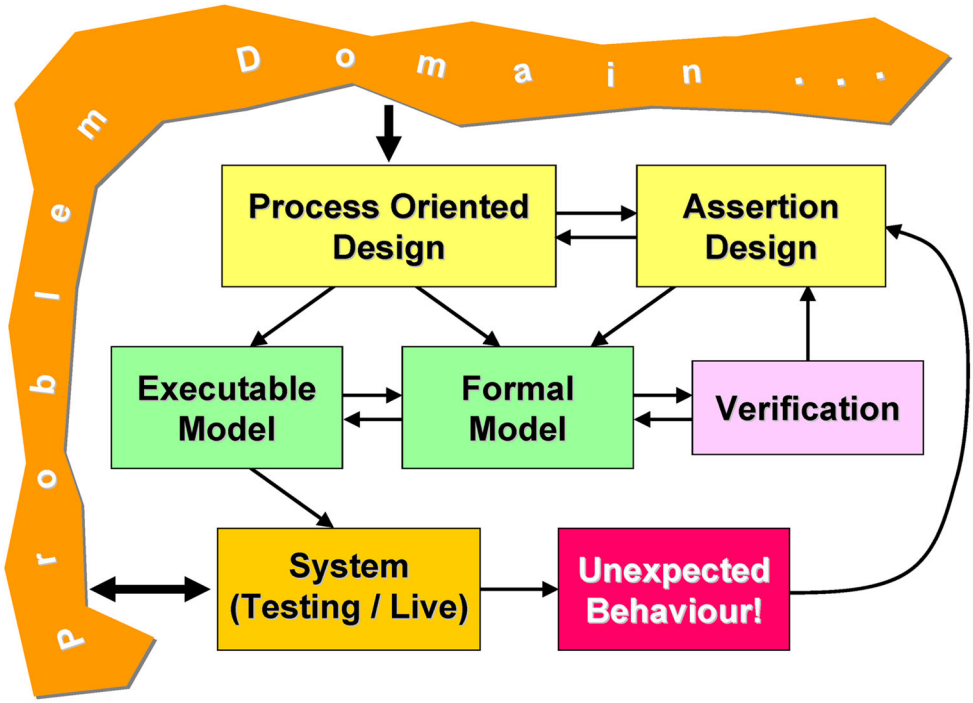


Fig. 1. Concurrency and Verification Workflow

8. **Unexpected Behaviour**. If this happens, it means that assertions denying the behaviour were not made. Such assertions must now be added and these will cause the verification to fail; the process-oriented design must then be revisited and corrected alongside consequent changes in the formal and executable models.

This development methodology forms the basis for how we teach process-oriented design with formal verification. This methodology is orthogonal to the typical software engineering development methodologies/models and therefore does not conflict with them. This means that our methodology can be incorporated into larger software development models when concurrency is considered.

In the first case study (Sect. 5) we show a simple use of this model that does not result in any unexpected behaviour, but utilises the model checker to explore and verify assertions about the program/model that we are implementing.

The second case study (Sect. 6) exposes students to a system where informal reasoning is persuasive of correct behaviour and extensive soak testing discovers no problem. However, a formal check immediately uncovers a possibility of deadlock (that would have catastrophic consequences in the live application). The students have to deal with this deadlock, considering various redesigns and finding a solution that passes formal verification (whilst maintaining all other assertions about required behaviour).

The Appendix gives a small example by Ben-Ari [BA10] showing behaviour not even he was expecting. His paper expresses the system formally in Promela and uses the Spin model checker [Hol03] to verify that the unexpected behaviour really can happen. We present a simple reworking of this example in CSP, following the use of our methodology, to show easily this can be done.

3. Process orientation

Process oriented design is an example of component-connector engineering. The components are active processes and the connectors are events (their *alphabets*) through which they synchronise and communicate. Key concepts are processes, channels, barriers, networks, network hierarchies, choice, protocols and synchronisation patterns. To be practical, a process-oriented programming language, or a library providing the necessary support for other languages, is essential—otherwise, the gulf between the theory underpinning the design and its realisation in code presents uncomfortable obstacles. Such tools must be easy to learn and use and have reasonably efficient implementation. Fortunately, all these exist—we just have to rise to the challenge of trying them.

3.1. Processes

A *process* is a self-contained self-executing unit that encapsulates private data and algorithms. This contrasts with object oriented programming where object methods are executed by an external caller's thread of control. An object is passive (it does nothing unless a method is invoked) whereas a process is active and can take the initiative. A process has sole control over its internal resources and no control (not even visibility) of the resources of another process. Interaction with other processes happens indirectly through synchronising primitives, such as channel communication and barrier synchronisation. Crucially, a process can *refuse* some, or all, of its external events—thereby blocking demands from other processes until it is in a good state to *accept* them. An object cannot refuse a method invocation, no matter its internal state; a *synchronized* method may block for a while, but it cannot ultimately refuse invocation.

3.2. Synchronising channels

The simplest form of process interaction is point-to-point synchronous unidirectional message passing along a zero-buffered *channel*. A channel has a sending end and a receiving end, though it is possible to share these between multiple senders or receivers. Zero-buffering means that a sender process must block if no receiver is ready (and vice-versa). Various kinds of channel buffering (e.g. blocking or overwriting FIFOs) can be obtained through splicing in appropriate buffer processes between the sender and receiver.

These communications differ from those in common message passing libraries for parallel computing. For example, in MPI [Don94] *any* process knowing the process identifier of a receiving process (within one of its own communicator groups) can send it a message. In CSP, there are named process types but individual processes have no names (or identifiers). Individual processes are bound to a particular set of events (channels, barriers, etc.) that *do* have names. Different instances of the same process type can, of course, be bound to different sets of events. A process sends to a named channel and whatever process has the other end receives.

Network connectivity is explicit, dynamic and constrained to what the system needs. The difference is subtle but is an important part of ensuring the **compositionality** of processes in CSP: the semantics of a process depends only on the process and is not changed by the presence of other processes in the system.

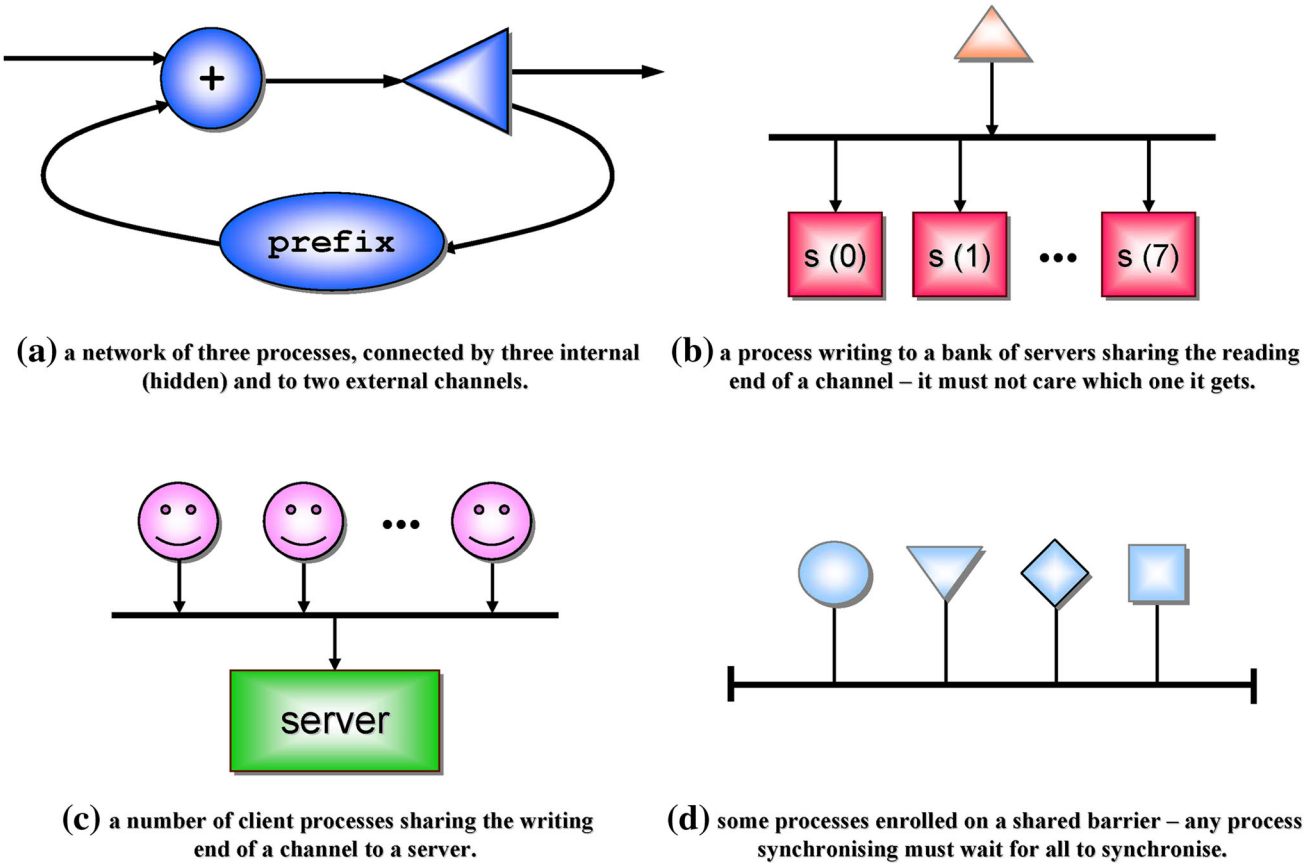
Processes cannot observe or modify each others' state, so need no locking mechanisms to maintain data integrity. To observe or modify such state, a process must communicate a request to the owning process via appropriate channels. That request may be ignored by the target process (blocking the requester) until such time as it chooses (e.g. when the request can be correctly processed). This means that reasoning about process behaviour can always be conducted *locally*—a process is in complete charge of its state.

The size of the state space of a process network is bound by the product of sizes of the state spaces of its component processes (less those that cannot be reached through the constraints of process synchronisation). Thus, the state space of a process network can grow large whilst the logic of its components remains simple. It is this gearing—together with a *compositional* semantics—that delivers the power of process-oriented design.

In contrast, threads concurrently managing shared state through locking mechanisms (mutexes, semaphores or monitors) have to be secure in the face of all possible interleavings through the shared objects. Reasoning is *non-local*: the logic of an individual method, class or thread cannot be devised, or understood, on its own. This is hard.

3.3. Synchronising barriers

Channels require two processes (the sender and receiver) to synchronise. A barrier is an event on which *many* processes can be enrolled and on which *all* must synchronise together. If one process offers to synchronise on a barrier, all must offer to synchronise for the event to happen—everyone must wait for everyone. A process may have any number of barriers in its alphabet. Unlike a channel, a barrier synchronisation communicates no data—only the fact that the synchronisation happened.



(a) a network of three processes, connected by three internal (hidden) and to two external channels.

(b) a process writing to a bank of servers sharing the reading end of a channel – it must not care which one it gets.

(c) a number of client processes sharing the writing end of a channel to a server.

(d) some processes enrolled on a shared barrier – any process synchronising must wait for all to synchronise.

Fig. 2. Process oriented design: components and connectors (from [Wel13a])

3.4. Networks

A network is simply a parallel composition of processes (which may themselves have internal networks), connected through a set of synchronising events (channels, barriers etc.). A network usually hides the events connecting internal components, leaving free those to be used for external connections. A network is, therefore, also a process. Network topologies can be constructed dynamically and may evolve (both in shape and size) in response to their environment.

3.5. Design by pictures and composition

Processes do not know—or need to know—with whom they are synchronising. Each process can be viewed as a *black box*, whose ties to its environment is a set of events (channel-ends, barriers, etc.)—its *alphabet* in CSP terminology. The behaviour of a process is described by the message structures allowed on its channels, the patterns of synchronisation with which it is prepared to engage on its channels and barriers, and the computational functions it performs. Networks of processes are simply built by 'wiring' them together using internal (hidden) channels and barriers. A network is itself a process—so hierarchical structures naturally emerge.

This method of construction has an obvious visual representation, lending itself to design through (structured) pictures—see Fig. 2. This should have resonance with hardware engineers, whose systems are physically concurrent. The discipline leads to a strong notion of *components* (the processes) and *connectors* (the synchronisation events), supporting concurrency, hierarchical design and code reuse. The processes run themselves and do not share memory. Innermost processes are sequential, require no locks, and synchronise using channels (i.e., external I/O operations) and barriers. The synchronisation semantics are simple and intuitive and all our skills for sequential programming remain valid.

We make frequent use of such diagrams when designing and discussing process-oriented programs.

3.6. Formal verification

Being able to reason formally about a program is valuable—crucially so if the application is safety or finance critical. Special difficulties arise with concurrently executing processes since the state space potentially explodes. If the concurrency formalism in which reasoning is conducted differs from the implementation primitives used, the reasoning is unsafe. If translation between the implementation and formal modelling languages is hard, maintaining coherence between the two will be a continuous overhead as the system evolves.

This gap between implementation and verification is reduced by using languages (or libraries) designed around formal methods for which verification tools exist. Almost all concurrency mechanisms within occam-π have a direct representation in CSP. FDR [GRABR16, GRABR14] is a model checker for CSP, allowing formal verification of freedom from deadlock and livelock, process refinement and equivalence—at least, for systems of finite (and sufficiently small) size. FDR has a long and successful history of use in the analysis of complex safety-critical systems [SD04, Bar95, HC02, BKPS97, BPS99, Low96, MS07].

Translation between occam and CSP is defined, [GRS93, GRS94], and can be automated. A tool, [BR10], exists to generate CSP automatically from occam-π, but this is not yet ready for general or classroom use. At present, we do this translation by hand and this paper gives several examples. In general, state space introduced by real programs (for example, a single 32-bit integer variable has potentially 4 giga-values) must be reduced to small finite numbers if the model checks are ever to terminate in acceptable times. Automating this raises challenges that are not the subject of this paper—see [WPB+11] for early ideas on this.

It should be noted that occam-π is not designed to be an execution engine for CSP—that is, translation from arbitrary CSP systems to occam-π is not always easy or, even, possible. Rather, occam-π is designed as a programming language with concurrency built in as a first-class mechanism, with a semantics directly expressed by CSP. It enables concurrency to be used with the same confidence, ease and overheads as, say, sequential procedures (or method invocation) and its formal basis enables verification.

4. A language binding for process oriented development

occam-π is an imperative statefull language built around the concurrency model of Hoare's CSP. Compiler enforced language rules prevent unsafe access to shared resources, so that no data race hazards can happen. Strict aliasing control enables this and provides a simple semantics for assignment. It extends the classical occam2.1 language [SGS95] through the introduction of *shared* channel-ends (modelled by CSP *interleaving*), barriers (corresponding to *multi-way* CSP events) and *mobility* (i.e., communication through channels) of those channel-ends and barriers (with semantics derived from the π-calculus [Mil99]).

The occam-π codes in this paper were developed with the KRoC [WW96, BWMW10] compiler, run-time system and library—an open-source project originated and hosted at the University of Kent. At present, compiled code is targeted only at IA32 platforms (taking full advantage of multi-cores). Memory overheads (up to 32 bytes per process) and run-time costs (the low tens of nanoseconds per synchronisation) enable millions of processes to be (multi-core) scheduled per processing node and perform useful work [RW07, RSB12]. An interpreted version (the Transterpreter [JJ04]) is available for almost any target platform, requiring a very tiny memory footprint. Two new compiler projects [Bar06, Sam07, SBR+10], targeting all platforms supported by a C compiler, are in development.

4.1. Processes, sequential composition and parallel composition

A *process* in occam-π is either a *primitive* or a *composition* of processes. A process, at any level, may make local declarations. A process may use its local declarations or anything declared globally (and not hidden)—normal block structuring rules apply.

It is just as easy, *syntactically*, to compose processes for sequential execution as it is for parallel:

```
SEQ                         PAR
  ...  process A              ...  process A
  ...  process B              ...  process B
  ...  process C              ...  process C
```

In sequential execution, each component sub-process may not start until the previous one has terminated. They may freely share and update global variables.

In parallel execution, all components run concurrently. The construct does not terminate until all its components have terminated. The components may only share global variables for reading: if one sub-process changes a global, the other sub-processes may not even look at it. These rules are statically checked and enforced by the compiler. Any component may have its own local variables.

The syntactic scope of occam-π structures is defined by *indentation*. This means that, in all circumstances, *what-you-see-is-what-you-get*. In languages where indentation has no syntactic significance and scope is defined, for example, by *curly brackets*, this is not true.

4.2. Primitive processes

There are ten forms of primitive executable process. The first is an assignment: evaluate some expression (RHS) and assign (`:=`) the result to a variable (LHS). Strong typing rules are enforced. Expression evaluation has no side-effects (as in a functional language). This, together with the strict anti-aliasing enforcement (no changeable entity can have different names in the same scope), means that the semantics of assignment is simple: the assigned variable is set to the assigned value *and nothing else changes*.[4]

Five other primitive processes are: channel input and output (Sect. 4.3), barrier synchronisation (Sect. 4.4), `SKIP` (which does nothing but terminate—sometimes needed for syntactic place holding), and `STOP` (which does nothing—not even terminate – and gives a concrete manifestation of deadlock, useful for semantic reasoning and model checking). Three more are obtaining time-stamps, setting timeouts and forking processes—but these are not used in this paper.

Finally, process *abstractions* may be named and parametrised:

```
PROC <name> (<parameters>)
  ...  process
:
```

The parameters may be any type, including data (by reference or value) and synchronisation elements (channel-ends and barriers). The colon marks the end of the declaration. A named abstraction may be invoked by its name and supplying correctly typed arguments—which is our final syntactic form of executable. Invoked in *sequence* with other processes, they may be thought of as procedures. Invoked in *parallel* with other processes, they become components of a network whose topology is determined by the synchronisation items they share.

The semantics of parameter passing in occam-π are one of *renaming* and do not introduce new variables—as such these process abstractions can be manipulated (e.g. refactoring program structure) without affecting the semantics of the program. This is not so easy in many other imperative languages, where parameters do introduce new variables.

4.3. Channels

Message passing happens through channel communication. Channels have a reading end and a writing end—they are unidirectional. A channel is declared as follows:

```
CHAN <type-list> <name>:
```

where `<type-list>` is a semi-colon-separated list of types.

The reading end of a channel is denoted by `<name>?` and the writing end by `<name>!`. To write to a channel named `c`, the syntax has the form:

```
c ! <expression-list>
```

where the types listed in the channel declaration and the individual expressions in the (semi-colon separated) list must match.

[4] This is not the case for most other imperative languages—such as C, Java, etc. where expressions may have side-effects and aliasing is uncontrolled.

Channels are zero-buffered, so a writing process will block until another process, running in parallel with it, executes a read on the other end of the same channel—for example:

```
c ? <variable-list>
```

where the types listed in the channel declaration and the individual variables in the (semi-colon separated) list must match.

A reading process will block until another process, running in parallel with it, executes a write on the other end of the same channel. Only when (or if) both processes reach these respective synchronisation points does the communication happen—whichever process gets there first must wait. After the communication, both processes go their separate ways.

(Note: when needed, any form of buffered channel is easy to make using a buffer *process*, with very low run-time overhead.)

4.4. Barriers

The last concept needed for this paper is the *barrier*. A barrier is multi-way synchronisation point. No process can proceed past the barrier until every process enrolled on the barrier has reached it. The syntax for declaring and enrolling processes on a barrier is as follows:

```
BARRIER <barrier-name>:

PAR ENROLL <barrier-name>
  ...  all processes here are enrolled
```

Synchronising on a barrier is the last occam-π primitive we need:

```
SYNC <barrier-name>
```

4.5. Choice

occam-π provides a simple way of waiting for one of a set of events to be offered and, then, making a response. Should more than one of these events become available, an *arbitrary* (i.e., non-deterministic) choice is made. An `ALT`ernative construct is a list of *guarded processes*:

```
ALT
  <guard>
    <process>
  <guard>
    <process>
  ...  etc.
```

The list order does not matter. The guards are the *waited-for* events—currently, only input processes (on offer when a message is pending), timeouts (on offer when expired) and `SKIP`s (always on offer) are allowed.

If control of the choice is needed should more than one event be on offer, a prioritised version (`PRI ALT`) is available. This resolves the choice in favour of the first one listed, so that the ordering of the guarded processes does matter in this case.

4.6. Conditions

Like most languages, occam-π provides a conditional structure for sequential control. An `IF` construct is a list of *conditional processes*:

```
IF
  <boolean-condition>
    <process>
  <boolean-condition>
    <process>
  ...  etc.
```

The ordering is significant. The conditions are evaluated[5] in sequence, stopping at the first one that is `TRUE` and executing the (indented) process that immediately follows. If all conditions evaluate to `FALSE`, this is a run-time error and the process `STOP`s.[6]

5. Case study: robot control system

> **Intentions:** The initial aims of this first study are to ensure proper understanding of the synchronisation mechanisms (channels and barriers), where verification checks that understanding. It then shows how questions about safe behaviour and liveness properties can be asked and verified. In addition to this, it shows how extra processes might need to be devised to enable verification of non-trivial application-specific behaviours by the model checker.

A simple component, highly abstracted, from a robot control system is presented. The intention of this case study is to check and reinforce correct understanding of process-oriented design (processes, channels, synchronisation and non-determinism), informal analysis of behaviour, and formal verification of a range of semantic questions through model checking and simple deductive reasoning. The study demonstrates five stages of the methodology shown in Fig. 1: Process Oriented Design, Assertion Design, Executable Model, Formal Model and Verification. Behaviour examined includes the initial orderings of signals to/from the device (how these are constrained by synchronisations between subcomponents within the device), what happens when its internal components starts looping, deadlock and livelock freedom, and operational safety ("do no bad") and liveness ("do good"). These behaviours, their design and analysis comprise the fundamental issues of the concurrency model that we are teaching. The study has been designed to be simple enough to be presented and understood by beginners, yet rich enough to provide non-trivial questions to be asked about these behaviours.

This case study has been developed from exercises originally presented and worked through live in a class at The University of Nevada Las Vegas (UNLV) over the last three years, as part of a concurrency module taught at the Masters level. Previously, they had studied in this course a range of approaches to concurrency, including material from the undergraduate Concurrency Design and Practice [Wel13b] course (an elective module taken by approximately half the cohort of second year CS undergraduate at the University of Kent since 2000).

By the time of this exercise, students were comfortable with using occam-π in several non-trivial projects (thousands of interacting processes). So, the example system here would be considered fairly small. Nevertheless, if the application were safety critical, it was appreciated that relying just on our intuition (based on understanding the low-level concurrency semantics of occam-π) was unsafe.

5.1. The device

> **Intentions:** introduce the concurrent system on which a range of basic questions about behaviour will be asked, together with the design of experiments that will reveal deeper semantic issues.
> **Methodology:** process-oriented design (Fig. 1).

We start by visualizing an apparently simple device, and continue with a brief explanation of its function and implementation.

[5] Note that this causes no side-effects.

[6] If no action is needed in such circumstances and execution should continue, a final `TRUE` condition must be appended to the list followed by a `SKIP`. Requiring the programmer to flag such inaction seems irritating at first, but pays dividends. The flag explicitly declares that *all* relevant conditions have been considered and this must be defended when the code is reviewed and/or analysed. Without such a flag, erroneously overlooked conditions cause an immediate `STOP`, which is much to be preferred over continued execution with bad state.

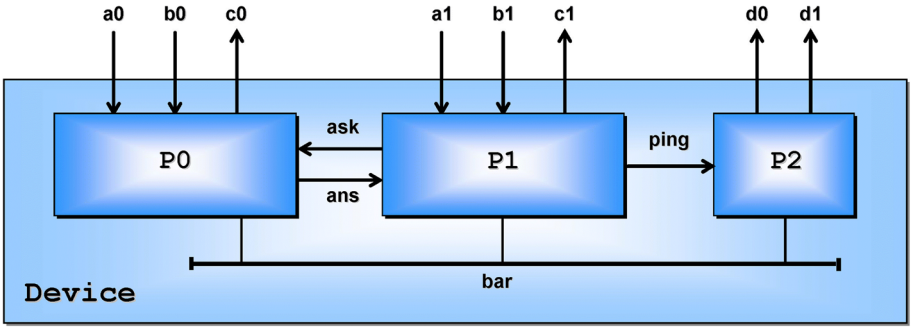


Fig. 3. Device with three sub-processes, 3 internal channels, one internal barrier, and 8 external channels

The diagram in Fig. 3 shows the internal structure of `Device`, a component of a real-time control system for an autonomous robot driving 8 channels (4 input and 4 output). There are 3 sub-components: `P0` (weapons systems), `P1` (vision processing), and `P2` (motion stabiliser) running concurrently. They exchange information using internal channels (`ask`, `ans`, `ping`) and coordinate some actions on an internal barrier (`bar`).

CSP semantics apply. Channel communication is unbuffered: the sender process must wait for the receiver and vice-versa (Sect. 3.2). Barrier synchronisation means that any process engaging on the barrier must wait until all processes (plugged into the barrier) engage—the last one unblocks them all (Sect. 3.3).

We present two representations defining the behaviour of `Device`: one in occam-π (for compiling to a runnable system) and one in CSP (for formal analysis). The representations are in one-to-one correspondence and our students have had little trouble shifting between them. Some behavioural questions we wish to check are:

Deadlock: might the `Device` stop (e.g., `P0` and `P2` want to synchronise on `bar`, but `P1` wants to communicate on `ping`)?

Livelock: might the `Device` get busy and refuse all external signals (e.g., `P0`, `P1`, and `P2` start engaging in an infinite sequence of internal channel or barrier synchronisations on `ask`, `ans`, `ping`, and `bar`)?

Safety: might the `Device` ever engage in an incorrect (and dangerous) sequence of external signals?

Liveness: will the `Device` stay alive, offering all permitted signal sequences?

5.2. Executable model

Intentions: present and explain the abstract logic for the three components in the `Device`. This is expressed through an implementation written in occam-π.
Methodology: executable model (Fig. 1).

For the safety and liveness analyses we want to make, data values and computations performed by this particular device are not relevant. For simplicity, they are omitted in the following, with all message content abstracted to zero. Here is the executable (occam-π) code:

```
PROC P0 (CHAN INT a0?, b0?, c0!, ask?, ans!, BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ? x   -- take question
      a0 ? y
      ans ! 0   -- return answer
                -- (depends on x & y)
      b0 ? z
      SYNC bar  -- wait for the others
      c0 ! 0
:
```

The code for P0 declares a process connected to its environment via 5 channels (3 for input, indicated by the "?" suffix qualifying the formal parameter name, and 2 for output, indicated by the "!" suffix) and one barrier. The body of the code shows its behaviour, which is an infinite loop performing the following actions in sequence: wait for a question on the ask channel, wait for a signal on a0, deliver an answer on the ans channel, wait for a signal on b0, synchronise on the barrier bar, and finally output a signal on c0. The codes for P1 and P2 follow similar patterns.

```
PROC P1 (CHAN INT a1?, b1?, c1!, ask!, ans?, ping!, BARRIER bar)
  WHILE TRUE
    INT x, y, z:
    SEQ
      ask ! 0   -- ask question
      ans ? x   -- wait for answer
      a1 ? y
      b1 ? z
      SYNC bar  -- wait for the others
      c1 ! 0
      ping ! 0  -- update neighbour
:

PROC P2 (CHAN INT d0!, d1!, ping?, BARRIER bar)
  WHILE TRUE
    INT x:
    SEQ
      SYNC bar  -- wait for the others
      d0 ! 0
      ping ? x  -- receive update
      SYNC bar  -- wait for the others
      d1 ! 0
      ping ? x  -- receive another update
:
```

The code implementing Device is a textual representation of the process network shown in Fig. 3.

```
PROC Device (CHAN INT a0?, b0?, c0!, a1?, b1?, c1!, d0!, d1!)
  CHAN INT ask, ans, ping:
  BARRIER bar:
  PAR ENROLL bar
    P0 (a0?, b0?, c0!, ask?, ans!, bar)
    P1 (a1?, b1?, c1!, ask!, ans?, ping!, bar)
    P2 (d0!, d1!, ping?, bar)
:
```

Note that three of the external channels from P0 (a0, b0, and c0) are connected directly to the same-named *external* channels of Device. The remaining two external channels from P0 (ask and ans) and its external barrier (bar) are connected to the same-named *internal* channels and barrier of Device. We have kept the names the same for convenience; these names are user-chosen and could be different here.

5.3. Informal analysis of the initial behaviour of Device

Intentions: To ensure correct understanding of synchronisation primitives and informally reason about the effects of these on program behaviour.
Methodology: Process-oriented design (Fig. 1)
Questions: What initial signalling sequences to/from the Device are possible?

Let us start by considering what patterns of signal are possible from Device, that is, what are possible orderings of signals on the *external* channels a0, b0, c0, a1, b1, c1, d0, d1? The *internal* signalling performed by Device is not visible to its environment and is not, therefore, part of its behaviour (as far as components using Device are concerned).

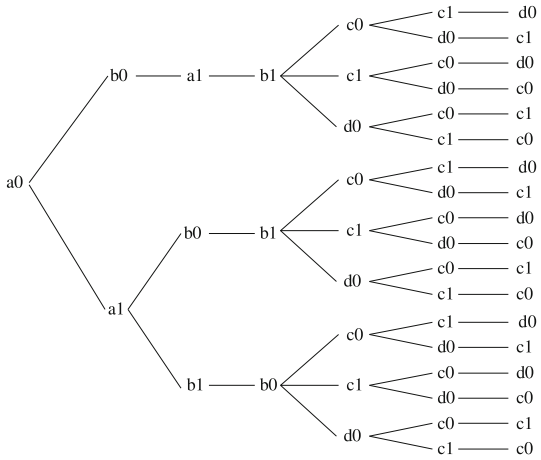


Fig. 4. The 18 possible orderings of the first 7 external signals

By inspecting the code, intuitively, `a0` is first. This comes from `P0`, which has just been asked a question by `P1` over the internal `ask` channel. `P1` cannot signal on `a1` until it has received an answer from `P0` on the internal `ans` channel. Furthermore, `P2` cannot get past its first barrier `bar`, as that requires `P0` and `P1` to also be present at the barrier.

Now, what is the second signal to happen? Since both `b0` in `P0` and `a1` in `P1` happen immediately following the internal communication on the `ans` channel, either of them could happen as the second signal.

If `b0` happened second, `a1` would definitely be third (again, no process can pass the barrier `bar` until all the processes reach that point). Obviously, `b1` would then follow as the fourth signal, and all three processes would be ready to synchronise on the barrier. In this circumstance, there can only be one event sequence (called a *trace*) to this point, which is ⟨`a0`, `b0`, `a1`, `b1`⟩.

If `a1` was the second signal, then either `b0` or `b1` will be third, and whichever was not third will be fourth. At this point, all three processes `P0`, `P1`, and `P2` have again reached the barrier. In this circumstance, two traces may have occurred: ⟨`a0`, `a1`, `b0`, `b1`⟩ or ⟨`a0`, `a1`, `b1`, `b0`⟩

Thus, three possible traces exist at this point. The *internal* barrier synchronisation now occurs.

The fifth, sixth and seventh external signals are `c0`, `c1` and `d0` in any order (6 possibilities). That gives us a total of 18 possible orderings of the first 7 signals. Figure 4 shows the 18 different signal orderings.

Are there any more first-7 signal sequences? What happens when the sub-processes start looping? Could `P0` signal again on `a0` *before* `P2` gave its first signal on `d0`? To be sure of answers to these questions and more, we need formal reasoning (as opposed to intuition and hand-execution). Fortunately, this is not hard.

Before leaving this section, we note that the diagram in Fig. 4 (with 18 initial traces) can be represented by the following *trace pattern* expression—where `;` is *concatenation* (reflecting sequential computation), and `|||` is *interleaving* (reflecting unsynchronised parallel computation):

⟨`a0`⟩; (⟨`b0`⟩ ||| ⟨`a1,b1`⟩); (⟨`c0`⟩ ||| ⟨`c1`⟩ ||| ⟨`d0`⟩)

The interleaving operator `|||` generates all possible combinations of traces of its operands. For example: ⟨`a`⟩ ||| ⟨`b`⟩ ||| ⟨`c`⟩ generates the following set of 6 traces, each with length 3: {⟨`a,b,c`⟩, ⟨`a,c,b`⟩, ⟨`b,a,c`⟩, ⟨`b,c,a`⟩, ⟨`c,a,b`⟩, ⟨`c,b,a`⟩}. Both operators return trace *sets* (the result for `;` contains just one element: the concatenation of its trace operands). Both operators also operate on trace sets: the result is the *union* of the operator applied to each pair of individual traces from its operand sets.

5.4. Formal model

Intentions: Learn the almost one-to-one correspondence between the executable and the formal model as well as learn some basic syntax of CSP_M.
Methodology: Executable and formal models (Fig. 1)
Questions: How do we reflect the occam-π in CSP?

We can formally verify the intuition of Sect. 5.3, and answer the open questions, with a CSP representation of the system. We write in CSP_M, the machine readable form used by FDR [GRABR16]. CSP treats channel communications and barriers in the same way: they are all events (declared as *channel*s in CSP_M). For our example system, we can abstract the channel communications even further by omitting the data sent (always zero) and the direction of communication (which is irrelevant to this formal analysis). The presentation is further simplified by not parametrising the process definitions,[7] which therefore operate directly on globally defined channels and barriers:

```
channel a0, b0, c0, a1, b1, c1, d0, d1       -- external channels
channel ask, ans, ping                       -- internal channels
channel bar                                  -- internal barrier

P0 = ask -> a0 -> ans -> b0 -> bar -> c0 -> P0
P1 = ask -> ans -> a1 -> b1 -> bar -> c1 -> ping -> P1
P2 = bar -> d0 -> ping -> bar -> d1 -> ping -> P2

P0_P1 = (P0 [| {ask, ans, bar} |] P1) \ {ask, ans}

Device = (P0_P1 [| {ping, bar} |] P2) \ {ping, bar}
```

This model first declares the events (channels and barrier) on which `Device` engages: 8 channels are external and 3, plus the barrier, are internal (conforming to Fig. 3). Three purely sequential processes follow. The remaining two processes put the first three together in parallel, *hiding* (using the operator "****", explained below) their internal events.

A process is defined by naming it, followed by an "`=`", followed by an expression defining its behaviour (which may be recursive). The "`->`" operator takes an *event* for its left operand and a *process* for its right operand and means: engage in the *event* (which requires synchronising with any processes running concurrently that have an interest in the event) and, then, behave as the *process*. The "`->`" operator is right associative, which means that a process like `a -> (b -> P)` can be written as `a -> b -> P`.

Process loops in occam-π code become tail recursion in CSP_M. The parallel operator in CSP_M, `[|` ***sync-set*** `|]`, is binary. Hence, the `Device` network (which has three sub-processes) is built in stages: two processes at a time. For an event in the ***sync-set*** of the parallel operator to occur, *both* process operands must engage. The hiding operator, "****", hides events in its *hide-set* operand (on its right) within its *process* operand (on its left), making those events invisible to the *environment* of that process. When combining two processes in parallel, we must hide those events (channels and barriers) that are used exclusively by those two processes. Hidden events of a process are distinct from same-named events used elsewhere, as though they were locally declared within the process (as they are in occam-π —see the implementation of `Device` in Sect. 5.2).

Note that CSP_M is a declarative (or functional) language, whereas occam-π is imperative. Students who love to program have no problem learning new syntax, so long as they understand why a different syntax is needed (occam-π is for building executables; CSP_M is for reasoning about them; they have the same concurrency semantics). Our students, at least, had no such problem. Informal rules for translating from occam-π to CSP_M may be found in the slides referenced in [WPB+11, WPBR11, WB11b]. Going in the other direction is much harder since many mechanisms within CSP (such as external choice over multi-way event synchronisations) have no direct counterpart in occam. Basic translation rules for CSP expressions that do have a direct counterpart in occam are given in [Ste03].

5.5. Formal analysis

Intentions: Learn about the semantics of CSP (traces, failures, divergences), simple model checking using FDR, the design of experiments for safety and liveness analysis (combining model checking with deductive reasoning).
Methodology: Formal model, assertion design and verification (Fig. 1)
Questions: Can we formally confirm our intuition of the initial event sequences, verify deadlock and livelock freedom, verify that particular bad behaviours cannot happen, verify that required (good) behaviour does happen?

[7] In Sects. 6.3 and 6.5.1, the processes need to be parametrised (and we say why).

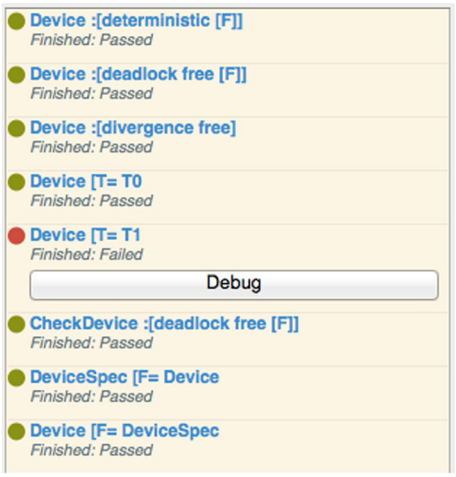


Fig. 5. Result of running FDR[9]

Having produced a formal model, which has one-to-one correspondence (both in structure and semantics) with the executable model, we can now utilise a model checker to answer questions like the ones posed towards the end of Sect. 5.3. Some questions can be answered immediately by the model checker without further thought (Sect. 5.5.1). Other questions require the construction of further models that have relevant and desired properties against which the original model may be compared (Sects. 5.5.2, 5.5.3 and 5.5.4).

Before continuing, we need to be more precise about what we mean by *traces*. A process may run forever, but a trace is always a *finite* sequence of events in which its process has engaged to some stage in its life. Infinite running processes will have an infinite set of (finite) traces. If `t` is a trace of a process, any prefix sub-trace of `t` will also be a trace of the process.[8] The empty trace, $\langle\,\rangle$, is a trace of every process and represents the initial state of a process, before any events have been engaged. Internal (i.e., hidden) events are not listed in the trace of a process – for example, events `ask`, `ans`, `ping` and `bar` will not appear in a trace of `Device`.

Now, the first and weakest form of semantic *refinement* (relating one process to another) can be defined:

> A process `P` *trace-refines* a process `Q` if and only if the traces of `P` are also traces of `Q`.

This also means that if there is some trace that `Q` *cannot* do, then `P` *cannot* do it either. If we consider `Q` as a specification, then `P` is *safe* in the sense that `P` cannot exhibit behaviour (presumably *'bad'*) that is disallowed by `Q`.

In CSP_M, `P` *trace-refines* `Q` is written: `Q [T= P`. *Trace refinement* is one of a range of semantic relations analysed by the FDR model-checker and we use it in Sect. 5.5.2. A second and stronger form, *failure-refinement*, is defined and used in Sect. 5.5.4.

5.5.1. Immediate results (deadlock and livelock freedom)

With the CSP model of `Device`, we can start asking questions. Loading it into the FDR GUI, we straight away discover it is free from deadlock and livelock (which CSP calls divergence), simply by clicking the buttons labelled to perform these checks. The first three lines with green discs in Fig. 5 establish that `Device` is *deterministic* (its behaviour can always be controlled by its environment) and free from deadlock and livelock. These checks are carried out with respect to the *failures* semantics (explained later, in Sect. 5.5.4) of CSP.

[8] The set of traces of a process is said to be *prefix-closed*.

[9] Processes `T0`, `T1`, `CheckDevice`, and `DeviceSpec` are defined in Sects. 5.5.2, 5.5.3, and 5.5.4, where the trace refinements referring to them in this figure are explained. The syntax of the refinement checks in assertions 4, 5, 7, and 8 of this figure seem back to front, but are explained at the end of Sect. 5.5.5

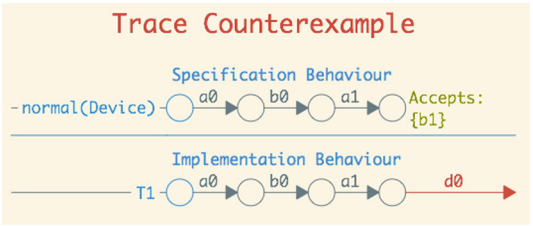


Fig. 6. FDR counter example

5.5.2. *Initial event sequences*

To confirm whether particular event sequences *may* initially be performed by `Device`, define processes that have no choice in the matter. For example

```
T0 = a0 -> b0 -> a1 -> b1 -> d0 -> c0 -> c1 -> STOP
T1 = a0 -> b0 -> a1 -> d0 -> b1 -> c0 -> c1 -> STOP
```

It is clear that the set of traces `T0` can perform is finite. We can easily list the whole set: { ⟨⟩, ⟨a0⟩, ⟨a0, b0⟩, . . . , ⟨a0, b0, a1, b1, d0, c0, c1⟩ } We can use the FDR model-checker to answer questions such as: are all these traces of `T0` also traces of `Device`? What about the traces of `T1`?

So, ask whether each of `T0` and `T1` *trace-refines* `Device`. FDR reports that `T0` does indeed do this (see the fourth line with a green disc in Fig. 5). This means, that any trace of `T0` can also be performed by `Device`. Clearly ⟨a0, b0, a1, b1, d0, c0, c1⟩ is a trace of `T0`. Therefore, it is also a trace of `Device`, which formally confirms one of the traces we discovered in our hand execution in Sect. 5.3. The remaining 17 initial traces could be verified in a similar manner. However, this would not prove that there were not any others.

FDR also reports that `T1` does *not* trace-refine `Device` (see the fifth line with a red disc in Fig. 5). This means that at least one of its traces cannot be performed by `Device`. FDR provides a counter-example: the trace ⟨a0, b0, a1, d0⟩ of `T1` (see Fig. 6). Since counter examples provided by FDR are minimal, we can conclude that the offending event is `d0`, which `Device` cannot perform as its fourth event in this case. This supports our intuition of the first 18 possible traces of the first 7 events (Fig. 4), none of which have `d0` as a possible fourth event.

5.5.3. *Safety (don't do bad things)*

Let us ask a more difficult question—this time about the behaviour of the continuously running system. Suppose the robot would do something very bad if its controller `Device` were ever to signal twice on `a0` without a signal on either `d0` or `d1` in between—an in-service failure. Might this happen?

Such questions can be answered simply by writing a process that monitors the signals from `Device`, looking for the bad scenario and deadlocks the system if spotted. For a programmer, this is just another function to write (though we need to know a bit more CSP_M):

```
Check (n) =
  if n >= 2 then
    STOP
  else
    a0 -> Check (n+1) [] d0 -> Check (0) []  d1 -> Check (0) []
    a1 -> Check (n) [] b0 -> Check (n) [] b1 -> Check (n) []
    c0 -> Check (n) [] c1 -> Check (n)
```

In the above, the operator `[]` (*external choice*) means: wait for one or more of its operand processes to be able to run, choose any *one* of them and run with it. The `[]` operator has lower precedence than the `->` operator. It is binary, right-associative (i.e., no parentheses are needed to bind more than two processes, as in the `else` clause of the definition of `check`) and commutative (i.e., its operands can be given in any order).

The parameter to Check records how many a0 signals have happened since the last d0 or d1. If this reaches 2, Check stops. Otherwise, if an a0 signal is taken, Check recurses with its parameter value increased by one. If a d0 or a d1 signal is taken, it recurses with its parameter cleared to zero. If any other signal is taken, it recurses with the parameter value unchanged.

A new process CheckDevice runs Device concurrently with Check (0), synchronising on *all* the external events on which Device engages. We start Check at zero since, initially, there have been zero a0 signals since the last d0 or d1:

```
CheckDevice = Device [| {a0, a1, b0, b1, c0, c1, d0, d1} |] Check (0)
```

If Check reaches its STOP process, Device cannot perform any of its external signals (since Check is refusing all of them). We know that Device is deadlock and livelock free. Therefore, it will always be *trying* to engage in external signalling. This leaves CheckDevice deadlocked (since Check and Device must synchronise on all signals from Device).

FDR quickly confirms that CheckDevice is free from deadlock (see the fifth green ball in Fig. 5). Therefore, Check can never reach its STOP. This formally verifies that the feared in-service problem cannot happen.

Any bad behaviour that can be described unambiguously can be programmed as a checker process in a similar manner. The important rules are that the checker must engage with *all* the external signals of the suspect device, and must stop engaging if the bad behaviour is detected. For any such process, the above reasoning applies for verifying the absence of the bad behaviour.

Protocol checking monitors, such as Check, are sometimes used live (e.g., in device drivers) to ensure correctness at run-time. It is important to note that we are using Check purely for static analysis—it has no role at run-time and, therefore, no impact on performance.

5.5.4. Liveness (do good things)

So far, our checks have concerned safety—namely that our system will not do incorrect things. This is not enough! After all, the STOP process does not do incorrect things—it does nothing; it has just one trace, namely ⟨ ⟩, a trace owned by all processes; thus STOP trace-refines every process (but is not a terribly useful implementation of anything). Trace-refinement is not enough.

More strongly, we need to consider *liveness*—namely that our system *will* do the right thing in all circumstances. For this, we need to check that our system *failure-refines* a specification of all those right things.

A process *state* is what a process has become after executing one of its traces (and may be represented by that trace). An event is *external* to a process if other processes may engage on it. A state is *stable* if there is no *internal* (i.e., hidden) event on which it may engage. A stable *resolution* of a state is a stable state reached by zero or more internal events only. A *stable failure* of a process is a state paired with a set of external events on which a stable resolution of that state refuses to engage. For brevity, in the rest of this paper, we shall refer to a *stable failure* simply as a *failure*.

A *failure* is not, therefore, a *'bad'* property for a process to have: refusing to synchronise on some of the events its environment may be offering is quite normal and, indeed, necessary (e.g., a full buffer should refuse further input). Of course, this should not be overdone: refusing to synchronise on *all* events means deadlock.

We can now give the definition of a second, and stronger, form of *refinement*:

> A process P *failure-refines* a process Q if and only if P trace-refines Q and the failures of P are also failures of Q.

In CSP_M this is written: Q [F= P. This refinement means that if a ⟨*state, event-set*⟩ pair is *not* a failure of Q, it is *not* a failure of P either. Now, if Q is a specification, then P fulfils its *liveness* conditions: if the specification (Q) says that in this state you *will react* to one of these events (i.e., there is no failure here), the implementation (P) *will react* (i.e., there is no failure there either). Furthermore, that reaction will be safe because P can only do *traces* of Q (i.e., the reaction is sanctioned). Failure-refinement makes a powerful statement!

Building on our intuition from Sect. 5.3, we consider the following trace pattern as a candidate for describing all possible traces[10] of Device, where * means repetition:

[10] To get *all* the traces, the *prefix-closure* of the (infinite) set generated by the pattern should be taken.

$$(\langle a0 \rangle ; (\langle b0 \rangle \,|||\, \langle a1, b1 \rangle) ; (\langle c0 \rangle \,|||\, \langle c1 \rangle \,|||\, \langle d0 \rangle) ; \langle a0 \rangle ; (\langle b0 \rangle \,|||\, \langle a1, b1 \rangle) ; (\langle c0 \rangle \,|||\, \langle c1 \rangle \,|||\, \langle d1 \rangle))^*$$

The first line of this pattern was proposed at the end of our informal analysis (Sect. 5.3). We now conjecture, in the full pattern above, that `Device` has no other initial traces and that it repeats this initial pattern indefinitely, but with `d1` alternating with `d0` in successive cycles.

From the pattern, we can directly transcribe a *process* that explicitly generates the traces in the pattern (and only those traces):

```
DeviceSpec = a0 -> (b0 -> SKIP ||| a1 -> b1 -> SKIP); (c0 -> SKIP ||| c1 -> SKIP ||| d0 -> SKIP);
             a0 -> (b0 -> SKIP ||| a1 -> b1 -> SKIP); (c0 -> SKIP ||| c1 -> SKIP ||| d1 -> SKIP); DeviceSpec
```

`SKIP` (the process that does nothing except terminate) is used as a place holder above for the right hand operand of "`->`" when no action is needed. The "`;`" means sequential composition of its operand processes (first the left, then the right). The interleave operator, |||, is shorthand for the parallel operator with an empty ***sync-set*** . It means the event sequences of its operand processes may interleave freely.

The behaviour of `DeviceSpec` is the sequential composition of five processes (one on each line above). The first line performs `a0` and then interleaves the event `b0` anywhere in the sequence ⟨`a1`, `b1`⟩. The second line interleaves `c0`, `c1`, and `d0` in any order.

This generates the traces defined by the first line of the above pattern. Lines 3 and 4 do exactly the same as lines 1 and 2, except that `d0` is replaced with `d1`. Line 5 just recurses.

FDR immediately verifies that `Device` *failure-refines* `DeviceSpec` (line 7 in Fig. 5).[11] This is all we need. All traces performed by `Device` are allowed by `DeviceSpec`—so it is as *safe* as the latter. All failures reached by `Device` are allowed by `DeviceSpec`—so it is as *alive* as the latter.

The good thing is that the latter—`DeviceSpec`—has transparent behaviour: it explicitly offers events according to, and only according to, the given trace pattern (so we can see what it *can* do) and it always remains alive to offer those events (because it has no internal synchronisations, on which it might deadlock or livelock, and no `STOP`s).

Whilst our intuition (Sect. 5.3) indicated that the first two lines of `DeviceSpec` reflected the initial behaviour of `Device`, it was unclear whether the pattern repeated cleanly as its sub-processes started looping. Without this verification, we might be tempted to add another barrier (`bar`) synchronisation at the end of each loop in `P0` and `P1` and half-loop in `P2`. This would impose stricter control on the looping within the three sub-processes—ensuring that each loop or half-loop are lock-stepped together. The above *failure-refinement*, confirmed by FDR, shows that the required pattern does indeed repeat cleanly and, so, this overhead is not necessary.

The reverse refinement is also confirmed (line 8 in Fig. 5). This means that `Device` and `DeviceSpec` have *exactly the same* traces and failures.[12] This is not really needed, but nice!

`Device` was not implemented in the more sequential manner as `DeviceSpec` (with localised parallel interleaving) because of the three independent functions (weapons systems, vision processing, and motion stability) it had to perform. The sequential logic of `DeviceSpec` would not cleanly separate the logic for these three functions. Process-oriented design led to the three communicating sub-systems in the actual implementation, that separately reflect these three functions. The above model-checking verifies that `DeviceSpec` gives us all the patterns of synchronisation `Device` can and will perform, on demand from its environment.

Of course, it would be preferable if `DeviceSpec` had been a part of the original functional specification of `Device`—but for pedagogical reasons we wanted to start with something concrete that could be explored.

5.5.5. *Assertions in CSP*$_{\mathrm{M}}$

To generate the display shown in Fig. 5 we need to write the list of assertions in the CSP$_{\mathrm{M}}$ script. Each time the script is loaded into FDR these script assertions are displayed. These assertions can be verified (green disc) (or refuted – red disc) individually by clicking on them, or all together by choosing the "Run All" button. For verifications described previously in this section, the assertions are:

```
assert Device :[ deterministic [F] ]      -- Device is deterministic (in the failures model)
assert Device :[ deadlock free [F] ]      -- Device is deadlock free (in the failures model)
assert Device :[ livelock free ]          -- Device is livelock free (in the failures-divergences model)
assert Device [T= T0                      -- T0 trace-refines Device
```

[11] See Sect. 5.5.5.

[12] This means they are *equivalent* in the failures semantics of CSP.

```
assert Device [T= T1                       -- T1 trace-refines Device (this fails)
assert CheckDevice :[ deadlock free [F] ] -- CheckDevice is deadlock free (in the failures model)
assert DeviceSpec [F= Device               -- Device failure-refines DeviceSpec
assert Device [F= DeviceSpec               -- DeviceSpec failure-refines Device
```

The syntax is a little strange, so comments have been added following each assertion.

6. Case study: mutually assured destruction

Intentions: This case study deals with a more realistic system than the one presented in Sect. 5 and illustrates more deeply the methodology of concurrency and verification being presented. Only one behavioural property is discussed: deadlock. The system is designed, the assertion that it is deadlock-free made, and formal and executable models developed. Despite an apparently safe design, verification discovers a potential for deadlock. The case study investigates and corrects this and verifies the correction to be deadlock-free.

This example is developed from a (undergraduate, year 2) exam question recently set for the *Concurrency Design and Practice* course [Wel13b] at the University of Kent. The case study in Sect. 5 investigated the behaviour of part of a robotics control system, verifying that it was safe with respect to certain user-specified criteria and establishing precisely its responsiveness to all events in all states. This new study considers a different part of the control system that has simpler behavioural requirements, but for which deadlock in its implementation becomes an issue. This lets the students see how verification exposes the potential for deadlock in a first design (that is so rare that it does not show up in extensive soak-testing) and leads to a revised design that is verified to be free from deadlock.

This study exercises all parts of the concurrency and verification workflow shown in Fig. 1. A process-oriented design is constructed followed by executable and formal models. Assertions about its behaviour are constructed and verification performed. The assertion about deadlock freedom fails verification. Rather than immediately returning to fix the design and the models, students can be asked to design a test harness to perform a soak test; when this is run, deadlock is unlikely to appear in any reasonable time (perhaps years). However, with the knowledge from the model checker of how the deadlock can occur, it is simple to design a test (by adding a single delay in one process) that will trigger the deadlock immediately. This shows the students the necessity of verification since, without it, a designer might convince herself that deadlock could not happen, and the system becomes installed and fails in service after a few years with possibly catastrophic consequences ("unexpected behaviour" in Fig. 1). Students are then asked to modify the process-oriented design and its executable and formal models so that all assertions, including the one about deadlock, now pass verification.

6.1. The problem

Intentions: introduce and explain the process-oriented design for the case study.
Methodology: process-oriented design (Fig. 1).

The system in Fig. 7 requires behaviour that is common not only for control systems, but also for applications in artificial intelligence, e-commerce, model-checking and elsewhere. Two processes are given a problem to solve; we are satisfied with a solution to either one of them; whichever process solves its problem first kills the other and makes a report; the one that is killed also reports that fact.

In the `MADsystem` (*"Mutually Assured Destruction"*) of Fig. 7, the problem solvers are `monitor` processes responsible for dealing with external sensor data. Each monitor is connected to a channel delivering its sensor data (which must be processed at all times), a command channel on which task parameters are sent (from time to time) and a report channel on which to report its performance after each task. The monitors are always commanded to do tasks in parallel.

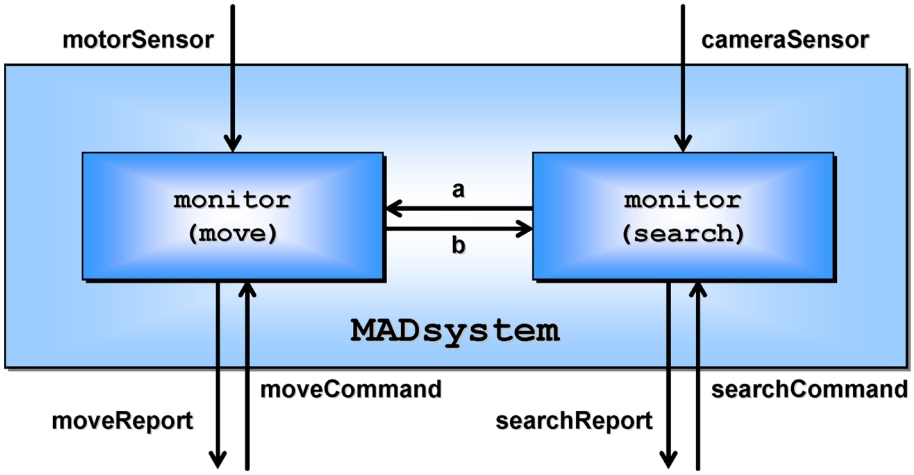


Fig. 7. Mutually Assured Destruction Sub-system

They are interconnected by two *kill* channels (labelled `a` and `b`), which the first process to finish uses to interrupt the other's work. When not engaged in a task, each monitor must continue to accept sensor input (which may be safely ignored) as it awaits its next command. The system should run indefinitely.

For simplicity in this presentation, the `monitor` processes in `MADsystem` are instances of the same template process, governed by a `mode` parameter that determines how sensor data is serviced when given a task.

In the left `monitor` of Fig. 7, the `move` parameter means that its task is to report back when the robot has moved a given `target` distance (given by its command channel). It does this by counting *clicks* on its sensor channel (the other end of which is connected to a sensor on one of its wheels).

In the right `monitor`, the `search` parameter means that its task is to report back when it detects a particular `target` feature[13] (given by its command channel) in an image delivered by its sensor channel (the other end of which is connected to a camera).

The purpose of this whole (sub-)system is to move the robot a prescribed maximum distance, stopping it early if some specified item of interest is discovered. This is achieved by two processes, one trying to achieve the set distance of movement and one looking for Martians. Whichever succeeds terminates the other *(Mutually Assured Destruction)*.

6.2. Executable model

> **Intentions:** present and explain the logic for the two components in `MADsystem`. This is expressed through a concrete implementation written in occam-π. Later, this design will be proven incorrect, though it looks very plausible at this stage and extensive testing will not reveal the error.
> **Methodology:** executable model (Fig. 1).

We use a mechanism of occam-π to define the kinds of messages carried by the the report channels:

```
PROTOCOL REPORT
  CASE
    me   -- monitor completed its task
    she  -- monitor was stopped in its task
:
```

Declaring a channel to carry this `REPORT` protocol (rather than, say, an `INT`) means that only tokens named `me` and `she` can be sent and received. We define the messages for the kill channels similarly:

[13] For a Mars rover vehicle, this could be a Martian of a particular colour.

```
PROTOCOL KILL
  CASE
    kill
:
```

Only one message token, `kill`, is defined—this will be extended later (Sect. 6.5.2).

The `monitor` process takes a `mode` parameter (determining which task it has to perform) and connects to five channels (see Fig. 7). For simplicity both the `command` and the `sensor` channels carry just integer values; these could, of course, be any data type pertinent to the sensor. The following code shows its main loop, whose body implements its *idling* state (i.e., waiting for a task command while consuming sensor data input):

```
PROC monitor (VAL INT mode, CHAN INT command?, CHAN INT sensor?, CHAN REPORT report!,
              CHAN KILL killYou!, killMe?)
  WHILE TRUE
    PRI ALT
      INT target:
      command ? target
        service (mode, target, sensor?, report!, killYou!, killMe?)
      INT x:
      sensor ? x
        SKIP
:
```

Note that priority is given to `command` messages over `sensor` messages. This is to guard against a highly active supplier of sensor data, so that a command to `service` the sensor data will always be taken the first time round the loop it appears (regardless of whether sensor data is pending). Such priority considerations are, however, irrelevant for deadlock analysis (which always has to allow for either choice to be made).

We show next a *template* for the `service` procedure. Details of local state and logic specific for the `mode` of operation required are not shown—place-holders for them are indicated by lines starting with three dots ("..."). Such details are not relevant for the synchronisation behaviour (and therefore deadlock potential) of `monitor`.

```
PROC service (VAL INT mode, target, CHAN INT sensor?, CHAN REPORT report!,  CHAN KILL killYou!, killMe?)
  ...  local state declarations and initialisation
  INITIAL BOOL running IS TRUE:
  WHILE running
    PRI ALT
      killMe ? kill
        SEQ
          report ! she
          running := FALSE
      INT x:
      sensor ? x
        SEQ
          ...  update local state with x  (depends on mode)
          IF
            ...  termination reached (depends on mode & target)
              SEQ
                killYou ! kill
                report ! me
                running := FALSE
            TRUE
              SKIP
:
```

Priority is given to the `killMe` channel over further processing of `sensor` data. This is for the same reason as before: to guard against a highly active sensor always supplying data, so that a `killMe` signal is always taken next if it comes. Again, we note that this prioritisation is irrelevant for deadlock analysis.

Finally we build the whole sub-system for Fig. 7:

```
VAL INT move IS 0:     -- mode values for
VAL INT search IS 1:   -- monitor operations

PROC MADsystem (CHAN INT moveCommand?, searchCommand?, CHAN INT motorSensor?, cameraSensor?,
                CHAN REPORT moveReport!, searchReport!)
  CHAN KILL a, b:
  PAR
    monitor (move, moveCommand?, motorSensor?, moveReport!, b!, a?)
    monitor (search, searchCommand?, cameraSensor?, searchReport!, a!, b?)
:
```

The opposite ordering of the `a` and `b` channel-ends in the instances of `monitor` reflects their different directions of use.

6.3. Formal model

> **Intentions:** Learn more CSP_M, in particular about abstracting away unimportant implementation detail for the formal model.
> **Methodology:** Executable and formal models (Fig. 1)
> **Questions:** What details from the executable model are not needed in the formal model?

CSP_M has its own mechanism for defining user-named tokens. The occam-π `REPORT` and `KILL` protocols translate to:

```
datatype REPORT = me | she
datatype KILL = kill
```

In Sect. 5.4, the formal CSP models for the `Device` processes (Fig. 3) were not parametrised, referring directly to globally defined connecting channels. This technique does not work for `MADsystem` (Fig. 7), since there are two instances of the same process (`monitor`) connected to different sets of channels.

For this formal model, therefore, we abstract the connecting channels for `monitor` into parameters. This forces its `service` procedure to be similarly parametrised. For consistency, the whole sub-system, `MADsystem`, is parametrised. In general, parametrising processes has many engineering benefits (e.g., for re-usability).

The CSP model of `monitor` closely follows the occam-π code, with the loop replaced by tail recursion. Because the particular details of the logic applied for analysing sensor data and deciding task completion are not relevant to the patterns of synchronisation performed by `monitor`, they are not included in the model. The `monitor` process, therefore, does not need a `mode` parameter. Further, only `sensor` and `command` *events* are relevant (i.e., the channels deliver data-less signals) – so, there is no `target` information to pass to the `service` procedure:

```
monitor (command, sensor, report,  killYou, killMe) =
  (command ->  service (sensor, report, killYou, killMe) [] sensor -> SKIP);
  monitor (command, sensor, report, killYou, killMe)
```

Note: CSP_M is a strongly typed formalism, but much of the typing is implicit. Parameter *types* are not explicitly declared and FDR deduces them either at compile-time (from the way the parameters are used) or during its model-checking. From the above, FDR infers that `command` and `sensor` are data-less channels, but has no information (yet) about its last three parameters.

One other note: CSP has no mechanism for *prioritised* choice. The `PRI ALT` from the occam-π `monitor` is modelled just by the (non-prioritised) external choice operator, `[]`. As explained in Sect. 6.2, priorities are not relevant for deadlock analysis.

Here is the `service` procedure (now bereft of `mode` and `target` parameters):

```
service (sensor, report, killYou, killMe) =
    killMe?kill -> report!she -> SKIP
    []
    sensor -> (killYou!kill -> report!me -> SKIP |~| service (sensor, report, killYou,  killMe))
```

FDR infers that the `report` parameter must be a channel carrying the `REPORT` data type and that `killYou` and `killMe` are channels carrying `KILL`. This inference transfers back to the same-named parameters of `monitor`.

This CSP `service` body follows the occam-π code, again modelling the loop with tail recursion. However, the presented model has been simplified and optimised from a *direct* model of the loop termination condition (which would require an extra parameter for the Boolean `running` value, an explicit test on it and its initialisation to `true`)[14]

The *internal* (non-deterministic) choice operator, `|~|`, used in the above `service` model has not been used in any of the earlier models presented in this paper and needs explaining. Suppose `P` and `Q` are processes. Then, `P|~|Q` is a process that behaves either as `P` or as `Q`. The choice is made internally—it cannot be influenced by the environment. If the environment offers only an event that `Q` will accept but `P` will not accept and if `P|~|Q` chooses to behave as `P`, then the environment and `P|~|Q` become deadlocked.

Internal choice is used in `service` to model the decision it makes, following receipt of `sensor` data, as to whether its task is complete. In the occam-π template (Sect. 6.2), we do not know the sensor data, how it is integrated with local state nor how the termination decision is made—and we do not care. All we care about is that a decision is made *either* to keep going *or* to kill the other `monitor` (and, then, report back and exit the service loop). This *do-not-care* is precisely modelled by `|~|` in the formal `service` code (following its acceptance of a `sensor` signal).

Finally, we model the `MADsystem`. Its internal channels are not parameters and have to be declared. In occam-π, these are declared local to the `PROC` body. In CSP_M, channels can only be declared globally. These internal channels are, of course, the *kill* channels and will be supplied as the last two parameters to the two instances to `monitor`. We have deduced earlier that these must carry `KILL` messages. In CSP_M, these are declared:

```
channel a, b : KILL
```

The `MADsystem` may now be constructed, remembering to hide these channels (including all events possible on them—that is, all possible messages):

```
MADsystem (moveCommand, searchCommand, motorSensor, cameraSensor, moveReport, searchReport) =
  ( monitor (moveCommand, motorSensor, moveReport, b, a) [| {| a, b |} |]
    monitor (searchCommand, cameraSensor, searchReport, a, b)
  ) \ {| a, b |}
```

Recall that the set-expression, `{| a,b |}` means the set of all possible events on channels `a` and `b`. Since these are `KILL` channels, only one message can (for now) be sent: `kill`. Hence, `{| a,b |}` is shorthand for the event set `{a.kill, b.kill}` where `a.kill` is the event representing the communication of `kill` over channel `a`.

In `MADsystem`, `{| a,b |}` is *both* the synchronisation set defined in the parallel operator (binding the two `monitor` processes together) *and* the set of events hidden from the environment within `MADsystem`—a common CSP idiom (also used for `Device` in Sect. 5.4).

6.4. Formal analysis

Intentions:

- to show the necessity of formal verification: the deadlock freedom assertion fails, which means there is a design error in the system.
- to learn how to use the model checker to find and understand an execution that leads to deadlock.
- to understand why testing did not reveal the deadlock.

Methodology: Formal model, assertion design and verification (Fig. 1)
Questions: Why did it deadlock and why did testing not show this?

[14] This directly translated model can be found on the supporting website [PW15], together with verification that it is equivalent to the one presented here.

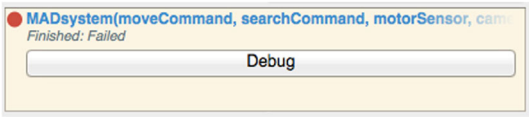


Fig. 8. Result of deadlock checking `MADsystem` – it's not!

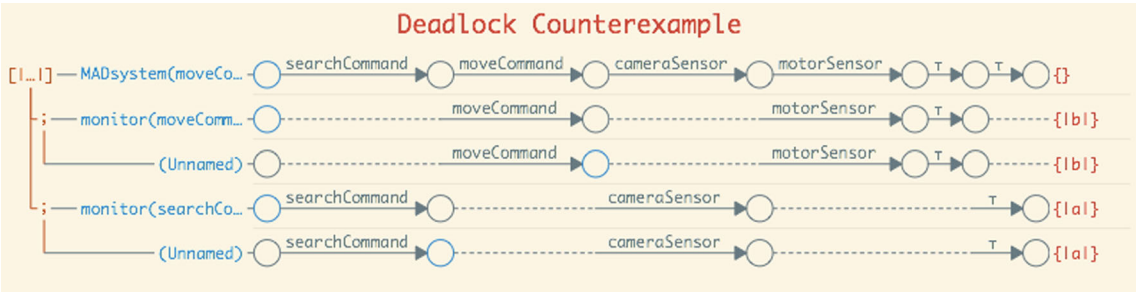


Fig. 9. Trace leading to a deadlock of `MADsystem`

We want to verify that `MADsystem` is deadlock free. To do this check, FDR requires a concrete instance of `MADsystem`. Since `MADsystem` takes parameters, actual channels for all six parameters must be supplied. Hence, we declare:

```
channel moveReport, searchReport : REPORT
channel motorSensor, cameraSensor
channel moveCommand, searchCommand
```

Now the assertion can be made:

```
assert MADsystem (moveCommand, searchCommand, motorSensor, cameraSensor, moveReport, searchReport)
 :[ deadlock free [F]]
```

When this script is loaded into FDR, this assertion appears in the rightmost window (see Fig. 8). Clicking "Run All" will verify all the assertions and produce a red disc , signifying that `MADsystem` is *not* deadlock free.

6.4.1. Tracking down the deadlock

Discovering that `MADsystem` has the potential for deadlock may be a shock. After all, the logic modelled by the `monitor` processes seems plausible and soak-testing (Sect. 6.4.2) of the occam-π executable model never triggered deadlock —*so this shock is extremely valuable*. Ignorance would eventually result in the loss of any machine controlled by the system, along with considerable time and money (in the case of a Mars rover) or lives (in the case of self-driving passenger cars).

In order to determine why the system deadlocked, simply clicking the "Debug" button brings up an FDR debugging window shown in Fig. 9. This shows that a deadlock has happened after the system has performed the trace:

$$\langle \texttt{searchCommand}, \texttt{moveCommand}, \texttt{cameraSensor}, \texttt{motorSensor}, \boldsymbol{\tau}, \boldsymbol{\tau} \rangle$$

where $\boldsymbol{\tau}$ represents a hidden internal event.[15]

By inspecting this trace (which is always a shortest that leads to a deadlock), we can track how the deadlock arose. FDR not only shows the interleaving of the entire system (`MADSystem`), but also the individual processes. Starting with the `monitor` process responsible for the motor control system (second line in Fig. 9), the list of events starts with `moveCommand` (which is the `command` parameter in the *move* `monitor` process—FDR names the

[15] Hidden events are not, formally, part of the trace. FDR provides this extra information in case it might assist our understanding of what has happened.

actual event, which here is the globally defined channel supplied as argument). This is followed by `motorSensor` (which is the argument given to the `sensor` parameter in the `service` procedure called by that `monitor`).

The *move* `monitor`, has reached an *unstable* state,[16] having just accepted data from its `sensor` within its `service` procedure. So, the hidden event can only be the resolution of the non-deterministic internal choice operator `|~|` (making the decision whether to continue with `service`). This hidden event is the last observed from the *move* `monitor` process. We can further deduce that this resolution was *not* in favour of the recursive call back to `service`, since in that case a further `sensor` (i.e., external `motorSensor`) event could have been taken and thus the system would not be in deadlock. Therefore, the resolution must have been in favour of offering the `killYou!kill` event (which is a write to the internal `b` channel—see Fig. 7 and the definitions in Sect. 6.3).

Exactly the same reasoning over the following `searchCommand`, `cameraSensor` and further hidden τ (fourth line in Fig. 9) shows that the *search* monitor also becomes blocked offering its `killYou!kill` event (which, in this case, is a write to the internal `a` channel).

We see the two `monitor` processes, having both satisfied their termination condition, trying to kill each other by sending on the channels `a` and `b`. Since neither process is offering to receive on those channels, no communication event can happen and the result is deadlock.

Naturally, knowing how this particular deadlock happens is not the end of this story. We still need to determine how to fix it both for this case and for every case—and to verify the fix. This is accomplished in Sect. 6.5.

6.4.2. Why did soak-testing not find the deadlock?

Correct operation of the `MADsystem` requires that both monitors are commanded to run their services together and that reports must be received back from them before further commands can be sent. Our *test-rig* honours this rule.

Define *service-end data* to be sensor data whose acceptance by a monitor would cause that monitor to complete its service. Define *kill-window* as the time period from when a monitor chooses to accept service-end data to when it offers its `killYou` signal.

Consider the *first* monitor to receive service-end data. No `killMe` can be pending (because it is the first), so it will enter its kill-window. If the other monitor receives service-end data *during this kill-window*, it will also enter its kill-window and the result will be deadlock—each monitor service trying to send a *kill* to the other and neither listening. Otherwise, the other monitor will still be in service at the end of the first monitor's kill-window, see the *kill* now on offer from the first monitor and take it in preference to any sensor data that may be available (including service-end data). In this case, both monitors will continue by making their separate reports and returning to their *idling* states—there will be no deadlock.

So, the deadlock reported by FDR (Sect. 6.4.1) happens if service-end data is taken by a monitor during a kill-window already started by the other. Whether we see this deadlock in our testing depends on the average length of kill-windows, the average interval between arrival of sensor data, the average number of sensor inputs taken to complete a service, the scheduling of processes on the hardware platform *and* how long we persist in the test.

Our test-rig consists of `MADsystem` (Sect. 6.2) in parallel with two *driver* processes simulating the two sensors (independently generating data for `MADsystem` at randomised intervals) and a *controller* process (generating commands to, and receiving reports from, `MADsystem` and recording progress statistics). This was running under the KRoC/CCSP [BWMW10] occam-π multicore scheduler [RSB12] on an Intel *i7* quad-core processor (2 GHz) with hyper-threading—i.e., there will be physically parallel execution. For this system, order of magnitude estimates for the quantities listed in the previous paragraph are 100 nanoseconds (for the kill-window), 10 milliseconds (for the average sensor data interval), 100 (for the average number of sensor inputs per service) and 1 day (for our persistence)—the last three being under our control.

With these figures, *service-end* data arrives at each monitor (on average) once per second. For each trial (an individual pair of services), deadlock requires *service-end* data arriving at the second monitor within the *kill-window* of the first. The chance of hitting that window is 100 nanoseconds out of 1,000,000,000—i.e., 1 in 10 million. With each trial averaging one second, we would need to wait towards 8 million seconds (over 90 days) to

[16] A state is *stable* if there is no internal (i.e., hidden) event on which it may engage—see Sect. 5.5.4.

reach a 50% chance of seeing that deadlock triggered. This analysis assumes that each process is statically locked to its own processor core (or virtual core)—i.e., that each process *will* run whenever it *can* run (because it does not have to compete for execution resource).[17]

If processes have to compete with each other to be run, scheduling factors reduce this chance of deadlock. The KRoC/CCSP kernel dynamically runs processes across all available cores, bundling and un-bundling them into batches for limited periods of software scheduling within individual cores. If the two *monitor* processes are in the same batch when one of them enters a *kill-window*, the other monitor will not be scheduled until that window ends and the deadlock will not arise. Given only five processes in the test-rig, this will happen quite often, reducing the chance of deadlock by an order of magnitude (and, possibly, more). Thus, for our hardware platform, these order-of-magnitude estimates suggest around 2 years of continuous testing would be needed for a 50% chance of discovering the deadlock. *If this system were ever put into live service (say, in an aeroplane) on the basis of having passed such a period of testing, we would not advise taking a flight.*

We note that verification immediately alerted us to the potential for deadlock in this system and showed us exactly how it could happen. There is no need, therefore, to test for its occurrence. This saves lots of time and money and, in the case of a negative result from testing, eventual disasters.

Of course, once aware of this potential for deadlock, we can adjust the parameters of our test-rig to expose it. The risk can be increased either by increasing the time for a *kill-window* or decreasing the time intervals between the arrival of *end-service* data. Inserting a delay of one second into the *kill-window* logic triggers the deadlock straight away. More interestingly, we fixed the driver data so that *end-service* data was generated *every* time and decreased the average interval between its delivery to 100 microseconds. Deadlock is now observed, but its time for occurrence varies considerably—from seconds to hours. We suspect there are other scheduling factors, further reducing the probability of hitting the deadlock scenario in these extreme conditions. Or, maybe, they are always present. Either way, this shows that the danger of relying on testing, with the target application parameters, is probably greater than that indicated by the above analysis.

6.5. Revised model

Intentions: To revise the design and models so it is deadlock free and formally verify this property.
Methodology: Process-oriented design, executable model, formal model, verification (Fig. 1)
Questions: How can the particular deadlock discovered by the model checker be eliminated? Does the solution introduce new deadlocks or effect otherwise correct behaviour? Are any other deadlocks possible?

One solution to the particular circumstances of this deadlock is[18] for the `service` procedure (within the `monitor` process) to read from its `killMe` channel (which is the `killYou` channel in the other `monitor`) in parallel with writing to its `killYou` channel (which is the `killMe` channel in the other `monitor`). Now, if both monitors commit to kill each other, both kill signals will be taken and there will not be deadlock.

In most circumstances, however, only one of the `monitor` processes is trying to kill the other. If we make no other change, the system will deadlock in that majority of cases—because the process being killed is not trying to kill the killer and so the killer's parallel *read-write* cannot complete. This is addressed by making the process receiving the kill message reply with an *acknowledgment*. This means the *kill* channels need to be able to carry two kinds of messages: a `kill` and an `ack`.

In all circumstances, now, interaction between the two monitor processes is always a *pair* of communications: either a `kill` in both directions or a `kill` in one direction followed by an `ack` in the other. We claim that programming the killer to write and read in parallel, and the killed to respond with an `ack`, prevents deadlock.

There is an added bonus with this arrangement. The killer process, by inspecting the message it receives from the process it is trying to kill, knows what happened in that other process. If it gets an `ack`, it knows the other process did not reach its termination condition and was stopped. If it gets a `kill`, it knows the other process did

[17] For up to 32 processes, such a platform is the XMOS XCore XS1-G4 [XMO13, Wik13].
[18] Another solution, initially attractive, is considered in Sect. 7.2.2.

succeed in its task. A process that gets killed (and is not trying to kill its partner) knows that the task of the other process has finished and, of course, that its own task has not. Therefore, each process, at the end of its `service` procedure, knows about the success status of both processes.

6.5.1. Revised formal model

The following shows the changes that need to be made to implement these revisions in the formal model. First, we need to extend the data types defining the kinds of messages carried by the kill and report channels. The `REPORT` has a new tag, `both`, for signalling that both processes reached their targets. The `KILL` has a new tag, `ack`, for acknowledgements:

```
datatype REPORT = me | she | both
datatype KILL = kill | ack
```

Only the `service` procedure needs changing:

```
service (sensor, report, killYou, killMe) =
  ( killMe?kill -> killYou!ack -> report!she -> SKIP
    []
    sensor ->
      ( ( killYou!kill -> SKIP
          |||
          ( killMe?ack -> report!me -> SKIP      -- my kill was acknowledged: I finished alone
            []
            killMe?kill -> report!both -> SKIP   -- she was killing me: we both finished
          )
        )
        |~|
        service (sensor, report, killYou, killMe)
      )
  )
```

In the above, the parallel write to `killYou` and read from `killMe` are implemented with the *interleaving* operator `|||` since they have no synchronisations in common. Two possible messages may now arrive on `killMe`. Reading and accepting either of them (and responding suitably) is implemented through an external choice `[]`.

No changes are needed anywhere else (i.e., the `monitor` and `MADsystem` processes). Verifying this new model with FDR gives a green ball to the assertion that previously failed (see Fig. 8), which assures us that the revised model is deadlock free.

6.5.2. Revised executable model

We now just have to transcribe the CSP back to occam-π. The two protocols are extended with the new tokens:

```
PROTOCOL REPORT
  CASE
    me          -- I finished
    she         -- she finished
    both        -- both of us finished
:

PROTOCOL KILL
  CASE
    kill
    ack         -- acknowledge kill
:
```

The revised `service` template becomes:

```
PROC service (VAL INT target, CHAN INT sensor?, CHAN REPORT report!, CHAN KILL killYou!, killMe?)
  ...  local state declarations and initialisation
  INITIAL BOOL running IS TRUE:
  WHILE running
```

```
    PRI ALT
      killMe ? kill
        SEQ
          killYou ! ack       -- acknowledge kill
          report ! she        -- she finished
          running := FALSE
      INT x:
      sensor ? x
        SEQ
          ...  update local state with x (depends on mode)
          IF
            ...  termination reached (depends on mode & target)
              SEQ
                PAR           -- send and receive in parallel
                  killYou ! kill
                  killMe ? CASE
                    ack       -- my kill was acknowledged: I finished alone
                      report ! me
                    kill      -- she was killing me: we both finished
                      report ! both
                running := FALSE
            TRUE
              SKIP
:
```

In Sect. 6.5.1, the read from `killMe` was modelled by an external choice that depended on the value received on the channel. In occam-π, the value being received on a channel cannot be specified as part of the channel input guard of an `ALT`—only the arrival of *any* input value can be a guard. Instead, occam-π provides a `CASE` mechanism (similar to a `switch` statement in Java or C) for selecting between whatever values are received on the channel.

7. Reflection

Intentions: from questions arising out of the two case studies, generate discussion concerning key issues for concurrency and verification as well as the workflow methodology.

We believe concurrency is fundamental to most aspects of computer science. It can and should be taught at the beginning *at the same time as and a necessary and natural complement to* sequential programming. The compositional nature of CSP (whose underlying mathematics is burnt into the languages and tools supporting process orientation) enables complex systems to be built, verified and maintained through the power of parallel design, without the usual fears of concurrency. On top of this, the parallel usage rules of occam-π,[19] combined with its strict controls on *name aliasing*, eliminates data race hazards from its systems. Our skills and intuition about serial programming remain valid, preserved intact within the concurrency semantics.

The aim of our teaching is to show students how implementation and verification of systems can be done concurrently and why it should be done concurrently. We have introduced a workflow methodology (Sect. 2 and Fig. 1) that describes relationships between overall system design, executable and formal models of the design, the development of assertions about the design, verification of those assertions, system testing and maintenance. The workflow allows many routes through this network, some of which may be done in parallel. For example, assertions about the behaviour of the system may be made before, during or after the construction of executable or formal models. Those models may be constructed in either order but, as the case studies illustrate, the constructions may proceed iteratively, synchronising at various stages of development as mistakes are found and corrected or new capabilities added.

The verifications presented in Sects. 5.5, 6.4 and 6.5 require some care and creativity in asking the right questions. However, this activity is very close to creative programming (e.g. Sect. 5.5.3) and can naturally be

[19] Other languages (e.g. Go [Rob12]) and libraries (e.g. JCSP [WBM+07]) supporting process-oriented design and implementing CSP mechanisms are, of course, available—though greater self-discipline is needed to abide by the rules.

taught and absorbed at the same time—anyone with a talent for programming also has a talent for verifying their programs. We claim that the demands on computer systems now, and certainly in the coming decades, require that programming and verification be practiced together. This view is echoed by other projects—for example Microsoft's Dafny programming and verification language [RL10] (which focuses on sequential issues). The essential simplicity and richness of CSP, engineered into languages like occam-π and future derivatives, make consideration of concurrency issues also possible today. It is time to change the culture and make a start.

We note that we are not making claims for *absolute* verification here. The verifications in this paper relate only to the logic and algorithms provided by the formal models. They do not, of course, guarantee accurate transcription back into the programming language; nor that the compiler correctly compiles down to machine-code, the scheduler correctly schedules the processes, the processor cores correctly implement the machine instructions *etc.*. They also do not guarantee that we have asked *all* the right questions when doing our verification.

Testing, therefore, remains necessary. What verification achieves is the raising of software reliability through the discovery and elimination of many design, formal modelling, and programming errors *before* testing starts. Some of these errors will be so subtle, rare and irreproducible that they would never be found through testing, only to show up in the field with catastrophic consequences (e.g. Sect. 6.4). Those that testing would have found will save the time and costs of those tests. Other errors that testing reveals will reflect verification assertions that had not been asked, but which can then be asked and against which the "fixes"can be checked. We do not intend to let the best (i.e., absolute verification) be the enemy of the good.

Building a formal model of a real system for use with model checkers inevitably requires abstracting away details. The example in the first study abstracted away from data values entirely. The example in the second case study has processes with inner loops whose exits depend on received data values. However, for this case the data values were able to be abstracted away (since they were not relevant to its problem with deadlock) and the loop exits modeled by non-deterministic internal choice. Many systems will have data dependencies that cannot be abstracted away—for example, the Ben-Ari study in the appendix. Some dependencies will be critical since the computational algorithms produce values that steer processes into synchronisation patterns that avoid deadlock (though this was not the case in our second case study). For such critical dependencies, if those data values are abstracted away, the formal model will show potential for deadlock. In such *false positive* cases, the abstraction has been taken too far and the formal model will need to capture the data values and algorithms computing them that prevent the deadlock. However, those data values may still need to be abstracted so that they range over rather small subsets and the engineer may need to learn the dark arts of model checking *compression* [RGG+95, WB11b]; otherwise, the state complexity within the formal model will quickly be beyond the capability of the model checker to analyse. Such skills are necessary and important to acquire for engineers designing and building complex concurrent systems, but are beyond the scope of the aims of this paper.

7.1. On the robot control system

Questions:

- How does synchronisation control the ordering of events?
- How does synchronisation enable the safe communication of information?
- How can safety issues (e.g. that the system cannot perform a sequence of actions identified in the specification as dangerous) be formally expressed and verified?
- How can liveness issues (e.g. that the system will respond to particular events with correct patterns of behaviour) be formally expressed and verified?

The case study in Sect. 5 was developed from one first worked through in a single lesson of a class on concurrency at the University of Nevada, Las Vegas in the spring of 2010. The class had previously studied a range of concurrency approaches, including process-oriented design material from the University of Kent's *"Concurrency Design and Practice"* course.

The students were comfortable with using process-oriented programming techniques in non-trivial projects, so this example system would be considered fairly simple. Nevertheless, it was appreciated that relying just on intuitive understanding is unsafe—especially if the application were safety critical.

During the exercise, students were given an overview (through examples) of the syntax of CSP_M, with the semantics of its operators defined by relating them to occam-π syntax and semantics. The functional nature of CSP_M, as opposed to the imperative nature of occam-π, was no particular obstacle.[20] The students tried their own test sequences of signals from `Device` and correctly obtained confirmation or rejection. Writing specific checking processes for long-term dangers (like `Check`) was harder, but they warmed to this after more practice with CSP_M.

What-ifs could be explored, and answers verified, without running any code. For example: can the bad behaviour described in Sect. 5.5.3 happen if the `ping` events were removed? (Yes). Do the `a0` and `a1` signals strictly alternate? (Yes). Do the `b0` and `b1` signals strictly alternate? (No).

Writing suitable specifications and working on failures refinement was beyond the scope of this exercise.

7.2. On mutually assured destruction

Questions:

- Why is testing especially inadequate for concurrent systems?
- Are there alternative solutions to the deadlock problem in this case study? If so, how do they compare with the one in Sect. 6.5?
- What benefits arise from synchronous communications as opposed to asynchronous ones–or vice versa?

The study in Sect. 6 presents an alarming state of affairs that may be representative of numerous mission and/or safety-critical systems in service today. A software controller, with seemingly logical design architecture and with correct implementation of that design, passes lengthy test trials and is put into service. After several years of faultless operation, it goes very wrong (deadlocking, in the studied case) and the machine in which it was embedded crashes, resulting in expensive mission failure (e.g. of a Mars rover vehicle) or large loss of life (e.g. of aeroplane passengers and crew).

The system is sufficiently small to be abstracted into a class exercise, homework or exam question. We show how straightforward modelling in CSP immediately exposes the design error, through the failure of a standard verification assertion for deadlock freedom. A correction to the error is proposed (that does not affect the computational tasks being performed) and verified through the model-checker now confirming the assertion. For the detection and correction of this bug, no testing was needed nor performed—only verification. The ideas behind this correction now form the basis of a *design pattern* that we teach for the safe management of server cross-communication. Discussions on a wide range of patterns for concurrent programming may be found in [Sam10, DT13, Cha16].

7.2.1. Testing and verification

Analysis of the original design error shows its chance of occurring is at most 1 in 100 million for each pair of monitor service cycles (averaging 1 s)—that is, around 2 years of continuous testing would be required to reach a 50% chance of triggering the deadlock. Curiosity testing of that design under extreme (i.e., unreal) operational parameters now shows up the error—but we needed to know the error in order to *'fix'* those parameters.

Confidence testing of the verified design, still needed because of the arguments about *absolute* verification in the introduction to Sect. 7, shows no deadlock under the extreme conditions that failed the original design. We have run it for 4G (over 4 thousand million) cycles (100 microseconds)—over 4 days – before deciding our confidence was high enough to give our test machine a rest.

[20] It is just programming!

7.2.2. An alternative solution to the deadlock

One student asked a very intelligent question: what would happen if *asynchronous* communication underlay the design? Surely, the original design would not deadlock in the discovered circumstances since the monitor service procedures would not block when they issue their *kill* signals, even though neither is listening?

The communication primitives of CSP and occam-π are *synchronous*. We have argued that this provides information that is very useful to the sending process: *if its message has been sent, it knows its message has been received*. Is such benefit cancelled by the increased danger of deadlock from communications that may block?

For this case study, the answer is no. Certainly, *immediate* deadlock would be avoided by asynchronous communications . . . but only at the expense of something that may well be worse: continued operation in corrupt state. If both monitor service routines enter a *kill-window*, both will send their *kills*, report success and return to their idle states. However, on their next service cycles, those *kills* will still be lurking in buffers and will now be taken—incorrectly aborting those services. To prevent this, a fix could be to maintain service sequence numbers and include them in the *kill* signals. Now, *kills* received with lower than the current sequence number could be discarded.

However, if the buffers supporting asynchronous communication were finite, there is still deadlock potential (though with greatly reduced probability, which may not be a good thing). There is no problem modelling buffered communications in CSP and a quick model-check—see the supporting website [PW15]—discovers the problem: a sequence of monitor services with both sides issuing kills, and neither taking them, will fill the buffers. Unless there is a policy of discarding oldest data in communication buffers or somehow providing infinite capacity, there will be deadlock. We know of no current concurrency model with the former policy and those that attempt to provide the latter open many other dangers for embedded systems (e.g. from runaway processes that waste all the memory with junk messages).

7.2.3. Cost of the verified fix and further wins

The verified protocol (Sect. 6.5) for terminating co-services securely, even in the presence of *mutual kills*, is simple (once known), lightweight, needs only a very small amount of extra memory (compared with the incorrect protocol) and has no need for sequence numbers. Further, because of the nature of synchronised communications, each service knows exactly what happened in the other when it ends: *either* it finished and the other did not, *or* it did not finish and the other did, *or* both finished. If more information were needed about the state of each process, this could be *piggy-backed* on top of the *kill* and *ack* signals with no extra synchronisation at run-time (or for model-checking) needed.

For some services, having this extra knowledge about co-services, may be crucial and it comes simply from synchronous channels. Of course, the same information (and security against deadlock) could be obtained from asynchronous channels, but only by copying the verified synchronous protocol. This introduces an explicit acknowledgement of each *kill* signal, which discards the semantic purpose of its asynchronous nature. So, the run-time overheads for management of the shared buffers supporting the asynchronous *kills* (not forgetting the asynchronous *acks*) are a waste of time. Those overheads are considerable compared with those for synchronous communications. Thus, it is not only simpler to reason using the latter, it also leads to faster running code.

7.3. Final thoughts

CSP and occam-π enable concurrency to be used to simplify complex system design. The occam-π run-time system imposes memory overheads of no more than 32 bytes per process and run-time overheads for synchronisation of the order of tens of nanoseconds on shared memory multicore systems. Small memory/power platforms and large scale complex system modelling (millions of processes) are addressed. It teams well with CSP to provide rich and flexible analysis. We have made proposals for verification abstractions and assertions to be introduced into the occam-π language, linking its compiler with the FDR model checker, so that only one syntactic representation is needed [WPB+11]. Examples of these verification abstractions for occam-π can be found in the CSP script for the first case study on the supporting web page for this paper [PW15].

There are many lessons about concurrency design from the two case studies presented in this paper. A key observation is that real *verification* of the behaviour of communicating processes is achieved, even though we have

engaged in only simple reasoning ourselves. The status of positive judgments from FDR is formal proof that the assertions are true (although purists may descend to complaints about the lack of formal proof of the correctness of the FDR implementation, the C++ compiler with which it was compiled, the computer hardware on which it was run, etc.). Those complaints apart, such verifications are a significant step forward in eliminating potentially catastrophic errors in concurrent systems and generally raising confidence in them. Yet all we feel we have done is program! This brings concurrency verification into the realm of students and everyday programming practice. Further reading may be found in [WP10].

Finally, can we teach students (those who love to program, anyway) concurrency so that they quickly develop a correct and intuitive understanding of the primitive mechanisms (e.g. processes, communication, synchronisation, networks) and higher level patterns (e.g. client-server, server cross-communication, phased barrier, I/O-PAR, . . .)? Can they use those primitives and patterns with the same fluency as they use serial computing primitives, without tripping over any dark hazards? Can they develop their own patterns when the standard ones do not apply? Can they use formal methods to verify good behaviour (e.g. freedom from deadlock and livelock, safety, liveness), without training in the underlying mathematics (process algebra, denotational semantics)? Can they do this as normal everyday practice, without any sense of fear? Yes, both we and they can.

This paper presents the ideas of learning about concurrent programming and formal verification at the same time. The case studies show that this mutually benefits both activities. However, this is only a foundation for developing these ideas and skills further. Currently, when considering more complex systems, care must be taken since there are areas of CSP that are not directly of efficiently implementable in occam (or any other language). For example, external choice over multiway synchronisation events (barriers) is easy to specify in CSP but is either unsupported in programming languages or, at best, inefficiently supported and with restrictions (such as shared-memory multicore only). Care must therefore be taken to limit the use of some capabilities of CSP in order to keep the iteration between executable and formal models simple.

We invite readers of this paper who use the ideas and materials in their teaching to contact us with a view to gathering data on their students' experiences on learning and applying these ideas and techniques, especially if these experiences can be contrasted to other concurrency models.

Acknowledgements

This work is part sponsored by the CoSMoS project (EPSRC Grants EP/E049419/1 and EP/E053505/1). It could not have been started nor completed without the skills, support and enthusiasm of all our colleagues on the CoSMoS project [SWT+07], the occam-π and KRoC development teams [BWMW10, SRJ+10], and the Software Engineering Institute (Carnegie Mellon University), who hosted and sponsored a sabbatical visit from one of the authors (PHW) at the start of this work. We owe special thanks to Fred Barnes, Martin Ellis, Carl Ritson, John Simpson, David Wood, Michael Goldsmith, Linda Northrop and Kurt Wallnau for assisting and enabling the development and presentation of our concurrency courses at Kent, CMU and UNLV. We acknowledge some giants of the field on whose shoulders we gratefully stand: Tony Hoare and Bill Roscoe (CSP), Robin Milner (π-calculus) and David May (classical occam). We would also like to thank generations of students at Kent and, more recently, at CMU and UNLV for choosing to attend our courses and providing exceptional and constructive feedback. Finally, we are extremely grateful to the diligent reviewers for their insight and many detailed suggestions that helped us so much in improving the paper.

Appendix A. Ben-Ari's twin process conundrum

In [BA10], Ben-Ari gives a seemingly simple program with 2 processes P and Q both updating a global variable n. Each process reads n, increments it locally, and writes it back; this is repeated 10 times. The pseudo-code presented in [BA10] is:

```
integer n = 0;

process P
integer regP = 0;
do 10 times
  load n into regP
  increment regP
  store regP into n
end

process Q
integer regQ = 0;
do 10 times
  load n into regQ
  increment regQ
  store regQ into n
end
```

He writes that for years, he taught students that the value of n at the end would be between 10 and 20, and was surprised when a student claimed to have gotten the value 9. He continues to explain that any number between 2 and 20 is possible.

A.1. A CSP model of the twin processes

This statement is easily verified using CSP, even without implementing an executable model at all. Let us go through a CSP model line by line:

```
channel load, store : {0..20}
channel kill
```

The first line declares a `load` and a `store` channel, bounding the values it may carry to between 0 and 20. Model checkers cannot deal with potentially unbounded numbers! These channels are used to communicate data between a process (`Var`, described below) representing the *shared* variable and the processes (`P`) incrementing it. Next, a `kill` channel is declared, just so that the process holding the value of n can be neatly terminated.

```
inc (x) = if x >= 20 then 20 else x + 1
```

This declares an increment function that, at first, looks a little strange. As just mentioned, FDR would have problems verifying assertions if integer values inspected during that verification are unbounded. We know that the value of n is never bigger than 20, so implementing `inc` in this way allows FDR to place an upper limit of 20 on n. If the argument to `inc`, x, is less than 20, 1 is added to x and returned.

```
P = ; x:<0..9> @
      load ? n -> store ! inc (n) -> SKIP
```

This declares the function that in [BA10] is referred to as P and Q. Note that `; x:<0..9> @ T` means repeat process `T` 10 times, replacing free `xs` with successive integers from the given range in successive stages of the sequence. In this case, process `P` just repeats its *load-increment-store* 10 times.

```
Var (n) =
  store ? x -> Var (x)
  []
  load ! n -> Var (n)
  []
  kill -> SKIP   -- terminate
```

The above process, `Var`, models the "global" variable holding n. It alternates between reading the `store` channel, writing to the `load` channel or accepting the `kill` signal. If the process accepts the `kill`, it terminates. Otherwise it simply recurses with the appropriate value of n: this is the value from the `store` channel if that was read, or n (i.e., unchanged) is the `load` channel was written.

```
PP_check =
  (P ||| P);
  load ? n ->
    if n == 2 then STOP
    else kill -> SKIP
```

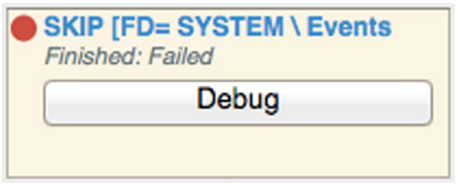


Fig. 10. Result of checking termination of `System`—it does not!

`PP_check` runs two processes in sequence. First, two copies of the process `P` are executed in parallel, interleaving freely over their use of the `load` and `store` channels. Second, a final `load` is performed to check if the result is equal to 2. If it is, this process `STOP`s and, as we will see, this will prevent any system of which it is a part from terminating. If it is not equal to 2, the process kills the `Var` process and terminates cleanly.

```
System =
  PP_check
  [| {| load, store, kill |} |]
  Var (0)
```

The above sets up the entire system by running `PP_check` concurrently with the `Var` process (which holds the value of n, initialised to 0). Of course, they have to synchronise on their use of the `load`, `store` and `kill` channels.

A.2. Analysis of the model

```
assert SKIP [FD= System \ Events
```

This asserts that `System` always terminates: technically it asserts that `System`, with all events hidden, is a *failures-divergence* refinement of the process `SKIP`, which does nothing but terminate and cannot refuse to terminate. Thus, if the assertion were true, `System` could also not refuse to terminate (i.e., it *would* terminate).[21]

But if 2 is a possible result for the value of the shared variable after the two `P` processes have finished, there is an execution path through `System` where `PP_check` will `STOP`, preventing `System` from terminating and the assertion will fail.

Running this model with FDR, the assertion does indeed fail (see Fig. 10). Along with its report of failure of the assertion, FDR offers a *Debug* option: clicking this button presents the trace of events FDR found that led to the failure. This is shown in Table 1, which documents the event sequence leading to a final value of 2.

Similarly, if we wanted to know an interleaving that would cause n to end with the value 9, all we need to do is replace the 2 (in `PP_check`) with 9 and rerun FDR. The same can be done for all values between 2 and 20 inclusive. However, if the number checked for is 1, the assertion succeeds which means that 1 is *not* a possible end result for the shared variable. These results verify Ben Ari's statements reported near the start of this Appendix.

However, might there be some other reason preventing termination, other than `PP_check` executing `STOP` because of some particular final value for the shared variable? The answer is no. Consider:

```
PP_no_check = (P ||| P); kill -> SKIP
```

This just runs the two `P` processes and then kills the `Var` process without checking its final value. If we plug this one into `System` (instead of `PP_check`), then the assertion of termination succeeds. We may conclude that failure to terminate is only because of the check made by `PP_check`.

These examples illustrate the power of being able to explore "what ifs" without actually observing executions that performed a bad execution. Many bad executions will not be exhibited by exhaustive testing as we have illustrated, but in this example, the bad behaviour was found and explained by the model checker immediately.

[21] The authors are grateful to Michael Goldsmith for this insight.

Table 1. Trace leading to n=2

Time	P	Q	n
0		load 0	0
1	load 0		0
2	store 1		1
3	load 1		1
4	store 2		2
5	load 2		2
6	store 3		3
7	load 3		3
8	store 4		4
9	load 4		4
10	store 5		5
11	load 5		5
12	store 6		6
13	load 6		6
14	store 7		7
15	load 7		7
16	store 8		8
17	load 8		8
18	store 9		9
19		store 1	1
20		load 1	1
21	load 1		1
22		store 2	2
23		load 2	2
24		store 3	3
25		load 3	3
26		store 4	4
27		load 4	4
28		store 5	5
29		load 5	5
30		store 6	6
31		load 6	6
32		store 7	7
33		load 7	7
34		store 8	8
35		load 8	8
36		store 9	9
37		load 9	9
38		store 10	10
39	store 2		2

A.3. Correcting the behaviour of the twin processes

The original Ben-Ari example was to show problems arising from uncontrolled access by concurrent processes to shared data. Even he was surprised by the range of results that could happen!

We end this exercise by adding control through a classical *mutex* and, on a positive note, verify that the system then behaves correctly—always!

```
channel wait, signal

Mutex =
  wait -> signal -> Mutex
  []
  kill -> SKIP
```

This `Mutex` process monitors `wait` and `signal` events, strictly switching between allowing first one and then the other. It also accepts a `kill`, allowing it to be shut down (in the same way as the `Var` process is shut down).

```
P' = ; x:<0..9> @
       wait ->
       load ? n -> store ! inc (n) ->
       signal -> SKIP
```

The P' process is the same as the previous P but with its *load-increment-store* operations sandwiched between a wait and a signal. Running this concurrently with Mutex ensures that those operations cannot be interleaved with those of any other instance of P' that might be also be running. So, running P' ensures that there will be 10 clean increments to the number held in Var. Running two copies means there will be 20 clean increments and the final answer held will be 20. Verifying this only needs a small change to the checking process:

```
PP_check' =
  (P' ||| P');
  load ? n ->
    if n != 20 then STOP
    else kill -> SKIP
```

This time, the check fails to terminate if the final result is anything but 20. Putting everything together:

```
SYSTEM' =
  PPcheck'
  [| {| load, store, kill |} |]
  Var (0)
```

we must not forget to add the mutex:

```
SAFE_SYSTEM =
  SYSTEM'
  [| {| wait, signal, kill |} |]
  Mutex
```

Now, the assertion becomes:

```
assert SKIP [FD= SAFE_SYSTEM \ Events
```

FDR verifies this assertion is true. This means that 20 is the only result possible in Var from running the twin processes, P' ||| P', and that the system, therefore, always behaves in the manner expected.

occam-π bindings for this corrected CSP model, and for the unsafe version from Sect. A.1 of this Appendix, are provided at the supporting website for this paper [PW15]. These (compilable and) executable versions have timing parameters that provoke the bad behaviour of the later but not, of course, the former.

References

[ACM12] ACM/IEEE-CS Joint Task Force for Computing Curricula. Computer science curricula 2013, Ironman Draft (Version 0.8), November 2012. http://ai.stanford.edu/users/sahami/CS2013/. Accessed 01 Aug 2013

[BA10] Ben-Ari M (2010) A primer on model checking. ACM Inroads 1(1):40–47

[Bar95] Barrett G (1995) Model checking in practice: the T9000 virtual channel processor. IEEE Trans Softw Eng 21(2):69–78. https://doi.org/10.1109/32.345823. Accessed 11 Dec 2017

[Bar05] Barnes FRM (2005) RMoX: an occam-π operating-system, January 2005. http://www.frmb.org/rmox.html. Accessed 1 Dec 2017

[Bar06] Barnes Frederick RM (2006) Compiling CSP. In: Welch PH, Kerridge J, Barnes FRM (eds) Communicating process architectures 2006, vol 64, WoTUG-29 of concurrent systems engineering series. IOS Press, Amsterdam, pp 377–388. ISBN: 1-58603-671-8

[BKPS97] Buth B, Kouvaras M, Peleska J, Shi H (1997) Deadlock analysis for a fault-tolerant system. In: Proceedings of the 6th international conference on algebraic methodology and software technology (AMAST97), pp 60–75

[BPS99] Buth B, Peleska J, Shi H (1999) Combining methods for the livelock analysis of a fault-tolerant system. In: Proceedings of the 7th international conference on algebraic methodology and software technology (AMAST98), pp 124–139

[BR10] Barnes FRM, Ritson CG (2010) Checking process-oriented operating system behaviour using csp and refinement. ACM SIGOPS Oper Syst Rev 43(4):45–49

[Bro08] Brown Neil CC (2008) Communicating Haskell processes: composable explicit concurrency using monads. In: Welch PH, Stepney S, Polack FAC, Barnes FRM, McEwan AA, Stiles GS, Broenink JF, Sampson AT (eds) Communicating process architectures 2008, vol 66 of Concurrent systems engineering. WoTUG, IOS Press, Amsterdam, pp 67–83.

[Bro10a] Brown NCC (2010) C++CSP home page. Programming languages and systems research group, University of Kent. http://www.cs.kent.ac.uk/projects/ofa/c++csp/. Accessed 11 Dec 2017

[Bro10b] Brown NCC (2010) Communicating Haskell processes home page. Programming languages and systems research group, University of Kent. http://www.cs.kent.ac.uk/projects/ofa/chp/. Accessed 11 Dec 2017

[BW03] Brown NCC, Welch PH (2003) An introduction to the Kent C++CSP library. In: Broenink JF, Hilderink GH (eds) Communicating process architectures 2003, WoTUG-26, Concurrent systems engineering, ISSN 1383-7575. IOS Press, Amsterdam, pp 139–156. ISBN: 1-58603-381-6

[BW04] Barnes FRM, Welch PH (2004) Communicating mobile processes. In: East I, Martin J, Welch PH, Duce D, Green M (eds) Communicating process architectures 2004, vol 62, WoTUG-27 of Concurrent systems engineering series, ISSN 1383-7575. IOS Press, Amsterdam, pp 201–218. ISBN: 1-58603-458-8

[BWMW10] Barnes FRM, Welch PH, Moores J, Wood DC (2010) The KRoC home page. Programming languages and systems research group, University of Kent. http://www.cs.kent.ac.uk/projects/ofa/kroc/. Accessed 11 Dec 2017

[Cha16] Chalmers K (2016) Communicating process architectures in light of parallel design patterns and skeletons. In: Communicating process architectures 2015. Open Channel Publishing Ltd., pp 227–244. ISBN: 978-0-9565409-9-7

[Don94] Dongarra J (1994) MPI: a message passing interface standard. Int J Supercomput High Perform Comput 8:165–184

[DT13] Danelutto M, Torquati M (2013) A RISC building block set for structured parallel programming. In: 21st euromicro international conference on parallel, distributed, and network-based processing, PDP 2013, Belfast, United Kingdom, February 27–March 1, 2013, pp 46–50

[GRABR14] Gibson-Robinson T, Armstrong P, Boulgakov A, Roscoe AW (2014) FDR3—a modern refinement checker for CSP. In: Tools and algorithms for the construction and analysis of systems 2014, vol LNCS 8413. Springer, Berlin

[GRABR16] Gibson-Robinson T, Armstrong P, Boulgakov A, Roscoe AW (2016) Failures divergences refinement (FDR) version 4

[GRS93] Goldsmith MH, Roscoe AW, Scott BGO (1993) Denotational semantics for occam2 (part 1). Transp Commun 1(2):65–91. Wiley, New York.

[GRS94] Goldsmith MH, Roscoe AW, Scott BGO (1994) Denotational semantics for occam2 (part 2). Transp Commun 2(1):25–67. Wiley, New York.

[HC02] Hall A, Chapman R (2002) Correctness by construction: developing a commercial secure system. IEEE Softw 19(1):18–25. https://doi.org/10.1109/52.976937. Accessed 11 Dec 2017

[Hoa85] Hoare CAR (1985) Communicating sequential processes. Prentice-Hall, Englewood Cliffs

[Hol03] Holzmann G (2003) The spin model checker: primer and reference manual. Addison-Wesley Professional, Reading

[JBV03] Jacobsen CL, Barnes FRM, Vinter B (2003) RMoX: a raw-metal occam experiment. In: Broenink JF, Hilderink GH (eds) Communicating process architectures 2003. IOS Press, Amsterdam, pp 269–288

[JJ04] Jacobsen CL, Jadud MC (2004) The transterpreter: a transputer interpreter. In: East IR, Duce D, Green M, Martin JMR, Welch PH (eds) Communicating process architectures 2004, vol 62, WoTUG-27 of Concurrent systems engineering series. IOS Press, Amsterdam, pp 99–106

[Low96] Lowe Gavin (1996) Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In: Tools and Algorithms for the Construction and Analysis of Systems, pages 147–166. Springer-Verlag

[Mil99] Milner R (1999) Communicating and mobile systems: the π-calculus. Cambridge University Press, Cambridge, ISBN-10: 0521658691, ISBN-13: 9780521658690

[MS07] McEwan AA, Schneider S (2007) Modeling and analysis of the AMBA bus using CSP and B. In: McEwan AA (ed) Schneider S, Ifill W, Welch PH (eds) Communicating process architectures 2007, vol 65 WoTUG-30 of Concurrent systems engineering series. WoTUG, IOS Press, Amsterdam, pp 379 –398

[PW15] Pedersen JB, Welch PH (2017) The symbiosis of concurrency and formal verification: teaching and case studies. On-line Support Material. http://www.santaclausproblem.net/verification/. Accessed 1 Dec 2017

[RGG+95] Roscoe AW, Gardiner PHB, Goldsmith MH, Hulance JR, Jackson DM, Scattergood JB (1995) Hierarchical compression for model-checking CSP, or How to check 10^20 dining philosophers for deadlock. In: Tools and algorithms for the construction and analysis of systems, vol LNCS 1019. Springer, Berlin

[RL10] Rustan K, Leino M (2010) Dafny: an automatic program verifier for functional correctness. In: 16th international conference on logic for programming artificial intelligence and reasoning, vol LNCS 6355. Springer, Berlin, pp 348–370

[Rob12] Pike R (2012) Go concurrency patterns, slides 5–8. http://talks.golang.org/2012/concurrency.slide. Accessed 1 Dec 2017

[Ros97] Roscoe A (1997) The theory and practise of concurrency. Prentice-Hall, Englewood Cliffs

[Ros10] Roscoe AW (2010) Understanding concurrent systems. Springer, Berlin

[RSB12] Ritson CG, Sampson AT, Barnes FRM (2012) Multicore scheduling for lightweight communicating processes. Sci Comput Program 77(6):727–740

[RW07] Ritson CG, Welch PH (2007) A process-oriented architecture for complex system modelling. In: McEwan AA, Schneider S, Ifill W, Welch PH (eds) Communicating process architectures 2007, vol 65, WoTUG-30 of Concurrent systems engineering series. IOS Press, Amsterdam, pp 249–266. ISBN: 978-1-58603-767-3

[RW10] Ritson CG, Welch PH (2010) A process-oriented architecture for complex system modelling. Concurr Comput Pract Exp 22:965–980

[Sam07] Sampson AT (2007) Compiling occam to C with Tock—CPA 2007 Fringe. Systems Research Group, University of Kent. http://www.wotug.org/paperdb/send_file.php?num=217. Accessed 11 Dec 2017

[Sam08] Sampson AT (2008) Two-way protocols for occam-π. In: Welch PH, Stepney S, Polack FAC, Barnes FRM, McEwan AA, Stiles GS, Broenink JF, Sampson AT (eds) Communicating process architectures 2008, vol 66, WoTUG-31 of Concurrent systems engineering series. IOS Press, Amsterdam, pp 85–97

[Sam10] Sampson Adam T (2010) Process-oriented patterns for concurrent software engineering. PhD thesis, University of Kent, October 2010. http://offog.org/publications/ats-thesis.pdf. Accessed 11 Dec 2017

[SBR+10] Sampson AT, Brown NCC, Ritson CG, Jacobsen CL, Jadud MC, Simpson J (2010) Tock (translator from occam to C from Kent) home page. Systems Research Group, University of Kent, http://projects.cs.kent.ac.uk/projects/tock/trac/. Accessed 11 Dec 2017

[SD04] Schneider Steve, Delicata Rob (2004) Verifying security protocols: an application of CSP. In: Abdallah Ali E, Jones Cliff B, Sanders Jeff W (eds) Communicating sequential processes. The first 25 years, volume LNCS 3525, pp 243–263. Springer, Berlin

[SGS95] SGS-THOMSON Microelectronics Limited (1995) occam 2.1 reference manual. Prentice-Hall, Englewood Cliffs

[SRJ+10] Sampson AT, Ritson CG, Jadud MC, Barnes FRM, Welch PH (2010) occam-π home page. Programming Languages and Systems Research Group, University of Kent. http://occam-pi.org/. Accessed 11 Dec 2017

[Ste03] Stepney S (2003) CSP/FDR2 to Handel-C translation. Technical Report YCS-2002-357, Department of Computer Science, University of York

[SWT+07] Stepney S., Welch PH, Timmis J, Alexander C, Barnes FRM, Bates M, Polack FAC, Tyrrell A (2007) CoSMoS: complex systems modelling and simulation infrastructure, April 2007. EPSRC grants EP/E053505/1 and EP/E049419/1. http://www.cosmos-research.org/. Accessed 11 Dec 2017

[WB05a] Welch Peter H, Barnes Frederick RM (2005) Communicating Mobile processes: introducing occam-π. In: Abdallah Ali E, Jones Cliff B, Sanders Jeff W (eds) 25 years of CSP, volume 3525 of Lecture notes in computer science, pp 175–210. Springer, Berlin

[WB05b] Welch Peter H, Barnes Frederick RM (2005) Mobile barriers for occam-π: semantics, implementation and application. In: Broenink Jan F, Roebbers Herman W, Sunter Johan PE, Welch Peter H Wood David C (eds) Communicating process architectures 2005, volume 63, WoTUG-28 of concurrent systems engineering series, pp 289–316. IOS Press, Amsterdam ISBN:1-58603-561-4

[WB08] Welch Peter H, Barnes Frederick RM (2008) A CSP model for mobile channels. In: Communicating process architectures 2008, volume 66, WoTUG-31 of concurrent systems engineering series, pp 17–33. IOS Press, Amsterdam. ISBN:978-1-58603-907-3

[WB11a] Welch P H, Brown N C C (2011) The JCSP (CSP for Java) Home Page, 2011. http://www.cs.kent.ac.uk/projects/ofa/jcsp/. Accessed 11 Dec 2017

[WB11b] Welch Peter H, Brown Neil CC (2011) Self-verifying dining philosophers. Presentation to IFIP Working Group 2.4, September 2011. https://www.cs.kent.ac.uk/research/groups/plas/wiki/IFIP_WG24. Accessed 11 Dec 2017

[WBM+07] Welch PH, Brown NCC, Moores J, Chalmers K, Sputh BHC (2007) Integrating and extending JCSP. In: McEwan Alistair A, Schneider S, Ifill W, Welch P (eds) Communicating process architectures 2007, volume 65 of concurrent systems engineering series, pp 349–370. IOS Press, Amsterdam. ISBN:978-1-58603-767-3

[WBM+10] Welch PH, Brown NCC, Moores J, Chalmers K, Sputh BHC (2010) Alting barriers: synchronisation with choice in Java using CSP. Concurr Comput Pract Exp 22:1049–1062

[Wel00] Welch Peter H (2000) Process oriented design for Java—concurrency for all. In: PDPTA 2000, vol 1, pp 51–57. CSREA Press. ISBN: 1-892512-52-1

[Wel13a] Welch Peter H (2013) Life of occam-Pi. In: Welch Peter H, Barnes Frederick RM, Broenink Jan F, Chalmers K, Pedersen Jan B, Sampson Adam T (eds) Communicating process architectures 2013, pp 293–318. Open Channel Publishing Ltd. ISBN:978-0-9565409-7-3. http://www.wotug.org/papers/CPA-2013/Welch13a/Welch13a.pdf. Accessed 11 Dec 2017

[Wel13b] Welch PH (2013) Concurrency design and practice, Course module. www.cs.kent.ac.uk/projects/ofa/sei-cmu/. Accessed 11 Dec 2017

[Wik13] Wikipedia (2013) XMOS XCore XS1. http://en.wikipedia.org/wiki/XCore. Accessed 11 Dec 2017

[WP10] Welch PH, Pedersen JB (2010) Santa Claus: formal analysis of a process-oriented solution. ACM Trans. Program. Lang. Syst. 32(4):37

[WPB+11] Welch Peter H, Pedersen Jan Baekgaard, Barnes Frederick R M, Ritson Carl G, Brown Neil CC (2011) Adding formal verification to occam-π. In: Communicating process architectures 2011, volume 68, WoTUG-33 of concurrent systems engineering series, pp 379–379. IOS Press, Amsterdam. ISBN:978-1-60750-773-4

[WPBR11] Welch Peter H, Pedersen Jan B, Barnes Frederick RM, Ritson Carl G (2011) Self-verifying concurrent programming. Presentation to IFIP Working Group 2.4, September 2011. https://www.cs.kent.ac.uk/research/groups/plas/wiki/IFIP_WG24. Accessed 11 Dec 2017

[WW96] Wood David C, Welch Peter H (1996) The kent retargetable occam compiler. In: O'Neill B (ed) Parallel processing developments, volume 47, WoTUG-19 of concurrent systems engineering series, pp 143–166, IOS Press, Amsterdam. ISBN:90-5199-261-0

[WWSK12] Welch PH, Wallnau K, Sampson AT, Klein M (2012) To boldly go: an occam-pi mission to engineer emergence. Nat Comput 11(3):449–474

[XMO13] XMOS Ltd. (2017) The xCORE difference. XMOS Ltd. http://www.xmos.com/products/silicon. Accessed 1 Dec 2017

Received 27 May 2016

Accepted in revised form 24 October 2017 by Jim Woodcock

Published online 20 December 2017