Santa Claus: Formal Analysis of a Process-Oriented Solution

PETER H. WELCH
University of Kent, Canterbury
and
JAN B. PEDERSEN
University of Nevada, Las Vegas

With the commercial development of multicore processors, the challenges of writing multi-threaded programs to take advantage of these new hardware architectures are becoming more and more pertinent. Concurrent programming is necessary to achieve the performance that the hardware offers. Traditional approaches present concurrency as an advanced topic: they have proven difficult to use, reason about with confidence, and scale up to high levels of concurrency. This paper reviews process oriented design, based on Hoare's algebra of Communicating Sequential Processes (CSP), and proposes that this approach to concurrency leads to solutions that are manageable by novice programmers - that is, they are easy to design and maintain, that they are scalable for complexity, obviously correct, and relatively easy to verify using formal reasoning and/or model checkers. These solutions can be developed in conventional programming languages (through CSP libraries) or specialised ones (such as occam-τ) in a manner that directly reflects their formal expression. Systems can be developed without needing specialist knowledge of the CSP formalism, since the supporting mathematics is burnt into the tools and languages supporting it. We illustrate these concepts with the Santa Claus Problem, which has been used as a challenge problem since 1994. We consider this problem as an example control system, producing external signals reporting changes of internal state (that model the external world). We claim our occam-τ solution is correct-by-design, but follow this up with formal verification (using the FDR model checker for CSP) that the system is free from deadlock and livelock, that the produced control signals obey crucial ordering constraints, and that the system has key liveness properties.

Categories and Subject Descriptors: D.1.3 [Concurrent Programming]: ; D.2.4 [Software / Program Verification]: Correctness Proof; D.2.11 [Software Architectures]: Languages; D.3.2 [Language Classification]: Concurrent, Distributed, and Parallel Languages; D.3.3 [Language Constructs and Features]: Concurrent Programming Structures; D.4.1 [Process Management]: Concurrency, Deadlocks, Multiprocessing, Scheduling, Synchronization; F.1.2 [Modes of Computation]: Alternation and Nondeterminism, Interactive and Reactive Computation, Parallelism and Concurrency; H.2.4 [Systems]: Concurrency

General Terms: Design, Languages, Reliability, Verification

Additional Key Words and Phrases: process orientation, concurrency, deadlock, event ordering, liveness, verification, novice programmer, occam-τ, CSP

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2009 ACM 0164-0925/2009/0500-0015 $5.00

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, 1 2009, Pages 1–35.
1. INTRODUCTION

1.1 Concurrency is our Friend

In the teaching of computer science, concurrency is commonly treated as an ‘advanced’ topic – only to be approached, if at all, once students have learned and become comfortable with sequential programming. In the practice of computer science, concurrency is commonly engaged with only as a last resort [Muller and Walrath 2000] – to deal with performance issues (such as response latencies in an embedded control system or the efficient use of a parallel supercomputer).

These mindsets have been formed by painful experience over five decades. Concurrency just causes too many surprises. In contrast with sequential logic, system state we thought we had under one thread of control gets hit by another – sometimes – and the result is chaos. So, we add protection in the form of locks and, should we get this wrong, the system deadlocks ... or livelocks ... or parts of it get starved of attention. Obviously, concurrent logic will be much harder than sequential stuff?

No. Concurrency is fundamental to the workings of the universe. It exists at all levels of granularity (e.g. nanoscale, human, astronomic). Complex, interesting and useful behaviour emerges from the concurrent actions of zillions of processes, each managing its own – and only its own – state, and synchronising and communicating to enable and/or constrain each others’ individual behaviours.

To provide useful service to its environment, a computer system needs to reflect that environment. That environment, being part of the natural world, is concurrent. The computer system, for simplicity therefore, needs to be concurrent.

A default restriction of programmed logic to sequential design and implementation is unnatural. A penalty is the increasing difficulty of managing complex behaviour. In an aerodynamically unstable airplane, we need to control both wings and the tail fin all at the same time! Programming the necessary logic in one thread of control is asking for trouble. Programming it as a network of Communicating Sequential Processes directly matches the problem structure and will be (much) easier. Richer complexity can be built through layers of network – just as in real life. Welcome to CSP.

The above thesis has lain around for nearly thirty years, which is about the time major ideas take to mature and be applied in computer science. Performance issues have (wrongly) been the driving force for concurrency to date. The commercial arrival of multicore processors – themselves forced on us by the laws of physics – brings that driving force into play immediately. On a quad-core processor, sequential code cannot use more than 25% of the processing power. If we need more, there has to be concurrently executable code - and lots of it. To be in with the chance of keeping the cores busy with useful work, there needs to be (an order of magnitude) more concurrency in the software than is available in the hardware – the principle of parallel slackness first discussed by Valiant [Valiant 1990].

There is another problem here. It is a suspicion that most existing multithreaded applications have concurrency errors. Many of these errors remain hidden because of favourable scheduling sequences on unicores processors. On multicores, with real parallel execution, any such errors have a greater risk of causing trouble. People are scared – but there is no avoiding it.

This paper reviews and demonstrates the use of the concurrency model outlined
above and inspired by Hoare’s CSP process algebra. The good news is that there
will be no maths here. The even better news is that there are elegant and powerful
mathematical properties (e.g. compositionality) underlying the model and built into
the languages, libraries and tools supporting it – we get the benefits simply by
using them. In addition to simplifications in application logic, safe and efficient
exploitation of multicore processors follows automatically. At least, all this is true for
the occam-π [Welch and Barnes 2005a; Barnes and Welch 2004; Welch and Barnes
2005b; 2008] multiprocessing language (used in this paper) and the JCSP [Welch
2000; Welch et al. 2007; Welch and Austin] and C++CSP [Brown and Welch 2003;
Brown 2007] libraries for Java/C++ respectively.

1.2 The Santa Claus Problem

For illustration, we consider a problem first suggested by John A. Trono in [Trono
1994]. This problem is known as The Santa Claus Problem. Trono’s description,
with a few added words, is as follows:

Santa repeatedly sleeps until wakened by either all of his nine reindeer (back from their holidays) or by a group of three of his ten elves (who have left their workbenches). If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go back on holiday and him to go back to sleep). If awakened by a group of elves, he shows each of the group into his study, consults with them on toy R&D and finally shows each of them out (allowing them to go back to work and him to sleep). Santa should give priority to the reindeer in the case that there is both a group of elves and a group of reindeer waiting. Initially, the reindeer are all on holiday, the elves are at their workbenches and Santa is asleep.

Although this problem seems simple at first, it presents challenges in concurrent
control that are typical of a wide range of computer application. Over the years,
solutions have been published demonstrating many approaches to concurrency (Sec-
tion 7.1). The first of these solutions (using semaphores [Trono 1994]) was shown to
be incorrect (by Ben-Ari et al. [Ben-Ari 1998]) and replaced by one using monitors.
The fact that the first published solution was wrong is evidence that the problem
is not, in fact, that simple.

We can imagine the changes of state in the Santa Claus system reflecting the
state of some real machine – for example, an airplane or nuclear power plant.
With the addition of output signals upon state change, the system might even be
controlling that machine. For many applications, that control will be safety critical
and verification of its safety (do no bad things) and effective functioning (do good
things) will be essential.

We extend the specification of the problem to include external reports from the
system, documenting what Santa, the elves and reindeer are doing as the system
evolves. We suppose that any machine it is controlling will break if those reports
occur in certain wrong orders (see Sections 5.5 and 6). We verify that none of these
can happen in our solution.

Concurrency tripwires such as race hazard, deadlock, livelock and process star-
vation must also, of course, be avoided. Particular safety constraints will demand
that certain state changes (and their corresponding signals) must happen in cer-
tain orders – that is, that Lamport *happens-before* relations [Lamport 1978] must
be enforced and verified. Liveness demands will specify that certain state changes
must happen following certain events. Building solutions to such problems that are
not only correct, but are seen to be correct, is a challenge for concurrent logic –
though not, we claim, as big as the challenge to solve it with purely serial logic.

1.3 Paper Roadmap

This paper presents a solution that is *process-oriented* (Section 2). Concurrent
design is expressed with process network diagrams (Figures 1-6) and directly refined
into executable *occam*-π code (Sections 3 and 4). *occam*-π is an extended version of
the classical *occam* [SGS-THOMSON Microelectronics Limited 1995] programming
language, incorporating dynamic network construction mechanisms (channel and
process mobility) from Milner’s π-calculus [Milner 1999]. Formal verification of a
range of correctness properties is achieved through direct mapping of the *occam-
π* source code to CSP and model checking using FDR [Formal Systems (Europe)
Ltd. 1998] (Sections 5 and 6). Switching between these representations (network
diagrams, *occam*-π code and CSP) is straightforward, allowing maintenance to be
led from any form – an example is given in Section 6.

The concurrency in our solution directly reflects the concurrency in the Santa
Claus story. This *simplifies* its design and implementation, rather than making
it hard. Indeed, aspects of this problem and solution have been set in formal
examination for second year computer science undergraduates (at the University of
Kent) and all passed. We claim that this design and development leads to solutions
that are ‘obviously correct’, but that backing this up with formal verification is not
hard and, probably, a good idea.

2. REVIEW OF PROCESS ORIENTED DESIGN

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 synchroni-
sation patterns. To be practical, a process oriented programming language, or a
library providing the necessary support for other languages, is essential – other-
wise, 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.

2.1 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.

2.2 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.

These communications differ from those in common message passing libraries for parallel computing. For example, in MPI [Dongarra 1994] 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. Individual processes are bound to a particular set of events (channels, barriers, ...) 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. A process engaging on a named barrier blocks until whatever other processes registered for the barrier also engage. Network connectivity is explicit and dynamic and constrained to what the system needs. The difference is subtle but helps engineer high cohesion within and zero coupling between processes.

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 – 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 of all threads through the shared objects. Reasoning is non-local: the logic of an individual thread cannot be devised, or understood, on its own. This is hard.

2.3 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 syn-
chronise 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.

2.4 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 cardinality) in response to their environment.

2.5 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 (channels-ends, barriers, ...) – its alphabet in CSP language. 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 any 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 Figure 1. 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 engage in channels (i.e., external I/O operations) and/or barriers. They are simple and familiar (except, possibly, for the barriers) and all our skills for sequential programming remain valid.

We make use of such diagrams all the time when designing \texttt{occam-\pi} programs. The design pictures for the Santa Claus system, for which we show an \texttt{occam-\pi} implementation and verify some correctness properties, appear later in this paper.

2.6 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 \texttt{occam-\pi} have a direct representation in CSP. FDR [Formal Systems (Europe) Ltd. 1998] 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 [Schneider and Delicata 2004; Barrett 1995; Hall and Chapman 2002; Buth et al. 1997; Buth et al. 1999; Lowe 1996; McEwan and Schneider 2007].

Translation between \texttt{occam} and CSP is defined, [Goldsmith et al. 1993; 1994], and can be automated. At present, we do this by hand and this paper gives an example. 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 several challenges that are not the subject of this paper. The Santa Claus system does not raise such problems and the translation is direct, preserving both syntactic and semantic structure.

It should be noted that \texttt{occam-\pi} is not designed to be an execution engine for CSP – that is, that translation from arbitrary CSP systems to \texttt{occam-\pi} is not always easy or, even, possible\footnote{Having said that, there is not much that cannot now be done.}. Rather, \texttt{occam-\pi} is designed as a programming language with concurrency built in as a first-class mechanism, with a semantics directly expressed by CSP. It allows concurrency to be used with the same confidence, ease and overheads as, say, sequential procedures (or method invocation).

3. A LANGUAGE BINDING FOR PROCESS ORIENTED DEVELOPMENT

\texttt{occam-\pi} is an imperative stateful language built around the concurrency model of Hoare’s CSP. Compiler enforced language rules prevent unsynchronised 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 [SGS-THOMSON Microelectronics Limited 1995] through the careful blending in of dynamic mechanisms from Milner’s \( \pi \)-calculus [Milner 1999] – for example, mobile channels, barriers and processes [Welch and Barnes 2005a; Barnes and Welch 2004; Welch and Barnes 2005b; 2008] These mobility concepts have much to offer in the modelling of the Santa Claus system, but will be considered in a later paper.

occam-\( \pi \) also extends classical occam through the introduction of shared channel-ends (modelled by CSP interleaving), barriers (corresponding to multiway CSP events) and extended rendezvous (simply modelled by adding an event marking the end of the rendezvous). These three mechanisms are employed to simplify the solution presented in this paper.

The occam-\( \pi \) codes were developed with the KRoc [Wood and Welch 1996; Welch et al.] 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 86 platforms (taking full advantage of multicores). 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 scheduled per processing node and perform useful work [Ritson and Welch 2007]. An interpreted version (the Transpreter [Jacobsen and Jadud 2004]) is available for almost any target platform, requiring a very tiny memory footprint. Two new compiler projects [Barnes 2006; Sampson 2007; Sampson et al.], targeting all platforms supported by a C compiler, are in development.

3.1 Processes, Sequential Composition and Parallel Composition

A process in occam-\( \pi \) 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.

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

\[
\begin{align*}
\text{SEQ} & \quad \text{PAR} \\
\ldots \ & \text{process A} \quad \ldots \ & \text{process A} \\
\ldots \ & \text{process B} \quad \ldots \ & \text{process B} \\
\ldots \ & \text{process C} \quad \ldots \ & \text{process C}
\end{align*}
\]

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. Note that any component may have its own locals.

3.2 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
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\(^2\).

Five other primitive processes are: channel input and output (Section 3.3), barrier
synchronisation (Section 3.6), \texttt{SKIP} (which does nothing but terminate – sometimes
needed for syntactic place holding), and \texttt{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 \textit{abstractions} may be named and parametrised:

\begin{verbatim}
PRC <name> (<parameters>)
\end{verbatim}

\begin{verbatim}
... process
\end{verbatim}

The parameters may be any type, including data (by reference or value) and
synchronisations (channel-ends and barriers). The colon above 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. In-
voked in \textit{sequence} with other processes, they may be thought of as procedures (or
methods). Invoked in \textit{parallel} with other processes, they become components of
a network whose topology is determined by the synchronisation items they share.
The main system developed in this paper (Section 4.3 and Figures 2 and 6) gives
an example.

3.3 Channels, Extended Rendezvous and Sharing

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

\begin{verbatim}
CHAN <message-type> <name>: 
\end{verbatim}

The reading end of a channel is denoted by \texttt{<name>?} and the writing end by
\texttt{<name>!}. To write to a channel:

\begin{verbatim}
c ! <expression-list>
\end{verbatim}

where the message type of the channel and individual expressions in the (semi-colon
separated) list must match. Channels are zero-buffered, so the writing process will
block until another process, running in parallel with it, executes a read on the other
end of the same channel:

\begin{verbatim}
c ? <variable-list>
\end{verbatim}

Here, the message type of the channel and 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.

\(^2\)This is not the case for most other imperative languages – such as C, Java, ...

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, 1 2009.
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 (concurrent) ways.

\textit{occam-π} allows an \textit{extended rendezvous} on channel input:

\begin{verbatim}
c ?? <variable>
...
\end{verbatim}

This extended input blocks until a message is pending on the \textit{c} channel. For a normal channel input (i.e., a \textit{single} ?), the writing process would be released at this point. With the above extended channel input (??), the release does not happen until after the indented rendezvous block has been completed by the reading process. The writing process is unaware of any such behaviour.

Either of the two ends, or both, can be shared: this is denoted by either \textit{SHARED} ? (shared reading end), \textit{SHARED} ! (shared writing end) or \textit{SHARED} (both ends shared) prefixed to the channel declaration. If a channel-end is \textit{SHARED}, it must be claimed for exclusive use:

\begin{verbatim}
CLAIM c!
...
\end{verbatim}

The \textit{CLAIM} will \textit{block} if some other process is already holding a claim on this channel-end. Processes wait on a FIFO queue – a different one for each end (if both are shared). A process has exclusive use of the channel-end within the indented block below the \textit{CLAIM}. The claim is automatically released at the end of this block.

It is fairly common that a \textit{CLAIM} block consists of a single line using the claimed channel:

\begin{verbatim}
CLAIM c!
c ! 42
\end{verbatim}

In such cases, the claim and use may be collapsed to a single line:

\begin{verbatim}
CLAIM c ! 42
\end{verbatim}

All uses of shared channel-ends in this paper fit this pattern.

\subsection{3.4 Choice}

\textit{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 \textit{arbitrary} (i.e., non-deterministic) choice is made. An \textit{ALTernative} construct is a list of \textit{guarded processes}:

\begin{verbatim}
ALT
  <guard>
  <process>
  ...
  <guard>
  <process>
\end{verbatim}

The list order does not matter. The guards are the \textit{waited-for} events – currently, only input processes (simple/extended, on offer when a message is pending), timeouts (on offer when expired) and \textit{SKIPS} (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) version 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.

Used with a SKIP as its final guard, a PRI ALT lets us poll channels for input and get on with something else if none is available. It also gives a way to force a process away from servicing an always-busy channel. The unfairness built in to the PRI ALT curiously, and simply, gives a way to guarantee fairness is servicing events. For example, the above multiplexor can be made fair by unrolling the loop body twice (in SEQUENCE), changing both ALTs into PRI ALTs and switching the order of the second one. It is also exactly suited to Santa’s duty to go with the reindeer in case both the reindeer and a party of elves are knocking on Santa’s door.

3.5 Replicators
The SEQ, PAR and ALT constructs may be replicated. Suppose XXX is one of these three keywords. Then:

```plaintext
XXX i = start FOR n
    <process.which.may.use.i>
```

is short-hand for:

```plaintext
XXX
    <process.with.i.replaced.by.start>
    <process.with.i.replaced.by.(start + 1)>
    ...
    <process.with.i.replaced.by.(start + (n - 1))>
```

The replicated SEQ corresponds to a traditional for-loop (with guaranteed termination). The replicated PAR sets up regular network topologies. In a replicated ALT, the `<process.which.may.use.i>` must be a guarded process and the construct waits for, and chooses between, an array of events.

3.6 Barriers
The last concept needed for this paper is the barrier. A barrier is multiway 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:

```plaintext
BARRIER <barrier-name>:

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

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

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

4. AN OCCAM-π IMPLEMENTATION
The Santa Claus system has three kinds of component: Santa, reindeer and elf. Their states are outlined in on-line Appendix A, along with the reports we require them to make as they cycle through them.
The Santa Claus story has near symmetry. The actions performed by reindeer and elves are much the same: they do things on their own for a while (holidaying, toy development), then rendezvous with Santa (9 out of 9, 3 out of 10) and work with him (delivering toys, consulting), then repeat. The partial nature of the Santa-elves rendezvous, however, requires special treatment compared with the full rendezvous between Santa and all the reindeer (which can be handled by a primitive CSP/occam-\pi multiway event). Symmetry could be restored by declining that primitive. Then, we would need only one type of process that could play the role of either a reindeer or elf – and Santa’s processing of either would be the same. However, in this presentation, we will work with separate logic and explore the different mechanisms.

4.1 Santa and the Reindeer

Figure 2 shows Santa and the reindeer processes connected in a network of channels and barriers.

There is a report channel, whose writing end is shared by all the processes and through which they report their life stories. This channel is external to the system and may be used to animate a display of what is happening inside the system. It could also be used to control a machine whose components are modelled by the Santa, reindeer and elf processes. It may be more appropriate to have separate (parallel) report channels – one for each component of the system. However, the semantics of the system are such that there is no difference between these two views. Each component makes its reports independently, identifying itself in each report. Those identifying tags partition the set of reports into component-unique subsets that can be viewed as reports on separate channels. For synchronisation analysis, all we need to know is that reports can be made at any time and will never be blocked by the actions (or inactions) of other components within the system. We may assume that the environment of the system will always accept these messages.

There is a just.reindeer barrier, on which the reindeer wait for each other to
return from holiday. There is a `santa.reindeer` barrier, on which the reindeer wait for Santa to harness all of them before starting the sleigh run. This is also used to wait for Santa to bring them all home after delivering toys. Finally, there is a `reindeer.2.santa` channel, whose writing end is shared by the reindeer. When all the reindeer are back from holiday, they report in to Santa (for harnessing) by sending their names through this channel – they will, of course, be blocked at this point if Santa is consulting with elves. The reindeer also use this channel to synchronise with Santa so that he can unharness them after delivering toys.

Here is the reindeer process:

```plaintext
PROC reindeer (VAL INT id, BARRIER just.reindeer, santa.reindeer, 
               SHARED CHAN INT to.santa!, SHARED CHAN REINDEER.MSG report!)
WHILE TRUE
  SEQ
    CLAIM report ! holiday; id -- "I'm on holiday" + id
    random.wait (HOLIDAY.TIME) -- sleep for a random amount of time
    CLAIM report ! deer.ready; id -- "I'm back from holiday" + id
    SYNC just.reindeer -- wait for all deer to return
    CLAIM to.santa ! id -- send id to Santa (to get harnessed)
    SYNC santa.reindeer -- wait for the others to get harnessed
    CLAIM report ! deliver; id -- "I'm delivering toys" + id
    SYNC santa.reindeer -- until Santa takes us all home
    CLAIM report ! deer.done; id -- "I'm back from the sleigh run" + id
    CLAIM to.santa ! id -- get unharnessed
```

Here is the part of the header and body of the Santa process that deals with the reindeer:

```plaintext
PROC santa (CHAN INT from.reindeer?, BARRIER santa.reindeer, 
... elf connections
              SHARED CHAN SANTA.MSG report!)
WHILE TRUE
  PRI ALT -- sleep until reindeer or elves show up
    ... deal with the reindeer
    ... deal with the elves
```

Each of the above `dealing` processes is prefixed by a signal indicating, respectively, that either all `N.REINDEER` (9) reindeer are back from holiday or that a group of `G.ELVES` (3) elves, out of `N.ELVES` (10), is ready to consult. The `PRI ALT` ensures that if there are both reindeer and elves signalling, Santa will choose the reindeer.

Here is Santa’s code dealing with the reindeer. The wake-up signal is just the message from one (any one) of the gathered reindeer, giving its name. This is followed by the messages from all the other reindeer – in some arbitrary order:

```plaintext
{{
  deal with the reindeer
  INT id:
  from.reindeer ? id -- the first reindeer is here
  SEQ
    CLAIM report ! reindeer.ready -- "Ho, Ho, Ho, the reindeer are here"
    CLAIM report ! harness; id -- "Harnessing reindeer " + id

```
SEQ i = 0 FOR N.REINDEER - 1 -- for the remaining deer
SEQ
  from.reindeer ? id          -- receive their id
  CLAIM report ! harness; id  -- "Harnessing reindeer " + id
 CLAIM report ! mush.mush     -- "Mush Mush"
 SYNC santa.reindeer          -- let the reindeer know all are harnessed
 random.wait (DELIVERY.TIME) -- deliver toys for some random time
 CLAIM report ! woah          -- "Whoa ... let's go home"
 SYNC santa.reindeer          -- signal everyone to return home
 SEQ i = 0 FOR N.REINDEER     -- for each deer
  from.reindeer ?? id         -- receive their id
  CLAIM report ! unharness; id -- "Unharnessing reindeer " + id
}}

The reindeer, once they have reported to Santa for unharnessing, loop around to announce they are on holiday again. Santa does the unharnessing and reports it, holding each reindeer in an extended rendezvous whilst that happens. This ensures that the unharnessing report happens before the holiday report. Note that this is not necessary for the harnessing reports, since the reindeer wait for the whole team to be harnessed (santa.reindeer) before their next report (that they are delivering toys).

There are other happens-before relationships enforced by the above santa and reindeer processes:

— all reindeer must have reported their return from holiday before Santa is woken up to announce that the reindeer are here — enforced by the SYNC on just.reindeer in the reindeer process;

— all reindeer must be harnessed before Santa says “Mush Mush” — enforced by sequential code in the santa process;

— Santa says “Mush Mush” before any reindeer reports it is delivering toys — enforced by the first SYNC on santa.reindeer in the reindeer and santa processes;

— Santa says “Whoa” before any reindeer reports it is back from delivering toys — enforced by the second SYNC on santa.reindeer in the reindeer and santa processes.

Later in this paper, we show how these relationships can be formally specified and verified.

4.2 Santa and the Elves

Neither CSP nor occam-π have primitives for the partial barrier synchronisation required for this part of the system. So, we have to model the ideas with special processes.

4.2.1 Partial Barriers. A partial barrier synchronises any $x$ out of $y$ (other) processes, where $0 < x \leq y$. It is managed by a simple protocol, comprising two channel communications per synchronising process to complete each partial barrier. The first makes the offer to synchronise and the second forces it to wait for enough offers to have been made. These communications are received and managed by a simple process that counts down the offers and, then, accepts the release signals:
(a) $n$ processes enrolled on a partial barrier (any process synchronising must wait for 2 others to synchronise).

(b) Implementation of a partial barrier (using current occam-$\pi$ mechanisms).

**Fig. 3. A partial barrier.**

PROC p.bar (VAL INT x, CHAN BOOL a?, b?)

WHILE TRUE

SEQ

SEQ i = 0 FOR x

BOOL any:

a ? any

SEQ i = 0 FOR x

BOOL any:

b ? any

The $x$ partial-barrier-synchronising processes plug into shared writing ends of the channels connecting to p.bar. To synchronise, they run:

PROC sync (SHARED CHAN BOOL a!, b!)

SEQ

CLAIM a ! TRUE -- offer to synchronise

CLAIM b ! TRUE -- wait for enough offers

This sync process completes when, and only when, it is one of $x$ processes (out of $y$) engaging in the partial barrier (i.e., plugged into p.bar).

This construction has an obvious visual representation. Diagram (a) in Figure 3 shows $n$ processes enrolled on a partial barrier requiring 3 synchronisation offers to complete. Diagram (b) shows the implementation given in this subsection.

4.2.2 Extended Partial Barriers. The p.bar alone is not sufficient to solve the elves waiting problem when they want to see Santa. If they sync as above on an p.bar set to 3, when three elves get through the barrier they will try to see Santa (who may be away with the reindeer). As soon as that group of elves are through the barrier, another group (out of the 7 left) can come along and also complete the barrier. There will now be two groups of three elves trying to see Santa – this is not allowed!

A common idiom with barriers is to schedule some special code to be run on completion of the barrier but before the processes engaging on the barrier are released. This can be achieved by generating a signal from the process managing the barrier to trigger that code: To do this, we extend p.bar to become the xp.bar process by adding a signal channel (CHAN BOOL ping!) to its parameter list and outputting
TRUE on ping! between its two replicated SEQs. The code implementing this is in on-line Appendix B.

Now, all that is needed is to wrap the code to be triggered in a process that waits for the ping signal and runs it. To ensure that this code completes before the triggering barrier, simply run it inside an extended rendezvous:

```plaintext
BOOL any:
  ping ?? any
  ... process triggered by barrier
```

Sometimes that condition is not necessary – all that is needed is that the special code is triggered. This is the case for our system.

Diagram (a) in Figure 4 visualises n processes enrolled on an extended partial barrier requiring 3 synchronisation offers to complete. Diagram (b) shows the implementation given in this subsection.

4.2.3 Elves and Santa. Figure 5 shows Santa and the elf processes connected in a network of channels and partial barriers.

There is the same report channel as shown in Figure 2. This is used by Santa and the elves for the same reasons as before. There is a just.elves extended partial barrier of size three, on which elves wait for two others to want to see Santa. When three are present, a (knock) on Santa’s door is generated, via the extension channel belonging to the barrier. When Santa accepts that knock, the barrier completes and the three elves can consult with Santa. This corresponds to the just.reindeer barrier from Figure 2.

There is a santa.elves (non-extended) partial barrier of size four. This is used by each group of three elves to wait for santa to greet all of them before the consultation starts. This is also used by the elves to wait for Santa to conclude the consultation. Only Santa and the current consulting group of elves will ever try to use this partial barrier. This corresponds to the santa.reindeer barrier from Figure 2.

There is an elves.2.santa channel, whose writing end is shared by the elves. When a group of three elves have assembled, they introduce themselves through this channel. They will not be blocked at this point, since Santa is expecting them (having been awoken by the knock on his door from xp.bar). This channel is also
used to say goodbye to Santa when the consultation is finished. Since occam-π does not (currently) support partial barrier connectors, the transformations defined by Figures 3 and 4 are applied to yield (the directly implementable) Figure 6.

In terms of Figure 6, here is the elf process:

```plaintext
PROC elf (VAL INT id, SHARED CHAN BOOL just.elves.a!, just.elves.b!,
          SHARED CHAN BOOL santa.elves.a!, santa.elves.b!,
          SHARED CHAN INT to.santa!, SHARED CHAN ELF.MSG report!)
WHILE TRUE
  SEQ
    CLAIM report ! working; id -- "I’m working" + id
    random.wait (WORKING.TIME) -- until I have a problem
```

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, 1 2009.
CLAIM report ! elf.ready; id -- "I want to consult with Santa" + id
sync (just.elves.a, just.elves.b) -- wait for two other elves
CLAIM to.santa ! id -- say hello to Santa
sync (santa.elves.a!, santa.elves.b!) -- wait for the other elves to say hello
CLAIM report ! consult; id -- "I'm consulting with Santa" + id
sync (santa.elves.a!, santa.elves.b!) -- until Santa has had enough
CLAIM report ! elf.done; id -- "I'm done consulting" + id
CLAIM to.santa ! id -- say goodbye to Santa

Completing the declaration of the Santa process header from Section 4.1, here are the channels for communicating with elves that were omitted:

CHAN BOOL just.elves.ping?, CHAN INT from.elves?,
SHARED CHAN BOOL santa.elves.a!, santa.elves.b!,

Recall that, for the elves, Santa is awoken by a ping from the extension channel of the partial barrier (and not by a communication from one of the elves):

{{{

deal with the elves
BOGL any:
    just.elves.ping ? any -- a party of elves is at the door
SEQ
    CLAIM report ! elves.ready -- "Ho, Ho, Ho, some elves are here"
    INT id:
    SEQ
        from.elves ? id -- receive his id
        CLAIM report ! greet; id -- "Hello elf " + id
        CLAIM report ! consulting -- "Consulting with elves"
    sync (santa.elves.a!, santa.elves.b!) -- let elves party know all are here
    random.wait(CONSULTATION.TIME) -- consult for a random time
    CLAIM report ! santa.done -- "Ok, all done - thanks!"
    sync (santa.elves.a!, santa.elves.b!) -- let elves know consultancy is over
    SEQ i = 0 FOR G.ELVES -- for each elf in the consult party
        INT id:
        from.elves ?? id -- receive his id
        CLAIM report ! goodbye; id -- "Goodbye elf " + id
    }}

Each elf is held (in an extended rendezvous) whilst Santa says goodbye for the same reason that Santa held each reindeer whilst unharnessing: so that the “goodbye” report happens before the elf can get back to work and report “working”. There are other happens-before relationships enforced by the above santa and elf processes:

—three elves must have reported that they want to see Santa before Santa is woken up and announces that the elf party is here — enforced by the extended partial barrier, just.elves, between the elf processes (moderated by the ping wake-up call to Santa);
— the elf party must be greeted before Santa says “Consulting with elves” — enforced by sequential code in the santa process;

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, 1 2009.
Santa Claus says “Consulting with elves” before any elf reports that it is consulting – enforced by the first partial barrier synchronisation on `santa.elves` in the `elf` and `santa` processes;

Santa says “all done” before any elf reports its consultancy session is over – enforced by the second partial barrier synchronisation on `santa.elves` in the `elf` and `santa` processes.

### 4.3 The Santa Claus System

The Santa Claus system consists of everything shown in Figures 2 and 6. The `report` channel is left open (an external parameter) with all other events (channels and barriers) hidden (local).

```san
PROC santa.system (SHARED CHAN MESSAGE report!)
  BARRIER just.reindeer, santa.reindeer:
  SHARED ! CHAN INT reindeer.2.santa:
  SHARED ! CHAN BOOL just.elves.a, just.elves.b: -- extended partial barrier
  CHAN BOOL just.elves.ping: -- channels for just.elves
  SHARED ! CHAN BOOL santa.elves.a, santa.elves.b: -- partial barrier channels
    -- for santa.elves
  SHARED ! CHAN INT elves.2.santa:
  PAR
    PAR ENROLL santa.reindeer
      santa (reindeer.2.santa?, santa.reindeer, just.elves.ping?, elves.2.santa?, santa.elves.a!, santa.elves.b!, report!)
    PAR i = 0 FOR N.REINDEER ENROLL just.reindeer, santa.reindeer
      reindeer (i, just.reindeer, santa.reindeer, reindeer.2.santa!, report!)
    PAR i = 0 FOR N.ELVES
      elf (i, just.elves.a!, just.elves.b!, santa.elves.a!, santa.elves.b!, elves.2.santa!, report!)
      xp.bar (G.ELVES, just.elves.a?, just.elves.b?, just.elves.ping!)
      p.bar (G.ELVES + 1, santa.elves.a?, santa.elves.b?)
```

The complexity of the implementation depends on the number of reports required and how strictly they must conform to `happens-before` relations. For instance, an elf has two waiting states it could report: a wait to get into the `front` group (or `waiting room`) that will be next to see Santa and, then, a wait to see Santa (who may be away delivering toys).

In the model built in this section, an elf only reports when it wants to see Santa. In Section 6.1, we extend the elf implementation to report both waiting states. We require three elf reports that they have made the front group `before` Santa reports he has been awakened by them. Initially, the arrival of the third elf in the front group automatically triggers a knock on Santa’s door and, if Santa is sleeping, his awakening. The third elf’s report (that it is in the front group) and Santa’s report (that he has been woken by elves) proceed in parallel and could happen in either order. The correct ordering is later enforced and verified, but extra synchronisations (and complexity) are needed.
5. VERIFYING PROPERTIES OF THE SOLUTION

The Santa Claus Problem can be viewed as an exercise in programming concurrent control logic for safety critical systems. The reliability of such a system is the most important issue. The software should be formally verified to assure that it is free from deadlock and livelock, free from race conditions, and that the ordering of control signals generated by the software does not violate any specified constraints.

A number of obstacles complicate formal verification of code. Firstly, a translation from the code into the formal model used for verification is necessary (or in the case where formal verification is performed before the implementation, a translation from the formal model to the programming language of choice). This is potentially a challenging task, especially if the language and the formalism have little in common. The greater the gap, the greater the risk of introducing bugs becomes. Secondly, today’s verification tools are rarely capable of checking a complete program without any user support.

Therefore, we believe that the best solution to producing correct (formally verified) code is by choosing a formalism and a language that are related and a verification tool that can assist in ‘computer driven verification’, that is, certain parts of the code can be verified, and the rest must be formally argued by the programmer.

The solution to the Santa Claus Problem presented in the previous section is expressed in the process oriented language occam-π, a language based on the formal process algebra CSP as well as the π calculus [Milner 1999]. A formalisation in CSP is extremely close to the actual occam-π code. This relationship is crucial in reducing the number of errors when translating between the language and the formalism or vice versa.

To verify the specification in CSP, we utilise the FDR [Formal Systems (Europe) Ltd. 1998] tool. FDR can check a number of properties about a CSP specification, for example: freedom from deadlock and livelock. It can also verify process refinement and equivalence under three semantic models of growing strength (traces, failures, and failures-divergences.)

5.1 The CSP Model

In this section, we translate the occam-π model of the Santa Claus system given in Section 4 into CSP. The CSP structure (syntax and semantics) directly matches the occam-π code and could, therefore, have been presented first.

However, occam-π does not pretend to be an implementation of CSP and translating in the other direction is harder. Some CSP features have large overheads (for the execution mechanisms currently known and in the occam-π run-time kernel) and are not yet supported – for example, external choice over arbitrary multiway events. General CSP systems can be implemented in occam-π, but they have to be transformed first into equivalent forms that do have direct representation [McEwan 2006]. This can sometimes be unpleasant.

occam-π is a programming language most of whose concurrency mechanisms have a direct model in CSP (and, therefore, semantics). We prefer, normally, to design within the patterns allowed by occam-π, knowing that our systems are directly and efficiently executable and that formal reasoning and/or model checking can be directly applied.
Some *occam*-π mechanisms are not addressed by CSP – such as timing and priorities. So, the random delays in the *occam*-π model (e.g. when a reindeer is on holiday) are treated as SKIPS in the CSP and the PRI ALT in the *santa* process is mapped to a plain external choice. Shared channel-ends are modelled by the sharing processes *interleaving* on them. The *occam*-π CLAIM mechanism is a refinement of this interleaving, since processes trying to use them are fairly *queued* (rather than having the *arbitrary* access allowed by interleaving).

The *occam*-π extended rendezvous is simply modelled in CSP by introducing an *acknowledgement* event for the channel communication. Both processes must engage on that acknowledgement following each communication – immediately by the sender but after the rendezvous block by the receiver.

We present the CSP using the machine-readable syntax, CSP-M, accepted by the FDR [Formal Systems (Europe) Ltd. 1998] model checking tool. The CSP constants, types and events used by the system are given in the on-line Appendix C. This appendix also contains the CSP definitions of the partial barrier processes (*P_BAR*, *XP_BAR*), the reindeer process and the whole Santa Claus network. Here, we present just the CSP version of the elf, partial barrier and Santa processes to demonstrate the relationship between *occam*-π executable code and CSP verifiable expressions.

5.2 Elves

We define the elf process directly from the *occam*-π declaration in Section 4.2.3:

\[
\text{ELF} \ (\text{id}) = \\
\text{report} \ ! \ \text{working} \ . \ \text{id} \rightarrow \ \text{RANDOM_TIMEOUT} \ (\text{WORKING} \ . \ \text{TIME}) ; \\
\text{report} \ ! \ \text{elfReady} \ . \ \text{id} \rightarrow \ \text{just_elves} \ . \ \text{a} \rightarrow \ \text{just_elves} \ . \ \text{b} \rightarrow \\
\text{elves2santa} \ . \ \text{id} \rightarrow \ \text{santa_elves} \ . \ \text{a} \rightarrow \ \text{santa_elves} \ . \ \text{b} \rightarrow \\
\text{report} \ ! \ \text{consult} \ . \ \text{id} \rightarrow \ \text{santa_elves} \ . \ \text{a} \rightarrow \ \text{santa_elves} \ . \ \text{b} \rightarrow \\
\text{report} \ ! \ \text{elfDone} \ . \ \text{id} \rightarrow \ \text{elves2santa} \ . \ \text{id} \rightarrow \\
\text{elves2santa_ack} \rightarrow \ \text{ELF} \ (\text{id})
\]

where, as mentioned above:

\[
\text{RANDOM_TIMEOUT} \ (\text{WORKING} \ . \ \text{TIME}) = \ \text{SKIP}
\]

Note: for simplicity, the above process has been bound directly to its channels. If needed (and it is not for this system), instances of this process could be bound to other channels through *channel renaming*. We will define the other processes similarly bound to their channels.

5.3 Santa

Here is the Santa process, following Sections 4.1 and 4.2.3:

\[
\text{SANTA} = \\
\ (\text{reindeer2santa} \ ? \ \text{id} \rightarrow \\
\ \text{report} \ ! \ \text{allReindeer} \ . \ \text{0} \rightarrow \ \text{report} \ ! \ \text{harness} \ . \ \text{id} \rightarrow \\
\ (\ ; \ \text{x} :<0..(\\text{G_REINDEER} - 2)> \ \emptyset \\
\ (\text{reindeer2santa} \ ? \ \text{id} \rightarrow \ \text{report} \ ! \ \text{harness} \ . \ \text{id} \rightarrow \ \text{SKIP})) ; \\
\text{report} \ ! \ \text{mushMush} \ . \ \text{0} \rightarrow \ \text{santa_reindeer} \rightarrow \\
\ \text{RANDOM_TIMEOUT} \ (\text{DELIVERY} \ . \ \text{TIME}) ;
\]

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, 1 2009.
5.4 Deadlock and Livelock

Deadlock freedom is a crucial property that all systems (probably) should have. The FDR tool contains an option for checking the presence or the absence of deadlock; it is as simple as loading the desired specification and clicking the ‘deadlock’ button. In practice, we might encounter one of the problems described earlier: if the model is large, FDR might either run for a very long time, or simply run out of memory (exploring the state space needed for the model) and terminate without an answer.

We did encounter this problem when asking FDR to analyse the full model with 10 elves, 9 reindeer, and one Santa. This is where fully automated verification becomes machine assistance to further reasoning.

Each report made by any SYSTEM process adds one state to that process. Each reindeer and elf cycle through four reports each. Santa has 21 different reports. With nine reindeer, ten elves and one Santa, the potential state space increase arising from the reports is over eight trillion. The actual increase will be less than this, since large areas of state space will be barred because of the other internal synchronisations. Nevertheless, that still opens up too much space to analyse.

We need to reduce the state space in the system given to FDR. The report channel is the only one not hidden within SYSTEM. None of the SYSTEM sub-processes synchronise with each other to make a report – they all interleave on its use. Therefore, no SYSTEM process can ever be blocked (for internal reasons) trying to make a report. For deadlock analysis of SYSTEM, this report channel is irrelevant and may be removed. Resubmitting SYSTEM (without any reports), FDR confirms that this system is deadlock free. We can now deduce that the original SYSTEM (with all the reports) is also deadlock free.

To prove the absence of livelock, we click the ‘livelock’ button. FDR immediately reports that the version of SYSTEM (without any reports) is not livelock free. This is because we removed all external signalling, but left the system still running. That is livelock!
To deal with this, a third version of SYSTEM is produced that leaves in just two reports, one in each branch of the alternation in the Santa process. We now have external reports whenever a group of elves or reindeer wake up Santa. FDR confirms that the system is livelock-free. There can be no infinite sequence of purely internal actions. Putting back all the reports merely adds external actions. Therefore, the original SYSTEM is also livelock-free.

This is a major achievement in terms of proving properties for safety critical software. For example, it would be catastrophic if the system controlling the flight correction system of the B2 plane deadlocked. (The B2 plane is aerodynamically unstable on all three axes and require constant flight correction; a task of which no human is capable, thus it must be done by computerised fly-by-wire systems [Wikipedia 2007]). This would most likely cause the plane to lose control and crash.

We have now succeeded in proving two of the three types of properties that we wished to reason about, namely the absence of deadlock and livelock. What remains is to reason about the ordering of output signals, which we will do in the following section.

5.5 Event Ordering and Synchronisation

In the previous section we illustrated the use of the FDR tool to verify that the implementation does not livelock or deadlock. Another issue is of equal importance exists, namely the ordering of any control signals it generates.

In our solution of the Santa Claus problem, all processes share a report channel. Considering these reports as control signals, we now concern ourselves with ensuring that our control logic does not generate incorrect sequences of these signals.

The idea of a shared reporting channel over which control messages for an embedded system are serialised is not very realistic. In such a system, each internal component may be wired directly to the device it controls, so that the signals may travel in parallel. In fact, CSP makes no semantic distinction between a realisation of the report channel as a shared channel (down which reports are serialised) and a parallel array of channels indexed by ReportTag – CSP semantics serialise all events. The occam-π implementation (Section 4) defines a SHARED report channel. This could trivially be changed to an array of separate reporting channels, with no change to the CSP formalisation.

Simply observing serialised reports from SYSTEM may reveal out-of-order reports. Of course, if none are observed, this does not prove that such things may not happen. However, we can capture assertions about trace ordering in CSP in such a way that their adherence by a system can be verified. Here is one such (informal) assertion:

Santa never says “Ho Ho Ho ... Some elves are here” until at least three elves have reported that they are ready to consult.

To check such assertions formally, we turn to the FDR tool once again. First we devise a process that performs the check and causes deadlock if the check fails. Then, we add this checking process to the system and ask FDR if it is still deadlock-free. If that passes, we know that the assertion is always honoured. For the given assertion, we need a checker that:
—accepts and counts inputs on the report channel (report ? x.y);
—ignores reports whose message type (x) is not elfReady or threeElves;
—adds one to the count if the message type is elfReady;
—STOPs if the message type is threeElves and the count is less than three – otherwise removes three from the count.

Here it is:

CHECK_A (n) =
  report ? x.y ->
  if x == elfReady then
    if n > N_ELVES then STOP else CHECK_A (n+1)
  else if x == threeElves then
    if n < 3 then STOP else CHECK_A (n-3)
  else
    CHECK_A (n)

where STOP is a process that refuses to engage in any activity – not even termination. It represents a deadlocked process.

FDR cannot analyse any process containing unbounded recursion. The test whose falsity triggers CHECK_A (n+1) in the preceding script will always be false when this checker is run in parallel with SYSTEM. However, this extra test bounds the recursion; so far as the FDR analyser is concerned, we need it! The other test whose falsity triggers CHECK_A (n-3) later in the script is, of course, essential for the purpose of the check.

Now, add this process into the system, initialising its count to zero:

CHECK_A_SYSTEM = SYSTEM \{ report \} CHECK_A (0)

The checking process and the original system synchronise on the report channel in the above. If SYSTEM ever generates a threeElves message without there being at least three outstanding elfReady messages, the CHECK_A (0) process stops. Any further reports from SYSTEM will be blocked. Since Santa has to generate a report (at least) once per cycle, Santa will stop. The reindeer and elves communicate with Santa (at least) once per cycle, so they will all stop. The two partial barrier processes also communicate with Santa (at least) once per cycle, so they will all stop. Hence, CHECK_A_SYSTEM will deadlock. FDR says CHECK_A_SYSTEM will not deadlock. Therefore, the assertion never fails.

6. REPORTING MORE DETAIL
There are more states in the SYSTEM than are currently being reported.

6.1 Santa’s Waiting Room
An interesting example concerns the assembly of elves into groups of three. There are two stages that each elf goes through, represented by the events just_elves_a and just_elves_b. We may imagine the state in between these events represents an elf being in Santa’s waiting room, a room that can hold only three of them (a property enforced by the logic within XP_BAR process). To observe these elf states, we add another report to the elf process (Section 5.2):
ELF (id) =

... just_elves_a -> report ! elfWaiting.id -> just_elves_b -> ...

Let us verify that the elfWaiting reports obey the same constraints as elfReady with respect to the threeElves reports from Santa:

Santa never says “Ho Ho Ho ... Some elves are here” until at least three elves have reported that they are in the waiting room.

Similar to CHECK_A in Section 5.5, we define:

CHECK_B (n) =
report ? x.y ->
if x == elfWaiting then
  if n > N_ELVES then STOP else CHECK_B (n+1)
else if x == threeElves then
  if n < 3 then STOP else CHECK_B (n-3)
else
  CHECK_B (n)

Fortunately, when FDR is asked to consider:

CHECK_B_SYSTEM = SYSTEM [| {} report {} |] CHECK_B (0)

its deadlock freedom check fails. The event trace leading to deadlock reported by FDR shows that a Santa threeElves report can occur before any elfWaiting report.

In CHECK_A_SYSTEM, a similar problem does not arise because Santa’s threeElves report requires a prior just_elves_ping, which requires the XP_BAR process to have received three just_elves_a events (each of which requires an elfReady report to have been made).

In the CHECK_B_SYSTEM, deadlock does arise. Santa’s threeElves report still requires a prior just_elves_ping, which requires the XP_BAR process to have received three just_elves_a events. None of these requires any elfWaiting report to have been made. FDR observes, therefore, that threeElves may occur first – provoking STOP in CHECK_B and deadlock in CHECK_B_SYSTEM. This needs attention.

6.2 A Better Waiting Room

We need to modify the system so that CHECK_B does not cause deadlock. We need to enforce the elfWaiting reports to be made before just_elves_ping. Once we know we have to do this, it is easy – simply add an acknowledgement event for the just_elves_a event and report elfWaiting in between. This requires modifications to both the ELF process and to XP_BAR. Because the latter now has behaviour beyond an extended partial barrier, we rename it:

WR (n) =

(; x:<1..n> Θ (just_elves_a -> just_elves_a_ack -> SKIP)) ;
just_elves_ping ->
(; x:<1..n> Θ (just_elves_b -> SKIP)) ; WR (n)

The elf process becomes:
ELF (id) = 
   just_elves_a -> report ! elfWaiting.id ->
   just_elves_a_ack -> just_elves_b ->

Now, rebuild SYSTEM as SYSTEM_WR, substituting WR for XP_BAR. Then, ask FDR to analyse:

CHECK_B_SYSTEM_WR = SYSTEM_WR [ | { report } | ] CHECK_B (0)

and it reports this deadlock free. Therefore, the assertion from Section 6.1 is verified for this new system.

6.3 An Even Better Waiting Room

The waiting room only has room to hold three elves at a time. We should be able to strengthen our assertion concerning the elf reports that they are in the waiting room and Santa’s welcome report as follows:

Santa never says “Ho Ho Ho ... Some elves are here” until exactly three elves have reported that they are in the waiting room.

Here is the necessary check:

CHECK_C (n) =
   report ? x.y ->
   if x == elfWaiting then
     if n >= 3 then STOP else CHECK_C (n+1)
   else if x == threeElves then
     if n < 3 then STOP else CHECK_C (0)
   else
     CHECK_C (n)

However, when FDR analyses

CHECK_C_SYSTEM_WR = SYSTEM_WR [ | { report } | ] CHECK_C (0)

it reports deadlock, with the triggering trace showing four elfWaiting reports and no threeElves. The problem is that, although the waiting room does not allow four elves to be present, as soon it wakes up Santa (just_elves_ping) it releases its three elves (just_elves_b) and recurses to allow the next group of elves to assemble (just_elves_a). The arrival of the first elf in this next group and its reporting of that fact proceeds in parallel with Santa’s report welcoming the previous group (threeElves). These two reports may happen in either order – hence the violation.

To enforce our required constraint on the ordering of these reports is also easy. Simply delay the recursion of the waiting room until Santa has made his welcoming report – no elf wanting to enter can do so until that recursion happens. To enforce this delay, modify the waiting room process to offer a second just_elves_ping to Santa, which he does not accept until he has made that welcoming report.

WR (n) =
   (; x:<1..n> ! (just_elves_a -> just_elves_a_ack -> SKIP)) ;
   just_elves_ping ->
Santa’s response to being woken up by a consulting group of elves needs to change:

\[
\begin{align*}
\text{SANTA} = \\
(\text{reindeer}_2\_\text{santa} \ ? \ id \rightarrow ...) \\
[] \\
(\text{just_elves_ping} \rightarrow \\
\quad \text{report} ! \text{threeElves}0 \rightarrow \\
\quad (; x:\langle0...G\_\text{ELVES} - 1\rangle \ (\text{elves}_2\_\text{santa} ? \ id \rightarrow \text{report} ! \text{greet} \ . id \rightarrow \text{SKIP}) ; \\
\quad \text{just_elves_ping} \rightarrow \text{report} ! \text{consulting}0 \rightarrow ... ) 
\end{align*}
\]

For verifying \text{CHECK}_C, the above \text{just_elves_ping} could have been made immediately following the \text{threeElves} report. We made SANTA issue his greeting reports to the arriving elves first to ensure these reports are completed before any new elves report they are in the waiting room — but that is a different issue.

Now, when FDR is asked to verify deadlock freedom of \text{CHECK\_C\_SYSTEM\_WR}, it does. Sections 4.1 and 4.2.3 list several other ordering constraints. These can all be verified in the same way.

### 6.4 Event Ordering through Refinement

Previously, we have demonstrated verification through deadlock checking; another approach uses refinement checking. FDR analyses CSP processes with respect to three semantic models: traces, failures and failures-divergences. A process trace is a finite ordered sequence of events that it can perform. A process failure is a trace paired with a refusal set (which is a set of events that, when offered to the process after it has performed the given trace, it may refuse). A process divergence is a trace such that, after performing that trace, it may livelock (i.e., it may perform an infinite sequence of internal actions, refusing all external events). The traces of a process is the set of all its traces. The failures of a process is the set of all its failures. The divergences of a process is the set of all its divergences. Traces-refinement is defined as this:

\[
\text{SPEC} \sqsubseteq_T \text{IMPL} \equiv \text{traces}(\text{IMPL}) \subseteq \text{traces}(\text{SPEC})
\]

which reads SPEC \sqsubseteq_T IMPL: “a specification process SPEC is traces-refined by an implementation process IMPL.”

Failures-refinement and failures-divergence-refinement are defined similarly:

\[
\begin{align*}
\text{SPEC} \sqsubseteq_F \text{IMPL} & \equiv \text{failures}(\text{IMPL}) \subseteq \text{failures}(\text{SPEC}) \\
\text{SPEC} \sqsubseteq_{FD} \text{IMPL} & \equiv (\text{failures}(\text{IMPL}) \subseteq \text{failures}(\text{SPEC})) \land \\
& (\text{divergences}(\text{IMPL}) \subseteq \text{divergences}(\text{SPEC}))^3 
\end{align*}
\]

A more in-depth discussion of traces, refinement and failures can be found in [Hoare 1985; Roscoe 1997; Schneider 1999].

\[^3\text{Note the direction of the} \sqsubseteq, \text{which might be a little counterintuitive.}\]
Now consider the following ordering rule:

After three elves have reported their arrival in the waiting room, Santa greets them and only them. The order in which Santa greets these elves may differ from the order in which they arrived. For each consultation, the arrival reports must happen before the greetings.

Here is a formalisation of the above specification:

\[\text{WG3} = \]\n\[\text{\mid}_{\sim} x: \text{E\_set} \ @ \ \text{report} ! \ \text{elfWaiting}.x \rightarrow (\]
\[\text{\mid}_{\sim} y: \text{diff(E\_set, \ \{x\}))} @ \ \text{report} ! \ \text{elfWaiting}.y \rightarrow (\]
\[\text{\mid}_{\sim} z: \text{diff(E\_set, \ \{x,y\}))} @ \ \text{report} ! \ \text{elfWaiting}.z \rightarrow (\]
\[\text{\mid}_{\sim} a: \text{\{x,y,z\}} @ \ \text{report} ! \ \text{greet}.a \rightarrow (\]
\[\text{\mid}_{\sim} b: \text{diff(\{x,y,z\},\{a\})} @ \ \text{report} ! \ \text{greet}.b \rightarrow (\]
\[\text{\mid}_{\sim} c: \text{diff(\{x,y,z\},\{a,b\})} @ \ \text{report} ! \ \text{greet}.c \rightarrow \text{WG3}\]
\[) ) ) ) \]

WG3 is a sequential process generating only elfWaiting reports and Santa greet reports. First, an elf with id x chosen from the set \(\{0, \ldots, 9\}\) says he is waiting; this is followed by a waiting message from an elf with an id y chosen from the set \(\{0, \ldots, 9\}\) \(\setminus\) \(\{x\}\). Now Santa greets an elf with id a from the set \(\{x, y, z\}\), then another greeting of an elf with id b from the set \(\{x, y, z\}\) \(\setminus\) \(\{a\}\), and finally greets an elf with id c from the set \(\{x, y, z\}\) \(\setminus\) \(\{a, b\}\).

The \(\text{\mid}_{\sim} e: S \ @ \ P\) construction is a replicated internal choice. That is, the process P is executed with a value of e chosen arbitrarily from the set of elements of S.

diff(S, T) = S \setminus T where S and T are sets.

To compare SYSTEM_WR against WG3, we must at least ensure trace equivalence. To do this, all reports other than elfWaiting and greet must be hidden (or commented out). Assume this has been done. FDR then confirms the following traces refinements: SYSTEM_WR \(\sqsubseteq_T\) WG3 and WG3 \(\sqsubseteq_T\) SYSTEM_WR. This means they are traces equivalent. Thus we can conclude that the two processes have exactly the same set of traces. In particular, there is no trace of SYSTEM_WR that violates the ordering rule.

Traces refinement does not allow us to conclude anything about the liveness of the systems analysed. For this, we need the failures model. FDR quickly verifies that WG3 \(\not\sqsubseteq_F\) SYSTEM_WR. This means that any failure of SYSTEM_WR is also a failure of the specification WG3 (i.e., it is allowed). Turning this around: suppose WG3 generates a trace, \(t\), and is offered a set of events, S, that is not a refusal set (i.e., it will definitely accept one of them). Then, when SYSTEM_WR generates that same trace and is offered the same set of events, SYSTEM_WR will definitely proceed with one of them. It is as alive as the specification WG3 – and never stalls.

We cannot compare WG3 and SYSTEM_WR under the failures-divergences model because the hiding of all reindeer reports allows the latter to diverge – and WG3 has no divergences. To make such comparison, WG3 must be enhanced to allow the interleaving of at least one reindeer report (e.g. Santa’s allReindeer) – and this must not be hidden in SYSTEM_WR (which eliminates its divergence).

We now have CSP formalisms for our Santa Claus system, verified for livelock and deadlock freedom and that a range of event ordering rules are obeyed. This
now needs to be turned into executable code – a process that is, and has to be, trivial – see on-line Appendix D.

7. RELATED WORK

7.1 Other Implementations

As earlier described, a number of other implementations of possible solutions to the Santa Claus Problem exist.

The original solution by Trono himself [Trono 1994] was in Java; the corrected solution by Ben-Ari [Ben-Ari 1998] was in Ada.

The people behind Polyphonic C# [Benton et al. 2004] (now known as Cω) have demonstrated a solution [Benton 2003] using chords in Cω. Chords allow a method to have multiple headers, and only when all headers have been called will the body execute; this makes ‘message passing’ (and synchronisation) between multiple processes possible.

Simon Peyton Jones also uses the Santa Claus Problem as a running example in his chapter “Beautiful Concurrency” in the Beautiful Code book [Jones 2007]. Jones’ solution is in Concurrent Haskell with Software Transactional Memory.

Other approaches using Actors with Multi-headed Message Receive Patterns [Sulzmann et al. 2008] (Actors, which preceded the notion of active objects, were originally proposed by Hewitt in [Hewitt 1977]), Active C# [Güntensperger and Gutknecht 2004], and state classes [Cameron et al. 2006].

We have additional solutions ([Hurt and Pedersen 2008]) written in Java, C#, Groovy, C with Pthreads and C with MPI [Dongarra 1994]. These are available at www.santaclausproblem.net, together with a number of other occam-π implementations (including a symmetric version that does not use barriers, and a version using process mobility).

Note that Trono’s description of the problem describes only the interactions between entities. It does not describe any external signalling, reflecting internal state changes, that are necessary to verify behaviour. Simon Peyton Jones’ solution to the Santa Claus problem has only four external signals (i.e., print statements reporting reindeer delivering toys, elves consulting with Santa, Santa being woken by all the reindeer, and Santa being woken by a group of three elves). In our version of the system, we defined 17 different kinds of report, reflecting key state changes within the entities – and, then, specified a range of properties on those reports to be verified. See on-line Appendix E for an abbreviated occam-π solution using only the four signals described by Peyton Jones.

7.2 Model Checking and Formal Verification

Analysing (or developing) code by modelling aspects of interest in a formal specification language has been an area of immense interest over the years. Trusted tools that can automatically check for a range of standard and user-specifiable definitions of good behaviour are required. We mention a few of these here.

One often used formalism, when it comes to protocol verification, is Murφ [Melton et al. 1996] (Examples include [Dill et al. 1992] and [Mitchell et al. 1997]). Murφ can be used to describe a system of iterated guarded commands, much like the Unity language of [Chandy and Misra 1988]. A Murφ specification is compiled to
C++ and linked with code for a verifier which checks for invariant violations, error statements, assertion violations, and deadlock [Dill et al. 1992]. This tool is based on Bryant’s Ordered Binary Decision Diagrams (OBDD) [Bryant 1986], and like other tools for (formal) verification the state space explosion becomes a problem, but certain techniques do exist to help alleviate this [Ip and Dill 1993].

Another popular tool, also based on BBDs is νSMV [Cimatti et al. 2002], which allows specifications to be expressed in Computational Tree Logic (CTL) and Linear Temporal Logic (LTL), using BDD-based and SAT-based [Biere et al. 1999] model checking techniques.

A similar approach can be found in the SPIN [Holzmann 1997] model checker, which is based on Linear Temporal Logic (LTL), and uses the Promela (Process Meta Language) to specify the system to be verified. A new version of SPIN for checking nonblocking MPI programs (mpiSpin) is also being developed [Siegel 2007].

A different approach to formal verification (or at least deadlock detection) is Petri nets and colouring games [Huber et al. 1985; Jensen 1997] where systems are modelled by graphs whose nodes can contain tokens, which are transferred to other nodes along the arcs of the graph according to certain logic rules. In order to determine deadlock-freedom in coloured Petri nets, an NP-complete task, an occurrence graph is constructed – an occurrence graph represents all possible markings, that is, all possible configurations of the Petri net, and strongly connected components are computed. If a strongly connected component exists that does not have any arcs leaving it, then a configuration exists that represents either a livelock or a deadlock. Techniques for reducing the size of the occurrence graph exist; even so, the size of the graphs and the time to evaluate them can both be exponential.

8. REFLECTIONS

8.1 Process Orientation

This paper presents a process-oriented solution to the Santa Claus problem. Initially, the overall design is laid out through process network diagrams, whose nodes represent processes and edges show allowed synchronisations (data flow and barriers). Executable occam-π code and model-checkable CSP script are then developed, with each process being produced independently (thanks to the compositionality of the underlying semantics) and the network descriptions derived directly from the diagrams. State information in this system is sufficiently small so that no state reducing abstractions are needed in the CSP version4. This means that the occam-π and CSP representations are in 1-1 correspondence and that development and maintenance can be performed in whichever form is most convenient.

8.2 Formal Verification

We claim that process-oriented design leads to solutions that directly reflect natural structures in the problem space and that, therefore, our confidence in these solutions is high – they have an element of obvious correctness that makes us feel comfortable. However, we have both executable and model checkable forms so that this confidence can be formally explored and reinforced. Occasionally, when verification shows our

---

4A single unconstrained 32-bit integer potentially introduces four gigastates!
confidence to have been misplaced, corrections are simple to perform and (re-)verify.

At least for Santa Claus, the evidence is this paper supports the above claim. The \texttt{occam-\pi} code worked first time – once it had passed the stringent safety rules checked by the compiler. Correct output appears and the system has never been seen to deadlock. Applying the FDR model checker to the CSP script indeed verifies the absence of deadlock and other bad behaviours, such as livelock. The solution is also free from data race hazards (unsynchronised access to shared information): such things have no expression in CSP or \texttt{occam-\pi}.

Other constraints, specific to the additional reporting we have added to the requirements, have been formalised and verified. Such signals provide external evidence of internal activity. They can be used to drive an animation of that activity or, perhaps, to control the operation of some complex machinery (whose rules of operation are modelled by the system). If such machines are safety critical and inappropriate (e.g. wrongly ordered) control signals would lead to breakdown, verification is essential.

### 8.3 The Santa Claus Experience

An example of the formal specification and verification of signal ordering rules is shown in Section 5.5. The rule is directly modelled by a CSP process, \texttt{CHECK.A}, observing the system reports. This observer provokes deadlock (by simply stopping and, hence, refusing further reports) if it sees a rule violation. However, the model checker verifies deadlock-freedom of the parallel observer-observed system. Therefore, we may deduce that the observed system always honours the rule.

A maintenance change, introducing further reports showing detail of more internal states, was described in Section 6.1. Perhaps these signals were needed to drive a more sophisticated animation (or machine). Here was an example where our confidence in the “obvious correctness” of our design was dented. The original reporting rule (whose verification still, of course, holds in the modified system) was itself modified by replacing some of the old reports with the new ones.

Although we expected the rule to be honoured – and we never saw it dishonoured in any run of the executable – the model checker immediately reported the potential for violation and told us exactly how it could occur! Armed with this information, the CSP script was easily changed to eliminate that particular problem. Formal verification then confirmed that the corrected system now obeyed the new rule, along with all the old rules. Because of the direct structural correspondence between the CSP script and \texttt{occam-\pi} code, this correction was trivially carried over to the executable form (on-line Appendix D). A similar development was described in Section 6.3, although this time we had anticipated the need for system correction.

A different approach to rule verification is described in Section 6.4. This time, a (highly) non-deterministic process is built to generate only the signals mentioned in the ordering rule and, explicitly, to honour that rule – this becomes the formal specification of the rule. The model checker then verifies that our system, with all signals other than those in the rule hidden, \texttt{failure-refines} that specification. This means that our system is as \textit{alive} as the specification – this is, after any trace of its execution, the implementation is able to perform anything that the specification demands.
8.4 Some Other Considerations

There is one aspect of our solution that we have not verified: Santa’s prioritisation of the reindeer wake-up signal over that from a group of elves. Our implementation is direct and trivial, using the PRI ALT construct of occam-π (Sections 3.4 and 4.1). Our CSP formalisation maps this to a plain external choice (Section 5.3). CSP does not address priorities, so we have no direct way to express this part of the specification formally. We can, however, express this indirectly (and verify) using an additional priority managing process with which Santa, the reindeer and the elves must all engage. This is discussed in the on-line Appendix F.

We note that, even though switching between executable occam-π code and model-checkable CSP script is not difficult, it would be simpler to deal with only one formalism. Prospects for this are discussed in on-line Appendix G.

Finally, the solutions developed in this paper work with only the static network mechanisms of occam-π. All connections between processes (whether through channels or barriers) are laid down in advance. They are always available to the processes and care must be taken that they are used only at appropriate times.

Mobile processes, along with mobile channels and barriers, enable process networks to be dynamic: they may change their size (number of processes, channels, barriers) and shape (connection topology) as they run – much like living organisms. One of the benefits is that all connections do not have to be established statically, in advance of when they are needed and open to abuse. This is discussed in on-line Appendix H.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: http://www.acm.org/pubs/citations/journals/toplas/2009-1-1/p1-welch.

ACKNOWLEDGMENTS

We thank Jason Hurt from the University of Nevada, Las Vegas, for the work put into the several versions of the Santa Claus problem in other languages that he wrote for us for comparison reasons [Hurt and Pedersen 2008].

Special thanks are also owed to Kurt Wallnau of the Software Engineering Institute at Carnegie Mellon University. Kurt pointed out the problem discussed in Section F of not having a verifiable solution to the prioritised choice exercised by Santa (between responding to reindeer or elves) and the apparent violation of this requirement by our solution. He also assisted in its resolution.

We thank all our colleagues in the concurrency research group [Barnes et al.] at the University of Kent for the (on-going) design and construction of the occam-π language and toolset, and their constructive criticisms on this paper.

Finally, we thank our anonymous reviewers of this paper for their diligence and valuable feedback and advice. This was much appreciated.

REFERENCES

Santa Claus: Formal Analysis of a Process-Oriented Solution


Santa Claus: Formal Analysis of a Process-Oriented Solution


Received December 2008; Revised July 2009; accepted Month Year

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, 1 2009.
A. STATES AND REPORTS

The Santa Claus system has three kinds of component: Santa, reindeer and elf. Their states are outlined below, along with the reports we require them to make as they cycle through them. Most of these outline ‘states’ are a composite of a short sequence of states. Some allow many alternative sequences (e.g. when harnessing the reindeer, the order in which they are harnessed does not matter). The reports are triggered by internal state changes – they signal their occurrence to the system environment (which may be a machine controlled by these signals). The state changes are triggered by event synchronisations between the system processes, hidden from their environment. Those events are not specified here – their design, together with any additional components deemed necessary, is an implementation matter (Section 4).

—Santa
—States:
—Sleeping — wait to be woken up either by all the reindeer ready for harnessing or a group of three elves wanting an audience;
—Delivering — harness all nine reindeer, deliver toys, unharness the reindeer, back to sleep;
—Consulting — greet the elves party, consult with them, show them out, back to sleep.
—Reports:
—who woke him up (reindeer or elves);
—harnessing each reindeer;
—start of delivery round;
—end of delivery round;
—unharnessing each reindeer;
—greeting each elf;
—start of consultancy session;
—end of consultancy session;
—showing each elf the door.

—Reindeer (nine of them)
—States:
—Vacationing – do nothing until bored, then go home;
—Waiting – wait for the other reindeer to come home, then approach Santa;
—Delivering – get harnessed, wait for the other reindeer to be harnessed, deliver toys, get unharnessed, then back on holiday;
—Reports:
—start of vacation postcard;
—returning home email;
—start of toy delivery message;
—return from toy delivery.

—Elf (ten of them)
—States:
—Working – work until a problem (or new idea) arises, then go to see Santa;
—Waiting – wait for two other elves who also want to see Santa, approach Santa as a group of three (this must be the only such group doing so);
—Consulting – greet Santa, wait for the other elves to greet Santa, consult together, say goodbye to Santa, then back to work;
—Reports:
—starting work shift;
—when a work problem (or idea) occurs;
—consulting with Santa;
—consultancy session finished.

B. EXTENDED PARTIAL BARRIER – OCCAM-\(\pi\)

The following code declares the occam-\(\pi\) process for an extended partial barrier, as described in 4.2.2.

```plaintext
PROC xp.bar (VAL INT x, CHAN BOOL a?, b?, ping!)
WHILE TRUE
SEQ
  i = 0 FOR x
  a ? any:
    ping ! TRUE -- barrier complete, trigger special process
    i = 0 FOR x
    b ? any:
      ;
```

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, 1 2009.
C. CSP SCRIPTS

C.1 Santa Claus Events

First, we declare all the constants, types and events used by the system:

```
N_REINDEER = 9  -- number of reindeer
G_REINDEER = 9  -- number of reindeer needed to meet Santa
N_ELVES = 10    -- number of elves
G_ELVES = 3     -- number of elves needed to meet Santa
```

```
nametype R_set = {0..(N_REINDEER - 1)}     -- reindeer ids
nametype E_set = {0..(N_ELVES - 1)}       -- elf ids
```

```
datatype ReportTag =
  holiday | deerReady | deliver | deerDone |      -- reindeer reports
  working | elfReady | elfWaiting | consult | elfDone |      -- elf reports
  allReindeer | harness | mushMush | woah | unharness |      -- santa reindeer reports
  threeElves | greet | consulting | done | goodbye      -- santa elf reports
```

```
channel report : ReportTag.E_set           -- report.id (E_set includes R_set)
channel reindeer_2_santa : R_set           -- reindeer id
channel reindeer_2_santa_ack              
channel elves_2_santa : E_set              -- elf id
channel elves_2_santa_ack                 
channel just_elves_ping, just_elves_a, just_elves_b -- extended partial barrier
channel santa_elves_a, santa_elves_b     -- partial barrier
channel just_reindeer, santa_reindeer     -- barriers
```

C.2 Partial Barriers

Like occam-\(\pi\), CSP has no primitives for partial barriers and we have to model them using processes and events. Here is the regular partial barrier process, again binding in the specific events for this system. It directly follows the occam-\(\pi\) code given in Section 4.2.1:

```
P_BAR (n) =
  (; x:<1..n> @ (santa_elves_a -> SKIP)) ;
  (; x:<1..n> @ (santa_elves_b -> SKIP)) ; P_BAR (n)
```

Note: the syntax \((; x:<1..n> @ P(x))\) indicates a replicated sequence. It means: P(1); P(2); ... ; P(n).

The following CSP expression corresponds to the xp.bar occam-\(\pi\) process described in Section 4.2.2:

```
XP_BAR (n) =
  (; x:<1..n> @ (just_elves_a -> SKIP)) ;
  just_elves_ping ->
  (; x:<1..n> @ (just_elves_b -> SKIP)) ; XP_BAR (n)
```
C.3 Reindeer

We define the reindeer process directly from the $\text{occam-}\pi$ declaration in Section 4.1:

\begin{verbatim}
REINDEER (id) =
  report ! holiday.id -> RANDOM_TIMEOUT (HOLIDAY_TIME) ;
  report ! deerReady.id -> just_reindeer ->
  reindeer_2_santa ! id -> santa_reindeer ->
  report ! deliver.id -> santa_reindeer ->
  report ! deerDone.id -> reindeer_2_santa ! id ->
  reindeer_2_santa_ack -> REINDEER (id)
\end{verbatim}

C.4 The Santa Claus System

We build this up a few processes at a time. From Figure 2, we see that the reindeer interleave on the report and reindeer_2_santa channels, but must synchronise on the just_reindeer and santa_reindeer barriers. To express this, simply omit report or reindeer_2_santa from the synchronisation set of the parallel operator$^5$:

\begin{verbatim}
ALL_REINDEER =
  ( [{| {just_reindeer, santa_reindeer} |} id:R_set @ REINDEER (id))
  {just_reindeer}
\end{verbatim}

Note: the just_reindeer event is hidden in the above, since it is a matter of concern only to the reindeer.

With the partial barriers represented by the shared channels and processes of Figure 6, the elf processes do not synchronise directly with each other at all – they interleave on all their channels:

\begin{verbatim}
ALL_ELVES = ||| id:E_set @ ELF (id)
\end{verbatim}

In CSP (and $\text{occam-}\pi$), concurrency is associative and commutative – so, it does not matter in which order we construct the full system. Following 6, we combine the elves with their extended partial barrier process:

\begin{verbatim}
XP_ELVES =
  ( ALL_ELVES [{| {just_elves_a, just_elves_b} |} XP_BAR (G_ELVES) )
  {just_elves_a, just_elves_b}
\end{verbatim}

and add in Santa:

\begin{verbatim}
SANTA_XP_ELVES =
  ( SANTA [{| {just_elves_ping, elves_2_santa, elves_2_santa_ack |} |] XP_ELVES )
  {just_elves_ping, elves_2_santa, elves_2_santa_ack |}
\end{verbatim}

and add in the (non-extended) partial barrier:

\begin{verbatim}
SANTA_XP_P_ELVES =
  ( SANTA_XP_ELVES [{| {santa_elves_a, santa_elves_b} |} P_BAR )
  {santa_elves_a, santa_elves_b}
\end{verbatim}

$^5$The CSP parallel operator used here consists of a synchronisation set of events embedded between the symbols $|$ and $\|$. Operand processes cannot engage in events from that set independently – both must engage for the event to happen. If the curly brackets of the synchronisation set also include bars (i.e., $|$ and $\|$), the set contains all the messages carried by the channels named.
Finally, we have the whole system:

\[
\text{SYSTEM} = \\
( \text{SANTA}\_\text{XP}\_\text{P}\_\text{ELVES} \\
\{\{\text{santa}\_\text{reindeer}, \text{reindeer}\_2\_\text{santa}, \text{reindeer}\_2\_\text{santa}\_\text{ack} \}\} ) \\
\{\{\text{santa}\_\text{reindeer}, \text{reindeer}\_2\_\text{santa}, \text{reindeer}\_2\_\text{santa}\_\text{ack} \}\}
\]

Note that all events are now hidden except for the report channel.

D. BACK TO OCCAM-π

Section 6.1 modifies the CSP formalisms for our Santa Claus system derived from the occam-π executable code presented is Section 4. These modifications address additional event ordering rules on additional reports providing more detailed information on the behaviour of the system. The modifications were verified for correctness and deadlock and livelock freedom. They need to be turned back into executable code – a process that is, and has to be, trivial:

— a waiting.room process replaces the extended partial barrier xp.bar – this corresponds to the CSP process WR (Section 6.3);
— this waiting.room needs an extra acknowledge channel for just_elves_a_ack and uses it as specified by WR;
— the occam-π elf process also needs to be modified to synchronise on this acknowledgement – as specified by the CSP process ELF (Section 6.2);
— the waiting.room also needs to perform the second synchronisation needed on its ping channel at the end of its cycle;
— the santa process accepts this second ping after making its welcome report and greeting the elf consulting group (Section 6.3).

Here is the waiting room (modified from xp.bar is Section 4.2.2, following WR from Section 6.3):

\[
\text{PROC waiting.room (VAL INT x, CHAN BOOL a?, a.ack?, b?, ping!)} \\
\text{WHILE TRUE} \\
\text{SEQ} \\
\text{SEQ i = 0 FOR x} \\
\text{BOOL any;} \\
\text{SEQ} \\
\text{a ? any} \quad \text{-- let an elf in} \\
\text{a.ack ? any} \quad \text{-- wait for the elf to report its entry} \\
\text{ping ! TRUE} \quad \text{-- wake up Santa} \\
\text{SEQ i = 0 FOR x} \\
\text{BOOL any;} \\
\text{SEQ b ? any} \quad \text{-- let this elf leave the waiting room} \\
\text{ping ! TRUE} \quad \text{-- wait for Santa to greet all the elves}
\]

The elf process needs the extra acknowledgement channel for its interaction with the waiting room. This interaction is more than the partial barrier protocol previously employed (the double output implemented by the sync process). These outputs must now sandwich the report (that the elf is in the waiting room) and
the acknowledgement. The changes are coded in-line below (modified from elf in Section 4.2.3 and following ELF from Section 6.2):

PROC elf (VAL INT id,
    SHARED CHAN BOOL just.elves.a!, just.elves.a.ack!, just.elves.b!,
    SHARED CHAN BOOL santa.elves.a!, santa.elves.b!,
    SHARED CHAN INT to.santa!,
    SHARED CHAN ELF.MSG report!)

WHILE TRUE
SEQ
...
 CLAIM report ! elf.ready; id -- "I want to consult with Santa" + id
 CLAIM just.elves.a ! TRUE -- let me into the waiting room
 CLAIM report ! waiting; id -- "I’m in the waiting room" + id
 CLAIM just.elves.a.ack ! TRUE -- OK, you can let someone else in now
 CLAIM just.elves.b ! TRUE -- wait until 3 elves are in waiting room
 CLAIM to.santa ! id -- say hello to Santa
...
:

Finally, Santa needs to accept the second ping from the waiting room. We modify santa from Section 4.2.3, following SANTA from Section 6.3:

{{{ deal with the elves
 BOOL any:
 just.elves.ping ? any -- a party of elves is at the door
 SEQ
 CLAIM report ! elves.ready -- "Ho, Ho, Ho, some elves are here"
 SEQ i = 0 FOR G.ELVES -- for each elf in the consult party
 INT id:
 SEQ
 from.elves ? id -- receive his id
 CLAIM report ! greet; id -- "Hello elf " + id
 just.elves.ping ? any -- let waiting room know all have arrived
 CLAIM report ! consulting -- "Consulting with elves"
... }}}}
E. A SIMPLER SANTA CLAUS

For completeness and comparison reasons, we have included the occam-$\pi$ version of the program that generates the same output as the Peyton Jones’ implementation (in Haskell using Software Transactions [Jones 2007]). As described in Section 7.1, the latter solution has only four external signals: reindeer delivering toys, elves consulting with Santa, Santa being woken by all the reindeer, and Santa being woken by a group of three elves.

PROC reindeer (VAL INT id, BARRIER just.reindeer, santa.reindeer, SHARED CHAN INT to.santa!, SHARED CHAN REINDEER.MSG report!)

WHILE TRUE
SEQ
random.wait (HOLIDAY.TIME) -- sleep for a random amount of time
SYNC just.reindeer -- wait for all deer to return
CLAIM to.santa ! id -- send id to Santa (to get harnessed)
CLAIM report ! deliver; id -- "I'm delivering toys" + id
SYNC santa.reindeer -- until Santa takes us all home

PROC elf (VAL INT id, SHARED CHAN BOOL just.elves.a!, just.elves.b!, SHARED CHAN BOOL santa.elves.a!, santa.elves.b!, SHARED CHAN INT to.santa!, SHARED CHAN ELF.MSG report!)

WHILE TRUE
SEQ
random.wait (WORKING.TIME) -- work until I have a problem
sync (just.elves.a, just.elves.b) -- wait for two other elves
CLAIM to.santa ! id -- say hello to Santa
CLAIM report ! consult; id -- "I'm consulting with Santa" + id
sync (santa.elves.a, santa.elves.b) -- until Santa has had enough

PROC santa (CHAN INT from.reindeer?, BARRIER santa.reindeer, CHAN BOOL just.elves.ping?, CHAN INT from.elves?, SHARED CHAN BOOL santa.elves.a!, santa.elves.b!, SHARED CHAN SANTA.MSG report!)

WHILE TRUE
PRI ALT
INT id:
from.reindeer ?? id -- the first reindeer is here
CLAIM report ! reindeer.ready -- "Ho, Ho, Ho, the reindeer are here"
SEQ
SEQ i = 0 FOR N.REINDEER - 1
from.reindeer ? id -- receive id from rest of the reindeer
random.wait (DELIVERY.TIME) -- deliver toys for some random time
SYNC santa.reindeer -- signal everyone to return home
BOOL any:
just.elves.ping ? any -- a party of elves is at the door
INT id:
SEQ
CLAIM report ! elves.ready -- "Ho, Ho, Ho, some elves are here"
SEQ i = 0 FOR G.ELVES
  from.elves ? id -- receive id from each elf
  random.wait (CONSULTATION.TIME) -- consult for a random time
  sync (santa.elves.a, santa.elves.b) -- consultancy is over
:
PROC santa.system (SHARED CHAN MESSAGE report!)
BARRIER just.reindeer, santa.reindeer:
SHARED ! CHAN INT reindeer.2.santa:
SHARED ! CHAN BOOL just.elves.a, just.elves.b: -- extended partial barrier
CHAN BOOL just.elves.ping:
SHARED ! CHAN BOOL santa.elves.a, santa.elves.b: -- partial barrier channels
SHARED ! CHAN INT elves.2.santa: -- for santa.elves
PAR
  PAR ENROLL santa.reindeer
    santa (reindeer.2.santa?, santa.reindeer,
      just.elves.ping?, elves.2.santa?,
      santa.elves.a!, santa.elves.b!, report!)
  PAR i = 0 FOR N.REINDEER ENROLL just.reindeer, santa.reindeer
    reindeer (i, just.reindeer, santa.reindeer, reindeer.2.santa!, report!)
  PAR i = 0 FOR N.ELVES
    elf (i, elves.a!, elves.b!, santa.elves.a!, santa.elves.b!,
      elf.santa!, report!)
  xp.bar (G.ELVES, just.elves.a?, just.elves.b?, just.elves.ping!)
  p.bar (G.ELVES + 1, santa.elves.a?, santa.elves.b?)
:
Only processes different from those presented previously are listed above. The processes managing the partial barriers, p.bar, xp.bar and sync are as defined in Sections 4.2.1 and on-line Appendix B.

F. PRIORITISED CHOICE

There is one aspect of our solution that we have not verified: Santa’s prioritisation of the reindeer wake-up signal over that from a group of elves. The implementation is direct and trivial, using the PRI ALT construct of occam-π (Sections 3.4 and 4.1). Our CSP formalisation maps this to a plain external choice (Section 5.3). CSP does not address priorities, so we have no direct way to express this part of the specification formally.

We can, however, express it indirectly, using an event-ordering observer as before. This observer takes note only of the reindeer reports saying they are back from holiday and Santa’s “Ho, Ho, Ho ...” messages greeting either the reindeer or the elves party. The observer keeps a count of reindeer reports, clearing this to zero upon the relevant Santa greeting. If Santa greets an elves party when the reindeer count is nine, the observer refuses all further signals, provoking system deadlock.

However, were we to do this, the model check would show the potential for deadlock – that is, our system does not honour the rule:

Santa never says “Ho Ho Ho ... some elves are here” if all nine reindeer
have reported they are back from holiday and Santa has not since said “Ho Ho Ho ... Some reindeer are here”.

This rule, however, is different from the rule actually required, which is that Santa gives priority to the reindeer wake-up signal (and is honoured).

There is a difference between the back-from-holiday reports from the reindeer and the wake-up signal to Santa: the former precede the latter! When the ninth reindeer reports the end of her holiday, the signal to Santa will certainly be made ... but the offer of that signal follows that last report and is not atomic with it. In fact, there is even a barrier synchronisation of all the reindeer in between them – though that is not relevant. What is relevant is that a group of three elves may signal Santa following the ninth back-from-holiday reindeer report and that Santa can quite properly accept that elves signal (since the reindeer signal has not yet been offered). These circumstances are rare, but we have observed it when running the occam-$\pi$ code.

Whenever Santa actually has a choice of wake-up signals, the occam-$\pi$ system will ensure Santa makes the right choice. These circumstances are quite common: we only need a long consultancy with one party of elves, during which another party assembles and all the reindeer return. Our solution honours the informal requirement regarding prioritised choice but we have no way (in CSP) to formalise and, hence, verify it.

If we want to honour the above event-ordering rule instead, of course we can. We need a priority managing process, PMP, to which Santa enquires for a decision as to which wake-up signal to process (and sleeps until given an answer). The reindeer and party of elves must register with PMP after their last member has arrived but before reporting that arrival.

On the elves side, this is easily managed by the Waiting Room process. For the reindeer, this is not so easy since the last reindeer back from holiday does not know that it is the last! To manage this, simplest is to abandon the just.reindeer barrier synchronisation and replace it with a (full) barrier process – similar to the waiting room (perhaps it is their Stable).

The PMP must always service such registration without blocking. It can be a simple server offering an external choice between an enquiry from Santa or registration from the reindeer or elves. No prioritisation or fairness is required for this service.

If a Santa enquiry is outstanding when a registration comes in, Santa is simply told to take the signal from the registering side (that will soon be made, following a report that it is about to make it). Otherwise, the registration is just noted (and the registering side goes on to report that it is ready to signal and, then, offers its signal to Santa – which, for now, will be blocked).

If no registrations have been made when Santa enquires, the enquiry is noted (leaving Santa blocked waiting for a reply). If one registration exists when Santa enquires, Santa is informed of the registering side and the registration is cleared. If both registrations exist, Santa is told to accept the reindeer signal and that registration is cleared.

Finally, Santa accepts the reply from the PMP with an extended rendezvous, during which the wake-up signal (from the instructed side) is accepted and the
“Ho, Ho, Ho” greeting is reported. This prevents the PMP from accepting a further registration (which would trigger a report of an impending signal) before Santa’s report of its decision has been made.

The PMP logic is easier to express in occam-π or CSP. The modifications to the rest of the system (to replace the just.reindeer barrier with a Stable process and simple changes to the Waiting Room logic to register with the PMP when the third elf arrives and before it reports that arrival), the PMP itself, the observer process checking the rule given in this section and the model checking to verify the rule are omitted for reasons of space. However, all are straightforward and available from the website www.santaclausproblem.net (along with all occam-π codes and CSP scripts in this paper).

G. A UNIFIED OCCAM-π MODEL CHECKER

Building CSP models automatically from occam-π is tractable. (Even the mobile channels, barriers and processes of occam-π, though not used here, have an operational semantics in terms of standard CSP [Welch and Barnes 2005b; 2008].) For producing scripts that do not generate unnecessarily large state spaces when model checking, a transformation tool allowing programmer interaction (to roll states together, carefully introduce non-determinism) would be helpful – although that was not needed for this Santa Claus system.

Building efficient executables automatically from CSP implementation scripts (i.e., those with internal choices resolved) is a little harder – especially if the full range of CSP expression (e.g. external choice over multiway synchronisations [Welch et al. 2006; Welch 2006], complex choice preconditions on channel message values) is allowed.

Either way, being able to translate automatically between these two formalisms unifies verification with the development of efficiently executable programs – one of the grand challenges in Computer Science. We look forward to developments in both these areas.

H. MOBILE CHANNELS AND PROCESSES

In the solution analysed in the paper, network topology is static – all processes have all their channels available all the time. Some care has to be taken not to use channels when inappropriate (e.g. a reindeer may try to communicate with Santa while on vacation). With occam-π mobiles, we can arrange that processes simply do not have those channels until they get into the right state – and not having such channels means that their misuse cannot even be expressed! Of course, it is a natural consequence of mobile system design that the arrivals of channels (or barriers or processes) are the very events triggering their use.

In occam-π, we can construct channels at run-time and pass their ends independently over existing channels to create new topology. For the Santa Claus problem, moving a channel-end from an elf (or reindeer) through the waiting room (or stable) and on to Santa, whilst retaining the other end, establishes a connection (between elf and Santa, or reindeer and Santa) when, and only when, it is needed.

In occam-π, we can construct processes (to well-defined plugin templates) at runtime, send and receive them through channels, plug them into local networks, fire
them up, stand them down and move them on again. We can model the reindeer
and elves as mobile processes that move through holiday resorts, stables, work
benches, waiting rooms, Santa’s Grotto and back again. All those destinations are
also processes though static ones. As the reindeer and elves arrive at each stage,
they plug in and do business.

We sense that these solutions are even more realistic and easy to design, program
and reason about, at least informally, than the static solutions expressed here.
Indeed, some of the careful extra synchronisations, added to the static network
solution presented here to prevent out-of-order external signalling, are not needed
(e.g. an elf only reports that it is in the waiting room when it is in the waiting room
and before it knocks on Santa’s door). Thus, the solutions become simpler.

For formal reasoning, an operational semantics maps occam-π mobile channels
and barriers down to CSP [Welch and Barnes 2005b; 2008]. We believe a similar
approach will yield an operational semantics for mobile processes in occam-π. More
recently, Roscoe has shown, [Roscoe 2008], that CSP+ (which is CSP with the addi-
tion of an exception-throwing operator) can simulate a rich set of new capabilities,
including the mobility features of the π-calculus. This promises a direct traces-
failures-divergences semantics for occam-π, for which new model checkers may be
built with direct ability for reasoning about mobility.

We will leave these approaches for another time. However, occam-π solutions
employing them will be placed on www.santaclausproblem.net.