Communicating Process Architectures 2008
P.H. Welch et al. (Eds.)
IOS Press, 2008

Representation and Implementation of CSP and VCR Traces

Neil C.C. BROWN [a] and Marc L. SMITH [b]
[a] *Computing Laboratory, University of Kent,*
Canterbury, Kent, CT2 7NZ, UK, neil@twistedsquare.com
[b] *Computer Science Department, Vassar College,*
Poughkeepsie, New York 12604, USA, mlsmith@cs.vassar.edu

Abstract. Communicating Sequential Processes (CSP) was developed around a formal algebra of processes and a semantics based on traces (and failures and divergences). A trace is a record of the events engaged in by a process. Several programming languages use, or have libraries to use, CSP mechanisms to manage their concurrency. Most of these lack the facility to record the trace of a program. A standard trace is a flat list of events but structured trace models are possible that can provide more information such as the independent or concurrent engagement of the process in some of its events. One such trace model is View-Centric Reasoning (VCR), which offers an additional model of tracing, taking into account the multiple, possibly imperfect views of a concurrent computation. This paper also introduces "structural" traces, a new type of trace that reflects the nested parallelism in a CSP system. The paper describes the automated generation of these three trace types in the Communicating Haskell Processes (CHP) library, using techniques which could easily be applied in other libraries such as JCSP and C++CSP2. The ability to present such traces of a concurrent program assists in understanding the behaviour of real CHP programs and for debugging when the trace behaviours are wrong. These ideas and tools promote a deeper understanding of the association between practicalities of real systems software and the underlying CSP formalism.

Keywords. Communicating Sequential Processes, CSP, View-Centric Reasoning, VCR, traces.

Introduction

Communicating Sequential Processes (CSP) [1,2] is a model of concurrency based on processes synchronising on shared events. To support this idea, Hoare developed a process algebra to permit the specification of concurrency, and defined a semantics based on traces, failures (deadlocks) and divergences (livelocks). This paper is only concerned with traces: records of a program's event behaviour.

The recent growth in multicore processors has led to the need for programming models that can exploit concurrency. In contrast to the popular locks and threading models, CSP and process-oriented design promise an elegant and powerful alternative when used properly. The occam-π programming language [3] and libraries for several other mainstream programming languages use CSP as the basis for their concurrency. CSP and formal methods have a reputation as being challenging and divorced from practice. However, applying CSP without proper training can diminish its advantages.

When programmers encounter problems in programming concurrent systems, they naturally turn to familiar methods of debugging – for example, adding "print" statements at points of interest in their programs. Programmers new to concurrency soon discover that due

to non-deterministic scheduling their programs can behave differently from run to run, and that adding these debugging statements can also change the behaviour of the program.

Programmers are effectively seeking to understand their program's behaviour. Debugging statements primarily reveal the behaviour of a single component in the system, not synchronisations between components. In contrast to print statements, traces record these interactions between components – such as channel communications, barriers, etc.

An additional frustration in using disciplined process-oriented programming (such as in occam-π) can be the addition of the necessary wiring that will enable debugging messages to be printed. Programmers must redesign their process networks to route output channels in order to print messages from processes. This further complicates mastering the new programming model.

We have augmented a CSP implementation for Haskell (Communicating Haskell Processes, CHP) [4] to provide a convenient tracing facility. When enabled, it records channel communications and barrier synchronisations that have completed. This is akin to a CSP trace of the program.

When a problem arises, this trace facility allows the programmer to see what events actually occurred, and their order. They can compare this behaviour with their intentions. While this is no substitute for formal reasoning and/or model checking, this is a practical aid for programmers, and introduces them to the notion of tracing. We believe tracing is a viable entry point to the CSP theory for programmers unfamiliar with formal reasoning and model checking tools such as FDR2[5], which can check various safety, refinement and liveness properties of CSP processes.

Our approach is intended to aid programmers in debugging real programs, which may not have been formally modelled. This is distinct from tools such as FDR2 and ProBE, which allow a programmer to investigate the behaviour of CSP models in a language-independent manner.

Recording traces of a computation brings with it a set of questions and challenges. Specifically, how shall traces be represented in a meaningful way, and how shall we implement these traces? View-Centric Reasoning (VCR) [6,7] was developed to address some of challenges posed by the CSP observer, including the bookkeeping practice of reducing observed concurrency to sequential interleavings of events. VCR offers an additional model of tracing, taking into account the multiple, possibly imperfect views of a concurrent computation.

One aspect of traces that programmers may find challenging is the flattening of their parallel-composed process network into a single sequential trace. For this reason, we also introduce structural traces. Structural traces contain notions of sequential and parallel composition that enable these traces to reflect to some degree the sequential and parallel logic of original program.

In this paper we will introduce all three trace types fully (section 1) before explaining how we have implemented the recording of each of these trace types (section 2). We will then present examples of each of the traces (section 3) before looking at related and future work (sections 4 and 5).

1. Background

Tracing is the recording of a program's behaviour. Specifically, a trace is a record of all the events that a process has engaged in. A trace of an entire program is typically an interleaving (in proper time order) of all the events that occurred during the run-time of the program. In this section we briefly recap CSP and VCR tracing, and introduce a new form of trace that will be used in this paper: structural tracing.

For our trace examples in this section, we will use the following CSP system:

$$(a \rightarrow b \rightarrow c \rightarrow \text{STOP}) \underset{\{a\}}{\|} (a \rightarrow d \rightarrow e \rightarrow \text{STOP})$$

1.1. CSP Tracing

A CSP trace is a sequential record of all the events that occurred during the course of the program. Hoare originally described the trace as being recorded by a perfect observer who saw all events. If the observer saw two (or more) events happening simultaneously, they were permitted to write down the events in an arbitrary order.

For example, these are the possible maximal traces of our system:

$$\langle a,b,c,d,e\rangle \quad \langle a,b,d,c,e\rangle \quad \langle a,b,d,e,c\rangle$$
$$\langle a,d,b,c,e\rangle \quad \langle a,d,b,e,c\rangle \quad \langle a,d,e,b,c\rangle$$

1.2. VCR Tracing

A VCR trace is similar to a CSP trace, but instead of recording a sequence of single events in a trace, the observer records a sequence of multisets. In the original model of VCR traces, each multiset represented a collection of simultaneous events, preserving information that the CSP observer had lost. In practice, simultaneity is a difficult concept to define, reason about or observe.

We have therefore adapted the meaning of event multiset traces in VCR. In our implementation in CHP each multiset is a collection of *independent* events. We term a and b to be independent events if a did not observably require b to occur first, and vice versa. The implication is that it was possible for events a and b to occur in either order without altering the meaning of the program.

By convention in this paper, we write multisets of size one without set notation. These are the possible maximal traces of our system:

$$\langle a,b,c,d,e\rangle \quad \langle a,b,d,c,e\rangle \quad \langle a,b,d,e,c\rangle$$
$$\langle a,d,b,c,e\rangle \quad \langle a,d,b,e,c\rangle \quad \langle a,d,e,b,c\rangle$$
$$\langle a,b,\{c,d\},e\rangle \quad \langle a,b,d,\{c,e\}\rangle$$
$$\langle a,d,b,\{c,e\}\rangle \quad \langle a,d,\{b,e\},c\rangle$$
$$\langle a,\{b,d\},\{c,e\}\rangle$$

VCR also permits multiple observers and imperfect observation (some events may not be recorded), but unless specifically stated, in this paper we will be using a single observer that records all events.

1.3. Structural Tracing

To explore other forms of trace recording, we have created structural tracing. CSP and VCR tracing are both based on recording the events in a single, fairly flat trace. Structural tracing is a constrasting approach where a structure is built up that reflects the parallel and sequential composition of the processes being traced. Traces can be composed in parallel with the || operator, and these parallel-composed traces may appear inside normal sequential traces.

There is only one possible maximal trace of our system:

$$\langle a, b, c \rangle \;||\; \langle a, d, e \rangle$$

Note that a single occurrence of an event is recorded multiple times (once by each process engaging it), which is distinct from both CSP and VCR tracing. This makes structural traces quite different from the classic notion of tracing.

1.4. Communicating Haskell Processes

Communicating Haskell Processes (CHP) is a new concurrency library in the tradition of JCSP *et al* for the Haskell programming language [4]. We have chosen to implement our traces in CHP because it was the easiest framework to add tracing to, given the differing implementations of the various CSP libraries and the authors' expertise. The techniques presented here should be immediately portable to any other CSP framework.

CHP is primarily built on top of Software Transactional Memory (STM), a library for implementing concurrency in Haskell [8]. STM is built on the idea of atomic transactions that modify a set of shared transactional variables. Event recording (at least for CSP and VCR traces) takes place in the same transaction as actually completing the event itself, and thus the recording is indivisible from the event.

2. Implementation

2.1. General

The Communicating Haskell Processes library has two types of synchronisations: channel communications (that transmit data) and barrier synchronisations (that have no data). This is also true for most CSP-derived languages and libraries. We have altered CHP so that after the completion of every successful event, the event is recorded. For CSP and VCR this is done by the process that completes the synchronisation. For structural traces, every process that engages in the event records it.

This highlights the fundamental difference between structural tracing and the other two forms; under structural tracing, events are recorded multiple times, but under CSP and VCR tracing events are only recorded once.

Tracing can be enabled or disabled at the start of the program's execution. Switching on tracing is as simple as changing a single line in the user's program. This makes the tracing facility very easy to use.

2.2. CSP Traces

CSP tracing is the most straightforward to implement. Hoare's perfect observer can be implemented by modifying a single sequence shared between all processes (with appropriate mechanisms, such as a mutex, to handle the concurrent updates). Due to Haskell's lists having $O(1)$ time complexity for prepending but not for appending, we add all new events at the head of the list, effectively recording the trace backwards.

2.2.1. Recording Bottleneck

The main problem with CSP tracing is that it serialises the whole program. All processes end up contending for access to the central trace, and thus with multiple threads this can slow down the program. An alternative approach would be to have each process record its own trace with events timestamped (as in notions of timed CSP), and merge these sorted traces by timestamp from sub-processes into their parent process when the sub-processes complete.

The problem with such timestamps is that modern computers (from the point of view of a portable high-level language) often have timers with relatively coarse frequency, such as once per millisecond. CHP is able to complete around a hundred events in that time, and thus events that occur within a millisecond of each other would be recorded in arbitrary order. Therefore we still record CSP traces using a single shared sequence.

2.3. VCR Traces

As described in section 1.2, we group independent events into multisets for VCR traces. If we can deduce a definite ordering of two events, we take the latter event to be dependent on the former. Otherwise, we take the events to be independent. For example, we know that two events must be sequential if they are both performed by the same process-thread. To be able to reason about this and other details, we record events with associated process identifiers.

2.3.1. Process Identifiers

We use process identifiers to deduce information about sequencing. Consider these two CSP systems:

$$(a \rightarrow (b \rightarrow \text{SKIP} \;|||\; c \rightarrow \text{SKIP})) \,;\, (d \rightarrow e \rightarrow \text{STOP})$$
$$(f \rightarrow \text{SKIP} \;|||\; g \rightarrow \text{SKIP}) \,;\, (h \rightarrow \text{STOP} \;|||\; i \rightarrow \text{STOP})$$

We wish to be able to deduce the following facts, where $<$ is the strict partial ordering of the events in time:

$$a < b, a < c$$
$$b < d, c < d$$
$$d < e$$
$$f < h, g < h$$
$$f < i, g < i$$

Process identifiers are one or more integers (sequence numbers) joined together by a sub-process operator, $\underset{x}{\triangleright}$ where x is itself an integer (a parallel branch identifier). The top-level process of the program starts with the identifier 0. Process identifiers only change when the process runs a parallel composition. Before and after the composition, the last (right-most) sequence number is incremented. The identifiers for the sub-processes are formed by appending $\underset{x}{\triangleright}0$ to the current identifier, where x is distinct for each sub-process.

Our example programs, with process identifiers, are shown in figures 1 and 2. Each event is given in a circle. The open double-headed arrows indicate sequencing, and sub-processes are shown in a vertical relation, with an empty (dot) event as a parent. The events are grouped into dashed boxes, which represent the actual processes that will be created when the program is run. Next to each event is the associated process identifier that will be recorded with it.

The sequence numbers in the top row of each figure are not in error. Events d and e are associated with the same sequence number. Event e will definitely be recorded after event d (since the same process is doing them in sequence) so we do not need to use the process identifier to tell them apart. It is only in the case of parallel composition that we must change the sequence numbers, in order to deduce an ordering between the parent's events (such as a and d) and the sub-processes' events (b and c). The incrementing before and after both parallel compositions is what leads to the sequence numbers being 1 and 3 for the second figure.

The following Haskell pseudo-code explains the algorithm for comparing process identifiers:

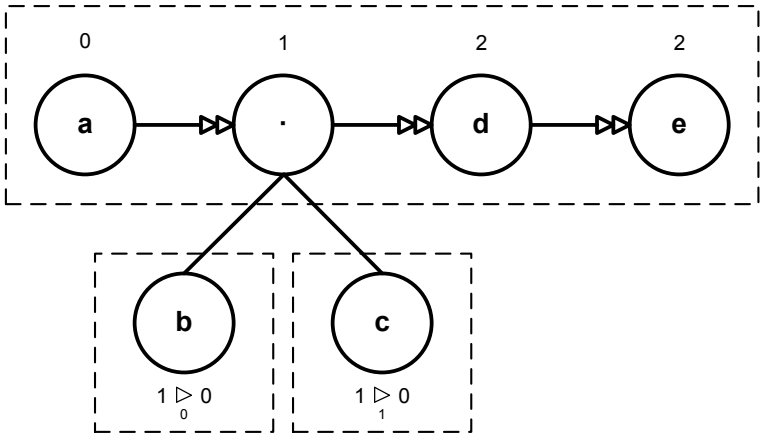

Figure 1. The program $(a \rightarrow (b \rightarrow \text{SKIP} \;|||\; c \rightarrow \text{SKIP})) ; (d \rightarrow e \rightarrow \text{STOP})$, with the identifiers for each event

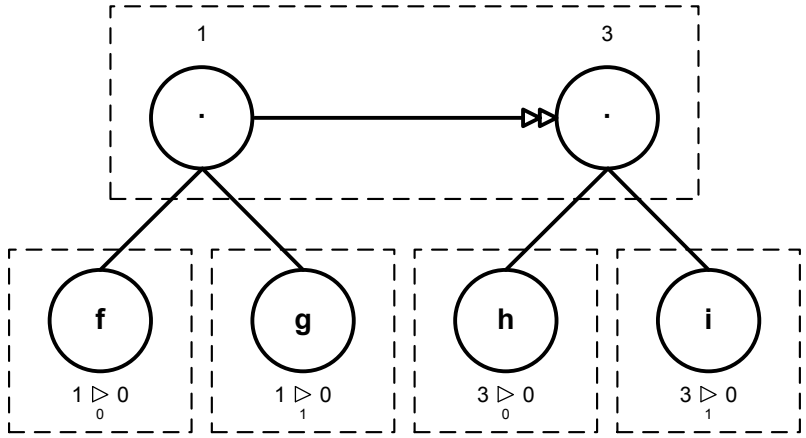

Figure 2. The program $(f \rightarrow \text{SKIP} \;|||\; g \rightarrow \text{SKIP}) ; (h \rightarrow \text{STOP} \;|||\; i \rightarrow \text{STOP})$, with the identifiers for each event

```
data Maybe a = Nothing | Just a

type ProcessId = (Integer, Maybe (Integer, ProcessId))

lessThan :: ProcessId -> ProcessId -> Bool
lessThan (x, Nothing) (y, Nothing) = x < y
lessThan (x, Just (xpar, xpid)) (y, Just (ypar, ypid))
  = if x == y
      then
        (if xpar == ypar
           then lessThan xpid ypid
           else False) -- Independent
      else x < y
```

The data type indicates that a process identifier is a pair of an integer, and an optional (indicated by the *Maybe* type) parallel branch identifier with the further part of the process identifier. So the identifier $2 \underset{1}{\triangleright} 4$ would be represented as (2, *Just* (1, (4, *Nothing*))).

If the process identifiers are both single sequence numbers, we simply compare these sequence numbers. If they are a compound identifier, we again start by comparing the sequence numbers. If they are not equal, we return the value of $x < y$. If however they are equal, we compare their parallel branch identifiers. If these identifiers are not equal, we know the identifiers come from parallel siblings, and we return false. If they are equal, we proceed with comparing the remainder of the process identifiers.

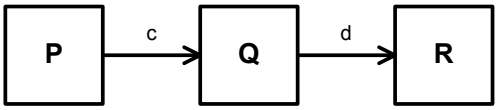

Figure 3. A standard pipeline of three processes

Due to the incrementing of the sequence number before and after running sub-processes, it can be seen that no recorded identifier will ever be an exact prefix of another. In our diagrams (figures 1 and 2), the parent with the same prefix is represented as a dot, and engages in no events itself. Thus, pairs of recorded identifiers will always differ within the length of the shorter identifier, or both identifiers will be equal.

2.3.2. Recording Rules

In the previous section we explained our process identifiers and a partial order over them. We will now show how we use this to record VCR traces, for the moment considering events that only involve one process.

When recording an event in a VCR trace, a process must look at the most recent set of parallel events, here termed Θ. Θ is a set[1] of pairs of an event (for which we use α, β, etc) and a process identifier (for which we use p, q, etc). We must determine, for a given (β, q) whether to add the new event to Θ or whether to start a new set of parallel events. Our rule is straightforward: if $\exists(\alpha, p) \in \Theta : p < q \vee p = q$, we must start a new set of parallel events because an event in the most recent set provably occurred before the event we are adding. Otherwise, add the new event to Θ (the events are independent).

Consider the following CSP traces of our example processes from figures 1 and 2, annotated with process identifiers:

$$\langle a[0], c[1 \underset{1}{\triangleright} 0], b[1 \underset{0}{\triangleright} 0], d[2], e[2] \rangle$$

$$\langle f[1 \underset{0}{\triangleright} 0], g[1 \underset{1}{\triangleright} 0], i[3 \underset{1}{\triangleright} 0], h[3 \underset{0}{\triangleright} 0] \rangle$$

Following our rules, the same traces recorded in VCR form would be:

$$\langle a, \{b, c\}, d, e \rangle$$

$$\langle \{f, g\}, \{h, i\} \rangle$$

2.3.3. Multiple Processes

We have so far considered events with just a single process. In reality, events involve multiple processes synchronising. Consider a process pipeline, where three processes are connected with channels c and d, as shown in figure 3. We will assume here that the reader happens to record the events (recall that events are recorded by the last process to engage in the synchronisation). Process Q will read from channel c, and record this event. Then it will write to channel d and the process R will read and record the event. Because the process identifiers for Q and R have no ordering, the events will be recorded as being independent. However, the communication on channel d was clearly dependent on the communication on from channel c, since process Q communicated on channel c before communicating on channel d.

In order to combat this problem, we modify our synchronisation events so that the party that records the event can know the process identifier of all the other processes that engaged in the synchronisation. The event is recorded with a set of process identifiers (in barriers, there may be more than two participants) rather than a single identifier.

[1]Technically in VCR, it is a multiset, but due to our recording strategy, no pair of event and process identifier will occur multiple times in the same set.

Thus we can adjust our previous rule, now that Θ is a set of pairs of single event identifier and a *set* of process identifiers (for which we will use capital P, Q, etc). For adding a new pair of event and identifier set (β, Q): if $\exists(\alpha, P) \in \Theta, \exists\, p \in P, \exists\, q \in Q : p < q \vee p = q$, start a new set of parallel events. Otherwise, add the new event to Θ.

In our previous example, the communication on channel c will have been recorded with the process identifiers for processes P and Q. When process R then records the communication on channel d with the identifiers for Q and R, the events will be seen to be dependent.

The reader may wonder what happens should it also be the case that $\exists\, p \in P, \exists\, q \in Q : q < p$ in the above rule. This is not possible in the CHP library because the recording of an event is bound into the synchronisation, so events will always be recorded in order. It cannot be the case that an event a that provably occurred *before* b will be recorded *after* b.

2.3.4. Relevance of Event Identifiers

The event identifier currently plays no role in our rules – it is usually subsumed by considering the identifiers of processes engaging in the events. Therefore the only cases where it would be of use are where an entirely different (unrelated) set of processes may engage on two successive occurrences of the same event.

Communication on unshared channels, and channels with only one shared end, are taken care of by considering the processes involved. In these cases, one process will always be involved[2] in the subsequent communications, and we can use this to deduce sequence information. Therefore we need only consider here channels that are shared at both ends.

With channels that are shared at both ends, it is possible for P and Q to communicate once on the channel, then release the ends, after which R and S claim them and perform another communication. In such a situation, we do not consider the second communication to be dependent on the first, and so we do not use the event identifier to deduce any sequence information.

2.3.5. Independence and Inference

The VCR theory describes events in the same multiset as being observed simultaneously, but we have altered this and implemented our traces to record independent events in the same multiset. This means that some of the theoretically possible VCR traces can never actually be recorded in our implementation, and also that some sequence information can be retroactively inferred.

Recall our CSP system from the example in section 1:

$$(a \rightarrow b \rightarrow c \rightarrow \text{STOP}) \underset{\{a\}}{\parallel} (a \rightarrow d \rightarrow e \rightarrow \text{STOP})$$

Imagine that the first two events to occur are a and b, giving us a partial trace: $\langle a, b\rangle$. According to the VCR theory, the possible maximal traces that may follow from this are:

$$\langle a, b, c, d, e\rangle \quad \langle a, b, d, c, e\rangle \quad \langle a, b, d, e, c\rangle$$
$$\langle a, b, \{c, d\}, e\rangle \quad \langle a, b, d, \{c, e\}\rangle \quad \langle a, \{b, d\}, \{c, e\}\rangle$$

Which trace is recorded will depend on which event occurs next. If the next event is c, the theoretical traces are:

$$\langle a, b, c, d, e\rangle \quad \langle a, b, \{c, d\}, e\rangle$$

[2]Or a process and its sub-processes, that we can deduce sequence information about.

In fact, the trace recorded in our implementation will always be the second trace. Because c and d are independent events, they will always be recorded in the same multiset. If c happens first, the partial trace will be $\langle a, b, c\rangle$. But then when d is recorded, because it is independent of all events in the most recent multiset (c, in a singleton multiset), it will be added to that multiset, forming $\langle a, b, \{c, d\}\rangle$, then become the second trace shown above. The trace $\langle a, b, c, d, e\rangle$ can never actually be recorded using our implementation.

If instead the next (third) event is d, the possible theoretical traces are:

$$\langle a, b, d, c, e\rangle \quad \langle a, b, d, e, c\rangle \quad \langle a, b, d, \{c, e\}\rangle$$
$$\langle a, b, \{c, d\}, e\rangle \quad \langle a, \{b, d\}, \{c, e\}\rangle$$

For similar reasons to the previous example, it is not possible to record any of the top three traces. The pairs of events (b, d), (c, d) and (c, e) are each independent, so the traces would always be recorded using the latter two traces (with appropriate multisets) rather than the earlier three traces with only singleton multisets.

It is even possible to infer sequence from a trace with multisets. Consider the trace $\langle a, b, \{c, d\}, e\rangle$. If event d had occurred before c, then b and d would have been grouped into the same multiset (since they are independent). Therefore to produce this trace, c must have happened before d.

These issues reflect the difference between the theory of VCR and our actual implementation of its tracing. Our trace may seem to add obfuscation over and above the CSP trace. But it also abstracts away from some of the unnecessary detail. Grouping two events into a multiset implies that they *could* have happened in either order. For trace compression (see section 2.5), comprehension and visualisation, a more regular trace that abstracts away from some of the scheduling-dependent behaviour of the program may well be appealing in some circumstances.

2.4. Structural Traces

A structural trace is the most natural – and fastest – to record. Each running process records events it has engaged in using a local trace. Since the trace is unshared, there is no contention or need for locks, and hence it is faster than any other method of recording. Sequential events are recorded by adding them to a list. As with the other traces, it adds to the front of the list (reversing the order) to be faster.

When a process runs several sub-processes in parallel, they all also separately record their own trace. Upon completion they all send back their traces to the parent process. These traces are received by the parent and form a hierarchy in the parent's trace. This process of joining traces means that the end result is one single trace that represents the behaviour of the entire program.

The drawback with structural traces is that information about the ordering of events between non-descendant processes is lost. Consider the trace:

$$\langle a, a, a, a, a, a\rangle \;||\; \langle b, b, b, b, b, b\rangle$$

We cannot tell from this trace whether all the events a happened before all the events b, vice versa, a strict interleaving or any other possible pattern.

2.5. Compression

One problem of recording the trace of a system is that a large amount of data is generated. IF the system may generate an event every microsecond[3], that is a million events a second.

[3]This figure is a very rough average of channel communication times on modern machines from various CSP implementations.

Assuming on a 32-bit machine that an event could be reduced to eight bytes per event (four bytes for an identifier, four for a pointer in a linked list or similar), in the worst case nearly a gigabyte of data would be generated approximately every two minutes. Generally programs are likely to fall far below this upper bound, but ideally we would like to reduce the space required.

Processes usually iterate (by looping or recursion), and thus display repetitive behaviour. This repetition is often diluted by observing the behaviour of several processes, but even large process networks can display regular behaviour.

Repeated patterns appearing in the trace means that it should be possible to compress the trace by removing this redundancy. To gain much benefit from this, we need to compress the trace on-line, while the program is still running, rather than off-line after the program has finished.

The obvious compression approaches are forms of run-length encoding that can reduce consecutive repeated behaviour, and dictionary-based compression methods that can spot common patterns and reduce them to an index in a lookup table of frequently occurring patterns.

We also note that viewing the compressed version is often more comprehensible to the programmer than its raw, uncompressed form. Most of the structural traces in this paper have been left in their compressed form for that reason.

2.6. Rolling Trace

A primary use of traces is for post-mortem debugging, especially in the case of deadlock or livelock. In this case the programmer is interested to see what the program was doing leading up to the failure.

This failure could occur after the program has been running for hours, or days. The earlier behaviour of the program will probably not be of interest. Therefore a good policy in these cases would be to keep a rolling trace which records, say, the most recent thousand events. This would require a constant amount of memory rather than accumulating the trace as normal (recording using an array, rather than a linked list).

This would allow tracing to place some time overhead on the program, but only a small amount of memory overhead, and would still be useful in the case of program failure.

3. Example Traces

In this section we present examples of traces of several different small programs. Each trace is recorded from a different run of the program, so the traces will not (necessarily) be direct transformations of each other.

3.1. CommsTime

Commstime is a classic benchmark used in process-oriented systems. Its configuration is shown in figure 4.

For this benchmark, we show approximately six iterations of the commstime loop.

3.1.1. CSP

⟨ prefix-delta, delta-recorder, delta-succ, succ-prefix,
prefix-delta, delta-succ, delta-recorder, succ-prefix,
prefix-delta, delta-succ, delta-recorder, succ-prefix,
prefix-delta, delta-succ, delta-recorder, succ-prefix,
prefix-delta, delta-succ, delta-recorder, succ-prefix,
prefix-delta, delta-succ, succ-prefix, delta-recorder,
delta-succ ⟩

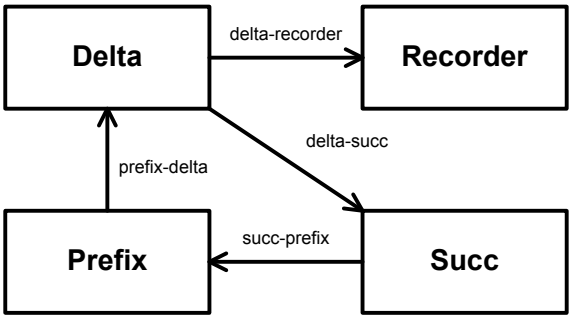

Figure 4. The CommsTime Network

3.1.2. VCR

⟨ {prefix-delta}, {delta-succ}, {delta-recorder, succ-prefix},
{prefix-delta}, {delta-succ}, {delta-recorder, succ-prefix},
{prefix-delta}, {delta-recorder, delta-succ}, {succ-prefix},
{prefix-delta}, {delta-recorder, delta-succ}, {succ-prefix},
{prefix-delta}, {delta-recorder, delta-succ}, {succ-prefix},
{prefix-delta}, {delta-recorder, delta-succ}, {succ-prefix}, {delta-succ} ⟩

3.1.3. Structural

⟨⟨⟨7*⟨delta-succ? , succ-prefix!⟩, delta-succ?⟩
|| ⟨7*⟨prefix-delta! , succ-prefix?⟩ , prefix-delta!⟩
|| ⟨6*⟨delta-recorder?⟩⟩
|| ⟨7*⟨prefix-delta? , ⟨delta-recorder! || delta-succ!⟩⟩⟩⟩⟩

3.1.4. Summary

The CSP trace reflects the interleaving that occurred between all the parallel processes, and shows how the order of events changes slightly at the beginning and end of the trace.

The lines in the middle of the VCR trace are probably what we would expect; singleton sets of prefix-delta and succ-prefix, and a set of the two parallel events from the delta process: delta-recorder and delta-succ. However, the early traces show that there is a different recording that can occur, with the delta-succ event happening first, and the delta-recorder happening in parallel with succ-prefix. This is a valid behaviour of our process network.

It is easy to see the four different processes in the structural trace, each with distinct behaviour. This would be visible even if the channels were not labelled. The regular repetition of all of the processes is also clear.

3.2. Dining Philosophers

The dining philosophers is a classic concurrency problem described by Hoare in his original book on CSP [1]. We use the deadlocking version for our benchmark. To keep the traces simpler and to provoke deadlock more easily, we use only three philosophers. The fork-claiming channels are named according to the philosopher they are connected to – the names of all of the channels are shown in figure 5. We show only the tail-end of the traces (leading up to the deadlock) as the full traces are too long to display here.

3.2.1. CSP

⟨ . . . , fork.right.phil1, fork.left.phil0, fork.left.phil2, fork.right.phil2, fork.right.phil0, fork.left.phil1,
fork.right.phil0, fork.left.phil0, fork.right.phil1, fork.left.phil2, fork.left.phil1, fork.right.phil2, fork.right.phil1,
fork.left.phil0, fork.left.phil2, fork.right.phil0, fork.right.phil2, fork.left.phil1, fork.left.phil0, fork.right.phil0,
fork.right.phil1, fork.left.phil2, fork.left.phil1, fork.right.phil1, fork.right.phil2, fork.left.phil0, fork.left.phil2,
fork.right.phil2, fork.right.phil0, fork.left.phil1, fork.right.phil0, fork.left.phil0, fork.left.phil2, fork.left.phil0 ⟩

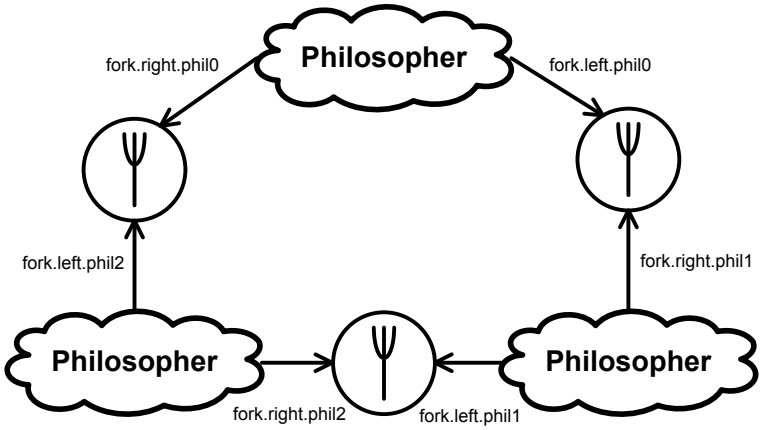

Figure 5. Dining Philosophers with Three Philosophers

3.2.2. VCR

⟨ ..., {fork.left.phil2}, {fork.right.phil0, fork.right.phil2}, {fork.left.phil0, fork.right.phil0, fork.left.phil1}, {fork.right.phil1, fork.left.phil2}, {fork.left.phil1, fork.right.phil1}, {fork.left.phil0, fork.right.phil2}, {fork.left.phil2, fork.right.phil2}, {fork.right.phil0, fork.left.phil1}, {fork.left.phil0, fork.right.phil0}, {fork.right.phil1, fork.left.phil2}, {fork.left.phil1, fork.right.phil1}, {fork.left.phil0, fork.right.phil2}, {fork.left.phil2, fork.right.phil2}, {fork.right.phil0, fork.left.phil1}, {fork.left.phil0}, {fork.right.phil0, fork.right.phil1}, {fork.left.phil1, fork.right.phil1}, {fork.left.phil0, fork.left.phil1, fork.left.phil2} ⟩

3.2.3. Structural

Unfortunately our current implementation of structural tracing in CHP cannot show its traces in the presence of deadlock, due to the fast way the traces are recorded (in local per-process storage) and the way that the error manifests (being caught outside the scope of this storage). We hope to remedy this situation in the near future.

3.2.4. Summary

The dining philosophers problem is an example of a larger process network with less regular behaviour. Another problem is that it is hard to identify which messages are "pick up" messages and which are "put down" messages. An automated tool for dealing with traces could easily fix this by labelling alternating messages on the same channel differently. At the end of each trace it can be seen (most visibly in the VCR trace) that each philosopher communicated to its left-hand fork but not its right-hand fork, immediately before the deadlock.

3.3. Token Cell Ring

This example is a token-passing ring using alting barriers (multiway synchronisations that support choice). The process network is a ring of cells, where each cell is enrolled on two alting barriers (here termed "before" and "after") and has a channel-end from a "tick" process. A cell can have two states: full or empty.

If the cell is full, it offers to either engage on its "after" barrier (to pass the token on) or read from its "tick" channel. If the barrier is the event chosen, the cell then commits to read from its "tick" channel. If the cell is empty, it offers to either engage on its "before" barrier (to receive a token) or read from its "tick" channel. Again, if the barrier is chosen, it then commits to reading from the "tick" channel.

The writing ends of all the tick channels are connected to a "ticker" process that writes in parallel to all its channels, then repeats. Six cells are wired in a ring, and initially the first (index: 0) is full, with the rest empty. All the barriers are named according to the cell

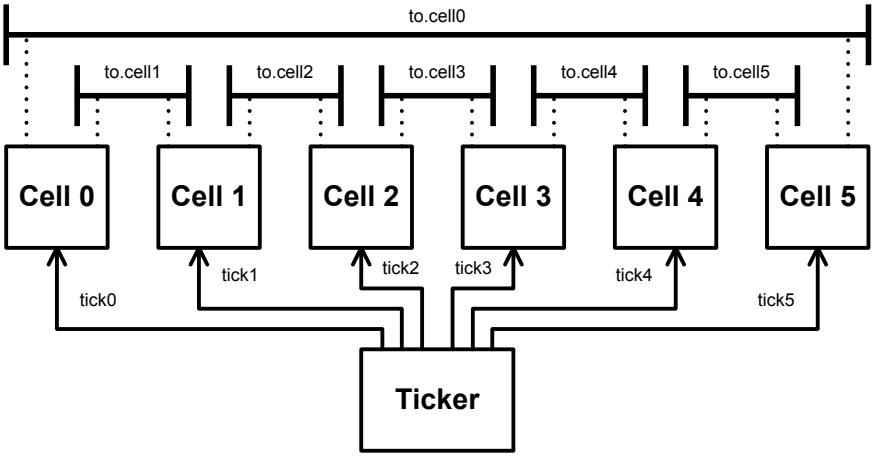

Figure 6. Token Cell Ring Network

following it – that is, the barrier that is "after" for cell 2 and "before" for cell 3 is named "to.cell3".

This network is portrayed in figure 6. The idea is a simple version of the model used for blood clotting in the TUNA project [9]. Eight iterations (i.e. eight ticks) of the network were run.

3.3.1. CSP

⟨ to.cell1, tick0, tick1, tick3, tick4, to.cell2, tick2, tick5,
tick2, tick0, tick1, tick3, tick5, to.cell3, to.cell4, tick4,
to.cell5, tick0, tick2, tick5, tick1, tick3, to.cell0, tick4,
tick0, tick2, tick3, tick5, to.cell1, tick1, to.cell2, tick4,
tick1, tick2, to.cell3, tick4, tick0, tick3, to.cell4, tick5,
tick2, tick3, tick4, tick5, tick0, to.cell5, tick1,
tick0, tick2, tick3, tick5, to.cell0, tick1, tick4,
tick1, tick2, tick4, tick0, tick3, to.cell1, tick5,
tick0, tick1, tick2, tick3, tick4, tick5 ⟩

3.3.2. VCR

⟨ {tick1, tick2, tick4, tick5, to.cell1}, {tick0, tick3, to.cell2},
{tick0, tick1, tick2, tick3, tick4, tick5},
{tick1, tick2, to.cell3}, {tick0, tick3, tick4},
{tick2, tick3, tick5, to.cell4}, {tick0, tick1, tick4, tick5},
{tick2, tick3, tick4, to.cell5}, {tick0, tick1, tick5},
{tick0, tick1, tick2, to.cell0}, {tick3, tick4, tick5, to.cell1},
{tick0, tick1}, {tick2, tick3, tick4, tick5, to.cell2},
{tick0, tick1, tick3, to.cell3}, {tick4, to.cell4}, {tick2, tick5, to.cell5},
{to.cell0}, {tick0}, {tick1}, {tick2}, {tick3}, {tick5}, {tick4} ⟩

3.3.3. Structural

⟨⟨⟨to.cell1* , 4*⟨tick0?⟩ , to.cell0* , tick0? , to.cell1* , 2*⟨tick0?⟩ , to.cell0* , tick0? , to.cell1* , tick0?⟩
|| ⟨to.cell1* , tick1? , to.cell2* , 4*⟨tick1?⟩ , to.cell1* , tick1? , to.cell2* , tick1? , to.cell1* , tick1? , to.cell2* , tick1?⟩
|| ⟨to.cell2* , tick2? , to.cell3* , 4*⟨tick2?⟩ , to.cell2* , tick2? , to.cell3* , 2*⟨tick2?⟩ , to.cell2* , tick2?⟩
|| ⟨8*⟨tick0! || tick1! || tick2! || tick3! || tick4! || tick5!⟩⟩
|| ⟨tick3? , to.cell3* , tick3? , to.cell4* , 4*⟨tick3?⟩ , to.cell3* , tick3? , to.cell4* , 2*⟨tick3?⟩⟩
|| ⟨2*⟨tick4?⟩ , to.cell4* , tick4? , to.cell5* , 3*⟨tick4?⟩ , to.cell4* , tick4? , to.cell5* , 2*⟨tick4?⟩⟩
|| ⟨3*⟨tick5?⟩ , to.cell5* , tick5? , to.cell0* , 2*⟨tick5?⟩ , to.cell5* , tick5? , to.cell0* , 2*⟨tick5?⟩⟩⟩⟩

3.3.4. Summary

We have separated the CSP and VCR traces approximately into the eight iterations of the process network – one per line. It can be seen that each iteration interleaves its events slightly

differently as the token passes along the process pipeline. It is possible to see the movement of the token in each of the different styles of tracing.

3.4. I/O-PAR Example

An I/O-PAR process is one that behaves deterministically and cyclically, by first engaging in all the events of its alphabet, then repeating this behavior. I/O-PAR (and I/O-SEQ) processes have been studied extensively by Welch, Martin, Roscoe and others in [10,11,12,13,14]. Roscoe and Welch separately proved I/O-PAR processes to be deadlock-free, and further that I/O-PAR processes are closed under composition. This proof is not simple, and reasoning about I/O-PAR processes from their traces is not straight-forward.

For an example of a simple IO-PAR network, we use the example originally presented in [15]. One process repeats *a* in parallel with *b* ten times. The other process repeats *b* in parallel with *c* ten times. The two processes are composed in parallel, synchronising on *b* together.

3.4.1. CSP

$$\langle\ a, b, c, c, a, b, a, b, c, a, b, c, a, b, c, a, b, c, a, b, c, a, b, c, a, b, c, a, b, c\ \rangle$$

3.4.2. VCR

$$\langle\ \{a, b, c\}, \{a, b, c\}, \{a, b, c\}, \{a, b, c\}, \{a, b, c\}, \{a, b, c\}, \{a, b, c\}, \{a, b, c\}, \{a, b, c\}, \{a, b, c\}\ \rangle$$

3.4.3. Structural

$$\langle\langle 10{*}\langle a \parallel b\rangle\rangle \parallel \langle 10{*}\langle b \parallel c\rangle\rangle$$

3.4.4. Summary

Given a simple process network, the structural trace again has a direct mapping to the original program. The VCR trace shows the regular parallelism in the system, whereas the CSP trace reveals a slight mis-ordering in the second triple of events. This was predicted in the original paper; two *c* events happen in-between two *b* events in the CSP trace, but this slush is ironed out in the VCR trace.

4. Related Work

While VCR is a model of true concurrency, and the basis for implementing a tracing facility in CHP, it is by no means the only such model. A model of true concurrency is one which does not reduce concurrency to a nondeterministic sequential interleaving of events. A comprehensive survey paper by Cleaveland, et al. [16] discusses models of true concurrency versus interleaving models (such as CSP). Of the models discussed in [16], the earliest example of a model of true concurrency is Petri nets [17]. An introduction to Petri nets can be found in [18]. Kahn nets [19] provide a fixed-point semantics for the concurrency found in dataflow systems. While both are models of true concurrency, neither Petri nets nor Kahn nets are trace-based models. Three trace-based models of true concurrency are discussed in [16], and they are Mazurkiewicz traces [20], pomsets (partially-ordered multisets) [21], and event structures, by Winskel [22]. Mazurkiewicz traces define an independence relation on events to identify potential concurrency in traces of execution, which is in the same spirit of events contained in CHP's event multisets. From a purely model-theoretic standpoint, pomsets and event structures are similar in spirit to VCR's parallel event multisets and ROPEs (randomly ordered parallel events).

The independence relation discussed here is also similar to Lamport's seminal work on the happened-before relation [23]. Lamport defined a partial ordering in time of events in

a system based on causality, inspired by the notion from special relativity that there is no single definitive ordering, and that different observers can disagree on ordering. This is the same idea that inspired VCR's tracing model. Lamport's work was in distributed systems with asynchronous communications. In this paper we have adapted the relation to a hierarchy of parallel processes that communicate synchronously. Our techniques do not inherently prohibit use in a distributed system, but are primarily suited to non-distributed systems.

Unlike much work on clocks (such as vector clocks [24]) in distributed systems, we do not attempt to synchronise time between different processes. Each process has its own process identifier (akin to a local clock), but it is never changed by, or synchronised with, other process identifiers. Our process identifiers reflect the process hierarchy, which bears some resemblance to work on hierarchical vector clocks [25] that tries to have a vector clock per level of the process hierarchy.

5. Future Work

There are several interesting avenues for future work, one of them inspired by two other tools. PRoBE is a formal CSP tool that allows step-by-step interactive exploration of the state space for a CSP program. At each step, one of the next possible events is chosen, and the program proceeds to the next step. The Concurrent Haskell Debugger [26] is a tool designed to visualise and step through Concurrent Haskell programs. It includes the capability to speculatively search ahead in the space of possible execution orderings to try to locate potential deadlocks [27].

We believe that the ideas behind these two systems could be combined, using the approach of the Concurrent Haskell Debugger with the trace recording facilities presented here, to provide programmers with a debugging tool that could present them with traces representing a deadlock in their program, searched for while they run the program.

We have given no consideration in this paper to the CSP notions of event hiding and renaming. Events have been taken to be globally visible, and cannot be renamed. It would be possible to augment the CHP API and trace recording mechanism to allow hiding and renaming of events. For example, in CHP one could write something like the following:

```
(p <||> q) <\\> ["c"]
```

This would represent the parallel composition of processes `p` and `q`, hiding event *c* in the resulting trace. This would be especially applicable to structural traces, because the hiding mechanism would be bound into the structure of the program.

6. Conclusions

We have explained how to implement the recording of CSP, VCR and structural traces, and have shown examples of each. We are not aware of any previous work on recording such traces from CSP implementations, besides the work of Barnes on compiling CSP [28]. We believe that being able to record traces is a useful tool for debugging.

One problem with recording traces that is especially apparent in our dining philosophers and token-cell examples is that traces can be large and difficult to understand. Tools to visualise and analyse traces will definitely be required for large and long-running programs. Process-oriented programming has always supported visual representations of its program layouts, and we hope that this could be integrated with replaying traces.

The work described here has been on the new Communicating Haskell Processes library, but the techniques should apply to any other CSP implementation or language. All that is really required is process-local storage and global shared data protected by a mutex – two

easily available features in most settings. It would be interesting to compare the traces from different implementations of the same program.

The Communicating Haskell Processes library is publicly available under a BSD-like licence. Details on obtaining and using the library can be found at its homepage: `http://www.cs.kent.ac.uk/projects/ofa/chp/`. The tracing facilities are contained in the `Control.Concurrent.CHP.Traces` module.

Acknowledgements

We would like to thank our anonymous reviewers, and also Peter Welch, for their incredibly helpful and detailed comments on this paper.

References

[1] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
[2] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
[3] Peter H. Welch and Fred R. M. Barnes. Communicating mobile processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer Verlag, 2005.
[4] N.C.C. Brown. Communicating Haskell Processes: Composable explicit concurrency using monads. In *Communicating Process Architectures 2008*, September 2008.
[5] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 Manual*. 1997.
[6] Marc L. Smith, Rebecca J. Parsons, and Charles E. Hughes. View-Centric Reasoning for Linda and Tuple Space computation. *IEE Proceedings–Software*, 150(2):71–84, April 2003.
[7] Marc L. Smith. Focusing on traces to link VCR and CSP. In East, Martin, Welch, Duce, and Green, editors, *Communicating Process Architectures 2004*, pages 353–360. IOS Press, September 2004.
[8] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05*, pages 48–60. ACM, 2005.
[9] P.H. Welch, F.R.M. Barnes, and F.A.C. Polack. Communicating complex systems. In Michael G Hinchey, editor, *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS-2006)*, pages 107–117, Stanford, California, August 2006. IEEE. ISBN: 0-7695-2530-X.
[10] P.H. Welch. Emulating Digital Logic using Transputer Networks (Very High Parallelism = Simplicity = Performance). *International Journal of Parallel Computing*, 9, January 1989. North-Holland.
[11] P.H. Welch, G.R.R. Justo, and C.J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In R. Grebe, J. Hektor, S.C. Hilton, M.R. Jane, and P.H. Welch, editors, *Transputer Applications and Systems '93, Proceedings of the 1993 World Transputer Congress*, volume 2, pages 981–1004, Aachen, Germany, September 1993. IOS Press, Netherlands. ISBN 90-5199-140-1.
[12] J.M.R. Martin, I. East, and S. Jassim. Design Rules for Deadlock Freedom. *Transputer Communications*, 3(2):121–133, September 1994. John Wiley and Sons. 1070-454X.
[13] J.M.R. Martin and P.H. Welch. A Design Strategy for Deadlock-Free Concurrent Systems. *Transputer Communications*, 3(4):215–232, October 1996. John Wiley and Sons. 1070-454X.
[14] A. W. Roscoe and Naiem Dathi. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289–327, December 1987.
[15] Mark Burgin and Marc L. Smith. Compositions of concurrent processes. In F.R.M. Barnes, J.M. Kerridge, and P.H. Welch, editors, *Communicating Process Architectures 2006*, pages 281–296. IOS Press, September 2006.
[16] Rance Cleaveland and Scott A. Smolka. Strategic directions in concurrency research. *ACM Computing Surveys*, 28(4), January 1996.
[17] C. A. Petri. Kommunikation mit automaten. Technical report, Schriften des IIm 2, Institut fur Instrumentelle Mathematik, Bonn, 1962.
[18] W. Reisig. *Petri Nets—An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, New York, 1985.
[19] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74*, North-Holland, Amsterdam, 1974.
[20] A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets, 1986, Part II; Pro-*

ceedings of an Advanced Course (Bad Honnef, Sept.), volume 255 of *Lecture Notes in Computer Science*, pages 279–324, Berlin, 1987.

[21] V. R. Pratt. Modeling concurrency with partial orders. *Int. J. Parallel Program.*, 15(1):33–71, 1986.

[22] G. Winskel. An introduction to event structures. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *REX School and Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354, pages 364–397, New York, 1989. Springer-Verlag.

[23] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[24] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference (ACSC'88)*, pages 56–66, February 1988.

[25] D.A. Khotimsky and I.A. Zhuklinets. Hierarchical vector clock: Scalable plausible clock for detecting causality in large distributed systems. In *Proc. 2nd Int. Conf. on ATM, ICATM'99*, pages 156–163, 1999.

[26] Thomas Böttcher and Frank Huch. A debugger for concurrent haskell. In *Implementation of Functional Languages 2002*, pages 129–141, 2002. Draft Proceedings.

[27] Jan Christiansen and Frank Huch. Searching for deadlocks while debugging concurrent haskell programs. *SIGPLAN Not.*, 39(9):28–39, 2004.

[28] F.R.M. Barnes. Compiling CSP. In P.H. Welch, J. Kerridge, and F.R.M. Barnes, editors, *Communicating Process Architectures 2006*, pages 377–388. IOS Press, September 2006. ISBN: 1-58603-671-8.