Communicating Process Architectures 2009
Peter Welch, Herman Roebbers and Tobe Announced (Eds.)
IOS Press, 2009

Relating and Visualising CSP, VCR and Structural 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. As well as being a useful tool for formal reasoning, a trace can provide insight into a concurrent program's behaviour, especially for the purposes of run-time analysis and debugging. Long-running programs tend to produce large traces which can be difficult to comprehend and visualise. We examine the relationship between three types of traces (CSP, VCR and Structural), establish an ordering and describe methods for conversion between the trace types. Structural traces preserve the structure of composition and reveal the repetition of individual processes, and are thus well-suited to visualisation. We introduce the Starving Philosophers to motivate the value of Structural traces for reasoning about behaviour not easily predicted from a program's specification. A remaining challenge is to integrate Structural traces into a more formal setting, such as the Unifying Theories of Programming – however, Structural traces do provide a useful framework for analysing large systems.

Keywords. Traces, CSP, VCR, Structural Traces, Visualisation

Introduction

Hoare and Roscoe's Communicating Sequential Processes (CSP) [1,2] is most well-known as a process calculus (or process algebra) for describing and reasoning about concurrent systems. As its name suggests, a concurrent system may be viewed as a composition of individual, communicating sequential processes. One of Hoare and Roscoe's big ideas was that one may reason about a computation by reasoning about its history of observable events. This history must be recorded by someone, and so CSP introduces the notion of an observer to perform this task. Thus, CSP introduced traces as a means of representing a process's computational history. The idea of observable events in concurrent computation is an extension of the idea from computability theory that one may observe only the input/output behaviour of processes when reasoning about properties of that process (e.g., will a given process halt on its input?). Communication (between two processes, or between a process and its environment) is another form of input/output behaviour, and therefore observable events are a reasonable basis for recording traces of computations.

In mathematics, algebra is the study of equations, and calculus is the study of change. How do these areas of mathematics relate to processes? It follows that a process algebra provides a means for describing processes as equations, and a process calculus provides a means for reasoning about changes in a system as computation proceeds. CSP provides a rich process algebra that is used for process specification, and a process calculus based on possible traces of a process's specification to help prove properties of a concurrent system. Cast in terms of change, a trace represents a system's computational progress, or the changes in the state of a system during its computation. Modern CSP [2] considers more than just traces in

its process calculus (i.e., possible failures, divergences, and refinements of a process), though in this paper we are concerned with investigating what further benefits traces can provide, not in the context of specification, but in the context of debugging large systems. The remainder of this introduction discusses the three different types of traces, a motivating example of distributed systems and the use of traces in debugging.

CSP, VCR and Structural Traces

In previous work [3], the authors extended the notion of what it means to record a trace, and describe how the shapes of these differently recorded traces extend the original definition of a CSP trace. In this section, we briefly summarize this previous work.

CSP traces are sequentially interleaved representations of concurrent computations. One way to characterize a CSP trace is that it abstracts away time and space from a concurrent computation. Due to interleaving in the trace, one cannot in general see a followed by b in the trace and know from the trace alone whether event a actually occurred long before event b, or whether they were observed at the same time and ordered arbitrarily in the trace. Likewise, if process P and process Q are both capable of engaging in event a, one cannot tell from the trace alone whether it was process P or process Q that engaged in event a, or whether a represents a synchronizing event for processes P and Q.

VCR traces support View-Centric Reasoning [4], provide for multiple observers, and a means for representing multiple views of a computation. Rather than a sequence of events, a VCR trace is a sequence of event multisets, where the multisets represent independent events. Independent events are events that may be nondeterministically interleaved by the CSP observer. When new events are observed for a VCR trace, they are added to a new multiset iff a dependence relationship can be inferred between the new event and the events in the current multiset [3]. If this dependency cannot be inferred, the event is added to the current multiset in the trace. In other words, a VCR trace preserves some of the timing information of events, but still abstracts away space from a concurrent computation (i.e., what process engaged in each of the events). A VCR trace permits reasoning about all the possible interleavings of a particular computation, but does not preserve which process engaged in which event.

Structural traces provide not only for multiple observers, but a means for associating those observers with the processes themselves! As the name suggests, Structural traces reflect the structure of a concurrent system's composition. That is, Structural traces abstract away *neither* time nor space. A parallel composition of CSP processes is recorded as a parallel composition of the respective traces in the Structural trace. Each process has a local observer who records a corresponding trace, for example the trace of a process P. If P contains a parallel composition of processes Q and R, then after both Q and R have completed, P's observer records the traces of Q and R as a parallel composition in P's trace. Besides the extra structure, the primary difference between Structural traces and other types is that Structural traces record an event synchronisation multiple times – once for each participant.

The grammar of the three different trace types discussed above is given in figure 1. As an example, given the CSP system:

$$(a \rightarrow \text{SKIP} \;|||\; b \rightarrow \text{SKIP}) \rightarrow (c \rightarrow \text{SKIP} \underset{\{c\}}{||} c \rightarrow \text{SKIP})$$

The possible traces are shown below:

CSP:	$\langle a, b, c\rangle$ or $\langle b, a, c\rangle$
VCR:	$\langle \{a, b\}, \{c\}\rangle$
Structural:	$(a \;\|\|\; b) \rightarrow (c \;\|\|\; c)$

Note that the synchronising event c occurs twice in the parallel composition in the Structural trace.

$$CSPTRACE ::= \underline{\langle\rangle} \mid \underline{\langle}\ EVENT\ (\underline{,}\ EVENT)^*\ \underline{\rangle} \tag{1}$$

$$VCRTRACE ::= \underline{\langle\rangle} \mid \underline{\langle}\ EVENTBAG\ (\underline{,}\ EVENTBAG)^*\ \underline{\rangle} \tag{2}$$

$$EVENTBAG ::= \underline{\{}\ EVENT\ (\underline{,}\ EVENT)^*\ \underline{\}} \tag{3}$$

$$STRUCTURALTRACE ::= \underline{()} \mid SEQ \tag{4}$$

$$SEQ ::= ((EVENT \mid PAR)\ (\underline{\rightarrow}\ SEQ)^?) \mid (NATURAL\ \underline{*}\ SEQ) \tag{5}$$

$$PAR ::= SEQ\ \underline{\|}\ SEQ\ (\underline{\|}\ SEQ)^* \tag{6}$$

Figure 1. The grammar for CSP traces (line 1), VCR traces (lines 2–3) and Structural traces (lines 4–6). Literals are underlined. A superscript '*' represents repetition zero or more times, and a superscript '?' represents repetition zero or one times (i.e. an optional item).

Distributed Systems

We can consider the differences between the three trace types with a physical analogy. Physics tells us that light has a finite speed. We can imagine a galaxy of planets, light-years apart, observing each other's visible transmissions. CSP calls for an omnipotent, Olympian observer that is able to observe all events in the galaxy and order them correctly (aside from events that appear to have occurred simultaneously) – a strong requirement! VCR's multisets of independent events allow us to capture some of the haziness of observation, by leaving events independent unless a definite dependence can be observed. Structural traces allow for each planet to observe its own events, thus removing any concerns about accuracy or observation delay. The trace of the whole galaxy is formed by assembling together, compositionally, the traces of the different planets.

This analogy may appear somewhat abstract, but it maps directly to the practical problem of recording traces for distributed systems, where each machine has internal concurrency and communication, as well as the parallelism and communication of the multiple machines. Implementing CSP's Olympian observer in a distributed system, that could accurately observe all internal machine events (from multiple machines) and networked events, is not feasible. The CSP observer must be present on one of the machines, rendering its observation of other machines inaccurate. The system could perhaps be implemented using synchronised high-resolution timers (and a later merging of traces) but what VCR and Structural traces capture is that the ordering of some events does not matter. If machine A and B communicate while C and D also communicate, with no interaction between the two pairs, we can deduce that the ordering of the A-B and C-D communications was arbitrary as they could have occurred in either order. In VCR terms, they are independent events.

Trace Investigation

We have already stated that CSP enables proof using traces (among other models) – should your program already be formally modelled, the need for further analysis is perhaps unclear. Two possible uses for trace investigation are given in this section.

CSP offers both external choice (where the environment decides between several offered events) and internal choice (where the process itself decides which to offer). There is sometimes an implicit intention that the program be roughly fair in the choice. For example, if a server is servicing requests from two clients, neither client should be starved of attention from

the server. This may even be a qualitative judgement; hence human examination of the traces of a system (that capture its behaviour) may be needed. We give an example of starvation in section 2.2.

Another reason is that the program may have capacity for error handling. For example, some events may be offered by a server (to prevent deadlock), but for well-formed messages from a remote client, these events should not occur. A spate of badly-formed messages from a client may be cause for investigation, and a trace can aid in this respect.

Organisation

It turns out that in some senses a Structural trace generalizes both VCR and CSP traces, which we discuss in section 1. The generality of Structural traces is important for two reasons. First, the CHP library [5] is capable of producing all three types of traces of a Haskell program, but producing Structural traces is more efficient than producing the other two trace types. (As previously discussed by the authors [3], these traces are different from merely adding print statements to a program in that recording traces does not change the behaviour of the program.) From a Structural trace, one can generate equivalent VCR or CSP traces (we give the algorithms in section 1). Second, Structural traces provide a framework for visualisation of concurrent programs, which we discuss in section 2, and argue how trace visualisation is helpful for debugging. In section 3 we discuss our progress on characterizing Structural traces more formally, and the challenges that remain.

1. Conversion

Each trace type is a different representation of an execution of a system. It is possible to convert between some of the representations. One VCR trace can be converted to many CSP traces (see section 1.1). A VCR trace can thus be considered to be a particular equivalence class of CSP traces. One Structural trace can be interleaved in different ways to form many VCR traces and (either directly or transitively) many CSP traces (see section 1.2). A Structural trace can also be considered to be a equivalence class of VCR (or CSP) traces. This gives us an ordering on our trace representations: one Structural trace can form many VCR traces, which can in turn form many CSP traces. These conversions are deterministic and have a finite domain. In general, conversions in the opposite direction (e.g., CSP to Structural) can be non-deterministic and infinite – for example, the CSP trace $\langle a \rangle$ could have been generated by an infinite set of Structural traces: a_0, or $(a_0 \mid\mid a_0)$, or $(a_0 \mid\mid a_0 \mid\mid a_0)$, and so on.

1.1. Converting VCR Traces to CSP Traces

One VCR trace can be transformed to *many* CSP traces. A VCR trace is a sequence of multisets; by forming all permutations of the different multisets and concatenating them, it is possible to generate the corresponding set of all possible CSP traces. A Haskell function for performing the conversion is given in figure 2 – an example of this conversion is:

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

Note that the number of resulting CSP traces is easy to calculate: for all non-empty VCR traces tr, the identity $length\ (vcrToCSP\ tr) == foldl1\ (*)\ (map\ Set.size\ tr)$ holds. That is, the number of generated CSP traces is equivalent to the product of the sizes of each set in the VCR trace.

```
type CSPTrace = [Event]
type VCRTrace = [Set Event] -- All sets must be non-empty.

cartesianProduct :: [a] -> [b] -> [(a, b)]
cartesianProduct xs ys = [ (x, y) | x <- xs, y <- ys ]

vcrToCSP :: VCRTrace -> Set CSPTrace
vcrToCSP [] = singleton []
vcrToCSP (s : ss) =
  fromList [ a ++ b | (a, b) <- cartesianProduct (permutations (toList s)) (toList (vcrToCSP ss))]
```

Figure 2. An algorithm for converting a VCR trace into a set of CSP traces.

1.2. Converting Structural Traces to VCR or CSP Traces

Our originally recorded Structural traces [3] could not be converted into CSP or VCR traces *post hoc* because it was not clear how events "lined up". Consider the Structural trace:

$$(a \rightarrow a) \;||\; a$$

It is not clear which (if either!) of the events from the LHS of the parallel composition are the same synchronisation as the RHS. Therefore it is not clear whether this should become the CSP trace $\langle a, a, a \rangle$ or $\langle a, a \rangle$.

A Structural trace can be converted into CSP or VCR traces if we record a little extra information. Specifically, we must record a sequence number with each communication event in the trace. In CSP terms, we effectively replace all synchronisations on each event a with a corresponding external choice:

$$(a \rightarrow P) \mapsto (\Box\, A \rightarrow P) \qquad \text{where } A = \{a_i \mid i \in \mathbb{N}_0\}$$

All uses of the event a in sets for hiding and parallel composition must also be replaced by the events in A. We must then compose our existing system P in parallel with a new process for that event:

$$P \mapsto (P \underset{A}{||} SEQ_{a,0}) \qquad \text{where } SEQ_{a,i} = a_i \rightarrow SEQ_{a,i+1}$$

In the implementation, the sequence number becomes part of the data structure underlying channels and barriers, and adds no significant overhead to the cost of communications.

With this sequence number, it is possible to match up the communications from different parts of the Structural trace. Our earlier example would become $(a_0 \rightarrow a_1) \;||\; a_0$, if the first synchronisation on the LHS was the one featured on the RHS. Furthermore, the format of the Structural trace means that it was already possible to derive the process identifiers needed to form the VCR trace. These two aspects combined will allow us to generate CSP and VCR traces from Structural traces.

1.2.1. Algorithm Description

We define the algorithm for converting Structural traces to CSP traces in Haskell, in figure 3. First, we define our data structures, based around an *Event* type with hidden definition (see figure 3, lines 1–7). A CSP trace is a list of events; a Structural trace is either empty or a sequential trace. A sequential trace is an event synchronisation (an event identifier paired with a sequence identifier) or a parallel trace, followed by an optional further sequential

trace in a *cons*-fashion. A parallel trace is a collection of two or more sequential traces. This corresponds to the grammar in figure 1.

To convert a Structural trace, we must first know how many processes were involved in each synchronisation. We thus define a *prSeq* function that builds up a map from an event and sequence identifier pair to the number of processes that synchronised on that event (see figure 3, lines 9–16).

The remainder of our algorithm can be implemented in a continuation-like style. The Structural trace is explored, and a map is built up with all the events initially available (and the number of processes available to engage in them), as well as a function that, given an occurred event, will return the next map and function. This is done by the *convSeq* function (see figure 3, lines 18–36).

The final piece is a function that iteratively picks the next available event from the merged maps and records it in a new CSP trace, until the trace is complete (see figure 3, lines 38–49). This picking of available events and then continuing the trace is strongly reminiscent of the execution of the CSP program. Effectively, our algorithm is the environment, picking arbitrarily from the available set of events and then continuing the computation. Of course, our Structural trace differs from a typical CSP system in that it is finite, lacks any choice, and by definition our replaying of a real Structural trace is deadlock-free.

1.2.2. Example

We give here an example of converting a Structural trace to its CSP and VCR forms:

$$
\begin{aligned}
&(a_0 \rightarrow b_0 \rightarrow b_1 \rightarrow a_1) \;||\; (c_0 \rightarrow b_0 \rightarrow (c_1 \;||\; d_0) \rightarrow b_1) \\
&\qquad \mapsto \{\langle a, c, b, c, d, b, a\rangle \\
&\qquad\quad , \langle a, c, b, d, c, b, a\rangle \\
&\qquad\quad , \langle c, a, b, c, d, b, a\rangle \\
&\qquad\quad , \langle c, a, b, d, c, b, a\rangle\} \\
&(a_0 \rightarrow b_0 \rightarrow b_1 \rightarrow a_1) \;||\; (c_0 \rightarrow b_0 \rightarrow (c_1 \;||\; d_0) \rightarrow b_1) \\
&\qquad \mapsto \{\langle \{a, c\}, \{b\}, \{c, d\}, \{b\}, \{a\}\rangle\}
\end{aligned}
$$

It can be seen that converting the set of CSP traces is the same as would be produced by converting the VCR trace into CSP traces.

1.2.3. Generalising and VCR Algorithm

Although our algorithm creates one trace through picking arbitrarily, it would be trivial to modify it to generate all possible traces through exploring all options. From a more abstract perspective, the set of all traces is of interest: generating an arbitrary trace is a needless specialisation. From a practical perspective, the set of all traces may be overwhelming and thus it may be easier to explore an arbitrary single converted trace.

The algorithm for converting Structural traces to VCR traces is a combination of the algorithm given here for converting Structural to CSP traces, and the pre-existing algorithm for recording VCR traces based on process identifiers [3]. One interesting aspect is the choice of available events. In our conversion to CSP, we pick arbitrarily (figure 3, line 49). In VCR, the choice (if we only wish to generate one trace) makes a noticeable difference to the resulting trace. Consider the Structural trace:

$$(a \rightarrow b) \;||\; c$$

```
1  type CSPTrace = [Event]
2
3  type SeqId = Integer
4  data StructuralSeq = Seq (Either (Event, SeqId) StructuralPar) (Maybe StructuralSeq)
5  data StructuralPar = Par StructuralSeq StructuralSeq [StructuralSeq]
6  data StructuralTrace = TEmpty
7                       | T StructuralSeq
8
9  combine :: [Map (Event, SeqId) Integer] -> Map (Event, SeqId) Integer
10 combine = foldl (Map.unionWith (+)) Map.empty
11
12 prSeq :: StructuralSeq -> Map (Event, SeqId) Integer
13 prSeq (Seq (Left e) s)
14   = combine [Map.singleton e 1, maybe Map.empty prSeq s]
15 prSeq (Seq (Right (Par sA sB ss)) s)
16   = combine (maybe Map.empty prSeq s : prSeq sA : prSeq sB : map prSeq ss)
17
18 data Cont = Cont (Map (Event, SeqId) Integer) ((Event, SeqId) -> Cont)
19           | ContDone
20
21 convSeq :: StructuralSeq -> Cont
22 convSeq (Seq (Left e) s) = c
23   where
24     c = Cont (Map.singleton e 1)
25              (\e' -> if e /= e' then c else maybe ContDone convSeq s)
26 convSeq (Seq (Right (Par sA sB ss)) s)
27   = merge (convSeq sA : convSeq sB : map convSeq ss) `andThen` maybe ContDone convSeq s
28
29 andThen :: Cont -> Cont -> Cont
30 ContDone `andThen` r = r
31 Cont m f `andThen` r = Cont m (\e -> f e `andThen` r)
32
33 merge :: [Cont] -> Cont
34 merge cs = case [ m | Cont m f <- cs] of
35               [] -> ContDone
36               ms -> Cont (combine ms) (\e -> merge [f e | Cont m f <- cs])
37
38 structuralToCSP :: StructuralTrace -> CSPTrace
39 structuralToCSP TEmpty = []
40 structuralToCSP (T s) = iterate (convSeq s)
41   where
42     participants = prSeq s
43
44     iterate :: Cont -> CSPTrace
45     iterate ContDone = []
46     iterate (Cont m f) = fst e : iterate (f e)
47       where
48         es = Map.filter (\(n, n') -> n == n') (Map.intersectionWith (,) m participants)
49         e = Map.findMin es -- Arbitrary pick
```

Figure 3. The algorithm for converting Structural traces to CSP traces.

Our first choice may be a, giving us the partial VCR trace $\langle\{a\}\rangle$. If our next choice is b, the VCR trace becomes $\langle\{a\},\{b\}\rangle$, but if our next choice is c, the VCR trace becomes $\langle\{a,c\}\rangle$. We could favour the second choice if we wanted more independence visible in the trace – which may aid understanding. We emphasise again that this is a moot point for generating the set of *all* traces, but we believe that for single traces, a trace with maximal

obvious independence (i.e. fewer but larger sets in the VCR trace) will be easier to follow.

1.3. Practical Implications

In our previous work [3] we discussed implementing the recording of three different types of traces: CSP, VCR and Structural. A CSP trace is most straightforward to record, by using a mutex-protected sequence of events as a trace for the whole program. A VCR trace uses a mutex-protected data structure with additional process identifiers. Both of these recording mechanisms are inefficient and do not scale well, due to the single mutex-protected trace. With four, eight or more cores, the contention for the mutex may cause the program to slow down or alter its execution behaviour. This deficiency is not present with Structural traces, which are recorded locally without a lock, and thus scale perfectly to more cores.

We could also consider recording the traces of a distributed system. Implementing CSP's Olympian observer on a distributed system (a parallel system with delayed messaging between machines) is a very challenging task. VCR's concepts of imperfect observation and multiple observers would allow for a different style of observation. Structural traces could be recorded on different machines without modification to the recording strategy, and merged afterwards just as they normally are for parallel composition.

The Structural trace should also be easier to compress, due to regularity that is present in branches of the trace, but that is not present in the arbitrary interleavings of a CSP trace. Given that the Structural trace is more efficient to record in terms of time and space (memory requirements), supports distributed systems well and can be converted to the other trace types *post hoc*, this makes a strong case for recording a Structural trace rather than CSP or VCR.

2. Visual Traces

It is possible to visualise the different traces in an attempt to gain more understanding of the execution that generated the trace. For an example, we will use P:

$$P = (Q \fatsemi Q) \underset{\{b\}}{||} ((b \rightarrow b \rightarrow \text{SKIP}) \,|||\, (c \rightarrow c \rightarrow \text{SKIP}))$$

$$\text{where } Q = (a \rightarrow \text{SKIP}) \,|||\, (b \rightarrow \text{SKIP})$$

One possible CSP trace of this system is:

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

This trace is depicted in figure 4a. It can be seen that for CSP traces, this direct visualisation offers no benefits over the original trace. As an alternative, the trace is also depicted in figure 4b. This diagram is visually similar to a finite state automata, with each event occurrence being a state, and sequentially numbered edges showing the paths from event to event – but these edges must be followed in ascending order.

A possible VCR trace of this system is:

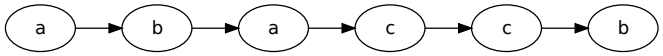
$$\langle \{a, b\}, \{a, c\}, \{c, b\} \rangle$$

This trace is depicted straightforwardly in figure 4c.

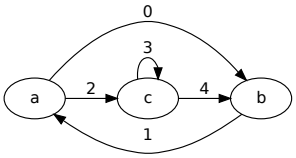
The Structural trace of this system, with event sequence identifiers, is:

$$((a_0 \,||\, b_0) \rightarrow (a_1 \,||\, b_1)) \,||\, ((b_0 \rightarrow b_1) \,||\, (c_0 \rightarrow c_1))$$

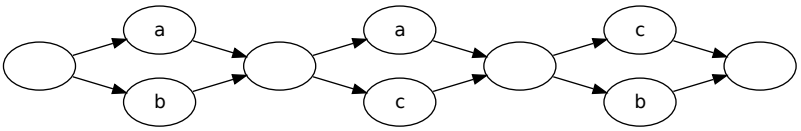
A straightforward depiction is given in figure 4d. An alternative is to merge together the nodes that represent the same event and sequence number, and use edges with thread-identifiers to join them together: this is depicted in figure 4e.



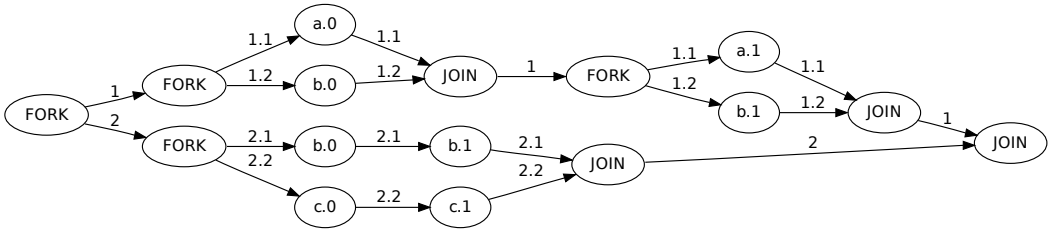
(a) A simple rendering of the CSP trace: $\langle a, b, a, c, c, b \rangle$.



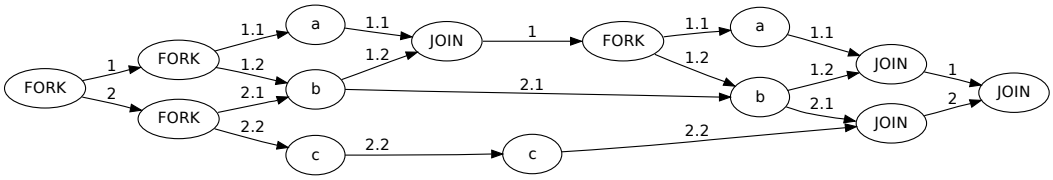
(b) A more complicated rendering of the CSP trace from (a): event synchronisations are rendered as state nodes, and ascending transition identifiers label the edges.



(c) A simple rendering of the VCR trace: $\langle \{a, b\}, \{a, c\}, \{c, b\} \rangle$



(d) A rendering of the Structural trace: $((a_0 \parallel b_0) \rightarrow (a_1 \parallel b_1)) \parallel ((b_0 \rightarrow b_1) \parallel (c_0 \rightarrow c_1))$. Event synchronisations are represented by nodes with the same name, and the edges are labelled threads of execution joining together the event synchronisations.



(e) A rendering of the Structural trace: $((a_0 \parallel b_0) \rightarrow (a_1 \parallel b_1)) \parallel ((b_0 \rightarrow b_1) \parallel (c_0 \rightarrow c_1))$. Event synchronisations are represented by a single node, and the edges are labelled threads of execution joining together the event synchronisations.

Figure 4. Various graphical representations of traces that could be produced by the CSP system $P = (Q \underset{\{b\}}{\parallel} ((b \rightarrow b \rightarrow \text{SKIP}) \mid\mid\mid (c \rightarrow c \rightarrow \text{SKIP}))$ where $Q = (a \rightarrow \text{SKIP}) \mid\mid\mid (b \rightarrow \text{SKIP})$.

2.1. Discussion of Visual Traces

Merely graphing the traces directly adds little power to the traces (figures 4a, 4c and 4d). The contrast between the CSP, VCR and Structural trace diagrams is interesting. The elegance and simplicity of the CSP trace (figure 4a) can be contrasted with the more information

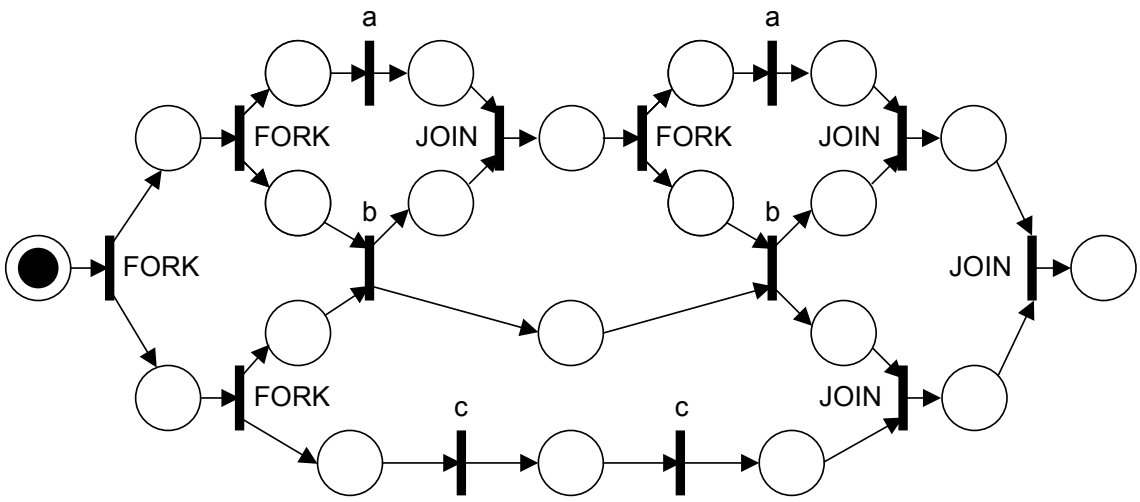


Figure 5. A petri net formed from figure 4e by using the nodes as transitions and the edges as places (unlabelled in this graph). Being from a trace, the petri net is finite and non-looping; it terminates succesfully when a token reaches the right-most place.

provided by the VCR trace of independence (figure 4c) and the more complex structure of the Structural trace (figure 4d).

The merged version of the Structural trace (figure 4e) is a simple change from the original (figure 4d), but one that enables the reader to see where the different threads of execution "meet" as part of a synchronisation.

Figure 4e strongly resembles the *dual* of a graphical representation of a Petri net [6], a model of true concurrency. The nodes in our figure (FORK, JOIN and events) are effectively transitions in a standard Petri net, and the edges are places in a standard Petri net. Thus we can mechanically form a Petri net equivalent, as shown in figure 5.

Figure 4e is also interesting for its relation to VCR traces. Our definition of dependent events [3] is visually evident in this style of graph. An event *b* is dependent on *a* iff a path can be formed in the directed graph from *a* to *b*. Thus, in figure 4e, it can be seen that the *c* events are mutually independent of the *a* and *b* events, and that the right-most *a* event is dependent on the left-most *b* event, and so on.

2.2. The Starving Philosophers

CSP models (coupled with tools such as FDR) allow formally-specified properties to be proved about programs: for example, freedom from deadlock, or satisfaction of a specification. Some properties of a system can be difficult to capture in this way – we present here an example involving starvation, based on Peter Welch's "Wot, No Chickens?" example [7].

Our example involves three philosophers. All the philosophers repeatedly attempt to eat chicken (no time for thinking in this example!). They are coupled with a chef, who can serve two portions of chicken at a time. In pseudo-CSP[1], our example is:

[1]The ampersand denotes conjunction [8]: the process $a \,\&\, b$ is an indivisible synchronisation on *a* and *b*.

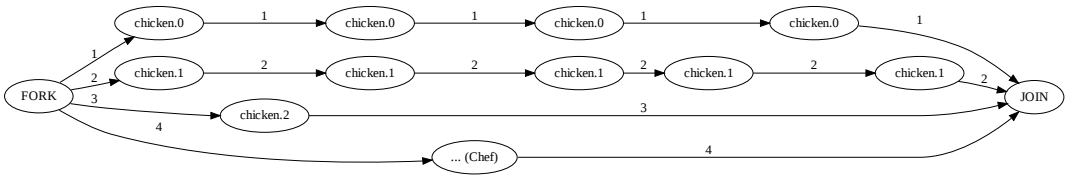


Figure 6. An example Structural trace of five iterations of the starving philosophers example. The chef is collapsed, as it could be in an interactive visualisation of the trace. It is clear from this zoomed out view that one philosopher (shown in the third row) has far fewer synchronisations than the other two, revealing that it is being starved.

$$PHILOSOPHER(n) = chicken.n \rightarrow PHILOSOPHER(n)$$

$$CHEF = ((chicken.0 \;\&\; chicken.1) \;\Box\; (chicken.0 \;\&\; chicken.2) \;\Box\; (chicken.1 \;\&\; chicken.2))$$

$$\S\, CHEF$$

$$SYSTEM = (PHILOSOPHER(0) \;|||\; PHILOSOPHER(1) \;|||\; PHILOSOPHER(2)) \;\underset{\{chicken.0, chicken.1, chicken.2\}}{||}\; CHEF$$

Our intention is that the three philosophers should eat with roughly the same frequency. However, the direct implementation of this system using the CHP library [5] leads to starvation of the third philosopher. Adding additional philosophers does not alleviate the problem: the new philosophers would also starve. In the CHP implementation, the chef always chooses the first option (to feed the first two philosophers) when multiple options are ready. This is a behaviour that is allowed by the CSP specification but that nevertheless we would like to avoid. Attempting to ensure that the three philosophers will eat with approximately the same frequency is difficult formally, but by performing diagnostics on the system we can see if it is happening in a given implementation. By recording the trace and visualising it (as shown in the Structural trace in figure 6) we are able to readily spot this behaviour and attempt to remedy it.

2.3. Interaction

The diagrams already presented are static graphs of the traces. One advantage of the Structural traces is that they preserve, to some degree, the organisation of the original program. The amenability of process-oriented programs to visualisation and visual interaction has long been noted – Simpson and Jacobsen provide a comprehensive review of previous work in this area [9]. The tools reviewed all focused on the design of programs, but we could use similar ideas to provide tools for interpreting traces of programs. We wish to support examination of *one* trace of a program, in contrast to the exploration of all traces of a program that tools such as PRoBE provide [2].

One example of an interactive user interface is shown in figure 8. This borrows heavily from existing designs for process-oriented design tools – and indeed, it would be possible to display the code for the program alongside the trace in the lower panel. This could provide a powerful integrated development environment by merging the code (what can happen) with the trace (what did happen).

An alternative example is given in figure 9. This shows the process hierarchy vertically, rather than the previous nesting strategy. Synchronising events are shown horizontally, connecting the different parts of parallel compositions. What this view makes explicit is that for any given event, there is an expansion level in the tree such that all uses of that event are

$$\text{CommsTime} = (\text{numbers}(d) \underset{\{d\}}{||} \text{recorder}(d)) \setminus \{d\}$$

$$\text{numbers}(out) = ((\text{delta}(b, c, out) \underset{\{b\}}{||} \text{prefix}(a, b)) \underset{\{a,c\}}{||} \text{succ}(c, a)) \setminus \{a, b, c\}$$

$$\text{delta}(in, outA, outB) = in?x \rightarrow (outA!x \rightarrow \text{SKIP} \;|||\; outB!x \rightarrow \text{SKIP}) \;\underset{9}{\circ}\; \text{delta}(in, outA, outB)$$

$$\text{prefix}(in, out) = out!0 \rightarrow \text{id}(in, out)$$

$$\text{id}(in, out) = in?x \rightarrow out!x \rightarrow \text{id}(in, out)$$

$$\text{succ}(in, out) = in?x \rightarrow out!(x + 1) \rightarrow \text{succ}(in, out)$$

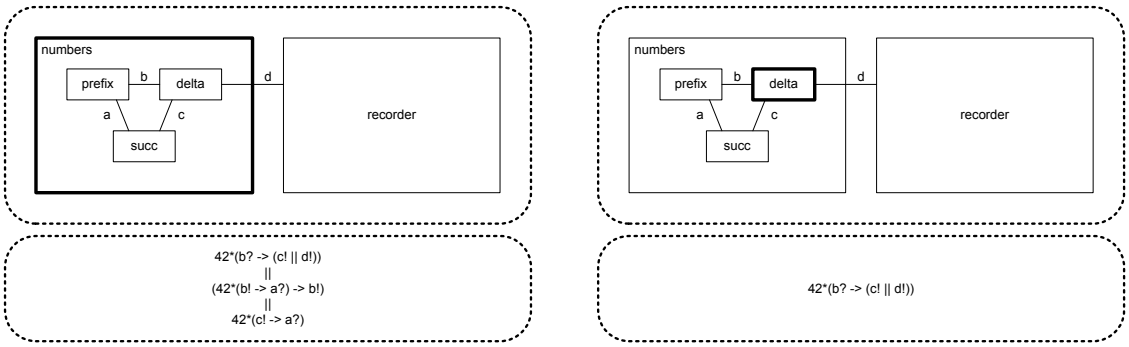
$$\text{recorder}(in) = in?x \rightarrow \text{recorder}(in)$$

(a) The CSP specification for the CommsTime network, *CommsTime*.

$$\begin{aligned}(&\text{numbers:} \\ &\quad (\text{delta:} 42 * (b? \rightarrow (c! \;||\; d!))) \\ &||\; (\text{prefix:} (42 * (b! \rightarrow a?) \rightarrow b!)) \\ &||\; (\text{succ:} 42 * (c! \rightarrow a?)) \\ &||\; (\text{recorder:} 42 * d?))\end{aligned}$$

(b) The Structural trace for the CommsTime network. Note that the labels come from code annotations.

Figure 7. The specification and trace of the CommsTime example network. Note that the hiding of events – important for the formal specification – is disregarded for recording the trace in our implementation in order to provide maximum information about the system's behaviour.



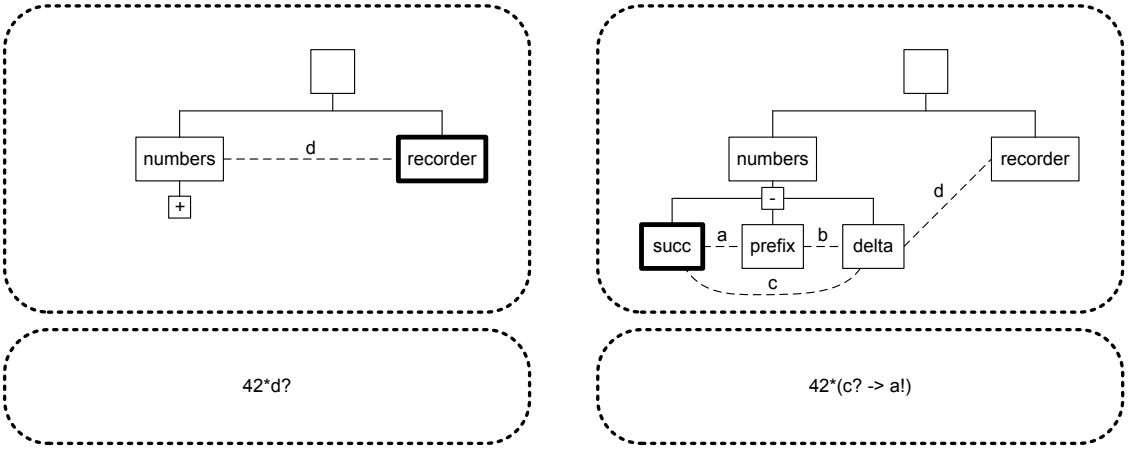
(a) The selected process is numbers. (b) The selected process is delta.

Figure 8. An example user interface for exploring the trace (from figure 7) of a process network. The top panel displays the process network, with its compositionality reflected in the nesting. Processes can be selected, and their trace displayed in the bottom panel. Inner processes could be hidden when a process is not selected (for example, recorder could have internal concurrency that is not displayed while it is not being examined).

solely contained with one process. For *a*, *b* and *c*, this process is numbers; for *d* it is the root of the tree.

3. Relations and Operations on Structural Traces

In their Unifying Theories of Programming [10], Hoare and He identified theories of programming by sets of healthiness conditions that they satisfy. Many of these healthiness conditions involve the following relations and operations on traces: prefix ($\leq$), concatenation ($\hat{}$) and quotient ($-$). For CSP the definitions are obvious. Prefix is the sequence-prefix relation,



(a) The recorder process is selected. (b) The succ process is selected.

Figure 9. An example user interface for exploring the trace (from figure 7) of a process network. The top panel shows a tree-like view for the process network. Processes can be selected and their trace displayed in the bottom panel. Notably, the internal concurrency of processes (and the associated events) can be hidden, as in the left-hand example for numbers. On the right-hand side, the numbers process is expanded to reveal the inner concurrency.

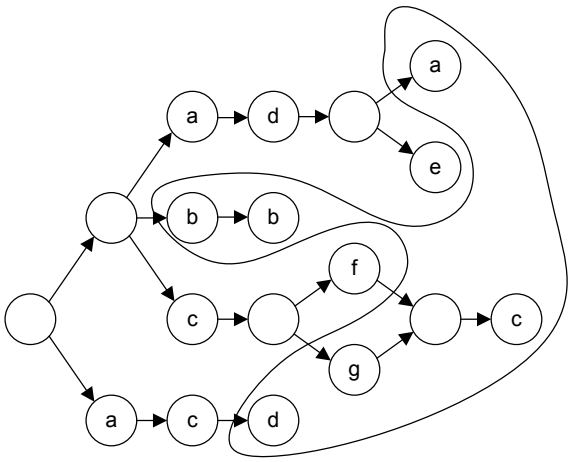


Figure 10. A partial Structural trace (nodes without identifiers are dummy fork/join nodes). Concatenation and quotient need to be defined such that the whole trace minus the shaded area can later be concatenated with the shaded area to re-form the entire trace.

concatenation joins two sequences, and quotient subtracts the given prefix from a sequence. VCR traces have previously been set within the UTP framework [11], and we should consider the same for Structural traces. However, the definition of these three relations and operations is difficult for Structural traces. Implicitly, traces must obey the following law:

$$\forall\, tr, tr' : (tr \leq tr') \Rightarrow (tr \mathbin{\hat{}} (tr' - tr) = tr')$$

Intuitively, the problem is that whereas a CSP trace has one end to append events to, a Structural trace typically has many different ends that events could be added to: with an open parallel composition, the new events could be added to any branch of the open composition, or after the parallel composition (thus closing it). We must find a suitable representation to show where the new events must be added. This is represented in graphical form in figure 10 – the problem is how to represent the shaded part of the tree so that it can be concatenated with the non-shaded part to form the whole trace.

We leave this for future work, but this shows that meshing Structural traces with existing trace theory is difficult or unsuitable. Hence, at this time we place our stress on the use of

Structural traces for practical investigation and understanding, rather than use in theoretical and formal reasoning.

4. Conclusions

We have examined three trace types: CSP, VCR and Structural. We have made a case for Structural traces being the most general form of the three and have shown that a given Structural trace can be converted to a set of VCR traces, and that a given VCR trace can be converted to a set of CSP traces, giving us an ordering for the different trace types. When developing concurrent systems, it might be instructive to examine the traces of a program under development, to verify that the current behaviour is as expected. This ongoing verification can be in addition to having used formal tools such as FDR to prove the program correct. Even after development is complete, it can be beneficial to check the traces of the system running for a long time under load, to check for patterns of behaviour such as starvation of processes. Additionally, it may be necessary to examine the program's behaviour following recovery from a hardware fault (e.g., replacement of a network router) or security intrusion to ensure the system continues to operate correctly – situtations that might be beyond the scope of a model checker.

Structural traces are the most general, the most efficient to record, and require the fewest assumptions for their recording strategy. Their efficiency arises from recording traces locally within each process, whereas recording CSP and VCR traces involves contending for access to a single centralised trace. They are also the most amenable to compression, as the repeating behaviour of an individual process is represented in the Structural trace. Regardless of which trace type is subsequently used for investigation, Structural traces can be used for the recording.

4.1. Future Work

Currently, two main areas of future work remain: visualisation and formalisation. We have explored visualising the different trace types, especially Structural traces, which provide the most useful information for visualising a trace. For large traces, Structural traces provide a means for implementing a trace browser that permits focusing on particular processes. Such a tool would provide the visualisation capabilities necessary to locate causes of problems recorded in the trace. Figure 8 provides examples of what the user interface of such a trace browser might look like, but this application has yet to be implemented.

While the Structural traces contain useful structure that permits visualisation and conversion to VCR and CSP traces, this same structure makes it less amenable to formal manipulation. We have explained how operations (concatenation and quotient) that are straightforward on CSP and VCR traces are much more difficult to define on Structural traces. Trying to define these operations has given us a greater appreciation of the elegance of CSP traces. Despite the challenge that defining concatenation and quotient presents, we remain motivated to solving this problem and ultimately drawing Structural traces into the Unifying Theories of Programming.

4.2. Availability

The CHP library [5] remains the primary implementation of the trace recording discussed here. We hope to soon release an update with the latest changes described in this paper and the accompanying programs for automatically generating the graphs.

Acknowledgements

We remain grateful to past reviewers for helping to clarify our thoughts on VCR and Structural traces, especially recasting the notion of parallel events as independent events. We are also grateful to Ian East, for his particularly inspiring "string-and-beads" diagram [12] that paved the way for much of the visualisation in 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] Neil C. C. Brown and Marc L. Smith. Representation and Implementation of CSP and VCR Traces. In *Communicating Process Architectures 2008*, pages 329–345, September 2008.
[4] 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.
[5] Neil C. C. Brown. Communicating Haskell Processes: Composable explicit concurrency using monads. In *Communicating Process Architectures 2008*, September 2008.
[6] W. Reisig. *Petri Nets—An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, New York, 1985.
[7] Peter H. Welch. Java Threads in the Light of occam/CSP. In *Architectures, Languages and Patterns for Parallel and Distributed Applications*, volume 52 of *Concurrent Systems Engineering Series*, pages 259–284. WoTUG, IOS Press, 1998.
[8] Neil C. C. Brown. How to make a process invisible. In *Communicating Process Architectures 2008*, page 445, 2008. Talk abstract.
[9] Jonathan Simpson and Christian L. Jacobsen. Visual process-oriented programming for robotics. In *Communicating Process Architectures 2008*, pages 365–380, September 2008.
[10] C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
[11] Marc L. Smith. A unifying theory of true concurrency based on CSP and lazy observation. In *Communicating Process Architectures 2005*, pages 177–188, September 2005.
[12] Ian East. *Parallel Processing with Communicating Process Architecture*. Routledge, 1995.