

# Representation and Implementation of CSP and VCR [and Structural] Traces

“Look Ma, no print statements!”

Neil Brown<sup>1</sup>    Marc Smith<sup>2</sup>

<sup>1</sup>Computing Laboratory, University of Kent, UK

<sup>2</sup>Department of Computer Science, Vassar College, NY, USA

8 September 2008



# Motivation: Debugging

When concurrent programs don't behave as intended

- Can't simply add print statements, which change behaviour
- But even worse for process-oriented programs: requires rewiring the entire network!

But how then can a programmer get an idea of what's happening?



# Recording a Program's Behaviour

A *trace* is a record of all the events a process has engaged in.

Consider the following CSP system:

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

Let's see what different types of traces look like for this system...



# Traces

Three types of traces:

**1** CSP traces

- A sequence of individual events, recorded by the observer; events observed simultaneously are interleaved

**2** VCR Traces

- A sequence of parallel event multisets; multiple observers account for different views

**3** Structural Traces

- Sequential and parallel composition of the trace reflects the program's structure



# Example of CSP Traces

Recall our example system:

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

The set of all possible maximal traces:

$$\begin{array}{lll} \langle a, b, c, d, e \rangle & \langle a, b, d, c, e \rangle & \langle a, b, d, e, c \rangle \\ \langle a, d, b, c, e \rangle & \langle a, d, b, e, c \rangle & \langle a, d, e, b, c \rangle \end{array}$$



# Example of VCR Traces

Recall our example system:

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

The set of all possible maximal traces:

$$\begin{aligned} &\langle a, b, c, d, e \rangle \quad \langle a, b, d, c, e \rangle \quad \langle a, b, d, e, c \rangle \\ &\langle a, d, b, c, e \rangle \quad \langle a, d, b, e, c \rangle \quad \langle a, d, e, b, c \rangle \\ &\quad \langle a, b, \{c, d\}, e \rangle \quad \langle a, b, d, \{c, e\} \rangle \\ &\quad \langle a, d, b, \{c, e\} \rangle \quad \langle a, d, \{b, e\}, c \rangle \\ &\quad \quad \langle a, \{b, d\}, \{c, e\} \rangle \end{aligned}$$



# Example of Structural Traces

Recall our example system:

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

The set of all possible maximal traces:

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



# Putting theory into practice

- Each of these trace styles could be useful to the programmer
- Once the program has been written, it is too late to implement the observer
- The observer should already be available, when desired, to record traces throughout a program's development



# Implementing The Observer

We have implemented the observer in the Communicating Haskell Processes library (more tomorrow), including:

- 1 CSP traces
- 2 VCR Traces
- 3 Structural Traces



# Implementing CSP Traces

Straightforward to implement:

- Append each event to global list.
- Potentially large amount of contention for write access.
- Recording and event are indivisible.
- *Requires no changes to the program.*



# VCR Traces

One *major* practical consideration: simultaneity

- True simultaneity is tricky
- We've changed the meaning of VCR's event multisets
  - The multisets no longer represent simultaneity
  - Instead they represent *event independence*

We deem  $a$  and  $b$  to be independent events if  $a$  did not observably require  $b$  to occur first, and vice versa.



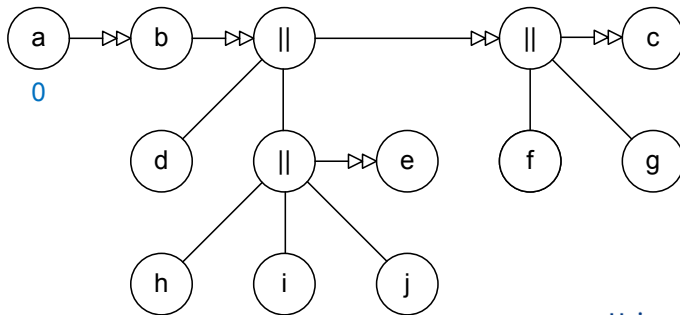
# Implementing VCR Traces

More difficult to implement:

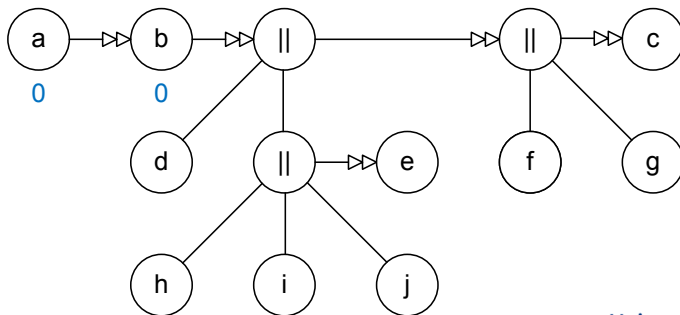
- Need to determine independence of events
- Maintain hierarchical process identifiers of the form:
 
$$2 \triangleright_1 4 \triangleright_0 1 \dots$$
  - Large numbers are sequence identifiers
  - Triangle subscripts are parallel identifiers
- Record the identifiers of all processes involved in an event
- Events  $a$  and  $b$  are independent iff all process identifiers recorded with  $a$  are independent from all identifiers recorded with  $b$



# Implementing VCR Traces

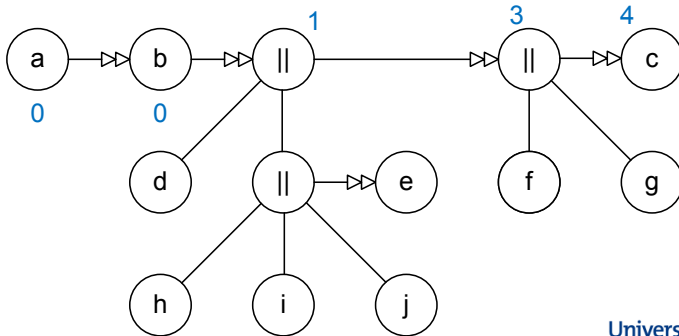
$$(a; b; (d \parallel ((h \parallel i \parallel j); e)); (f \parallel g); c$$


# Implementing VCR Traces

$$(a; b; (d \parallel ((h \parallel i \parallel j); e)); (f \parallel g); c$$


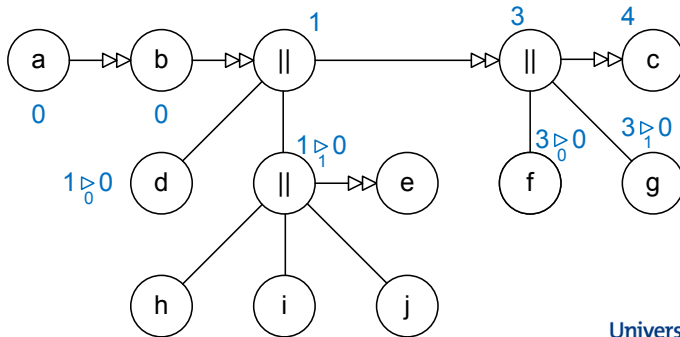
# Implementing VCR Traces

$(a; b; (d \parallel ((h \parallel i \parallel j); e)); (f \parallel g); c$



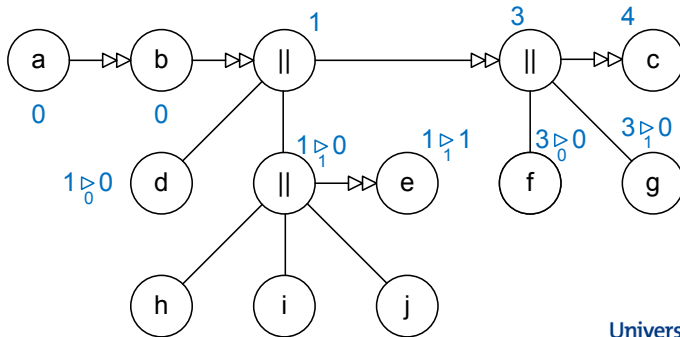
# Implementing VCR Traces

$(a; b; (d \parallel ((h \parallel i \parallel j); e)); (f \parallel g); c$

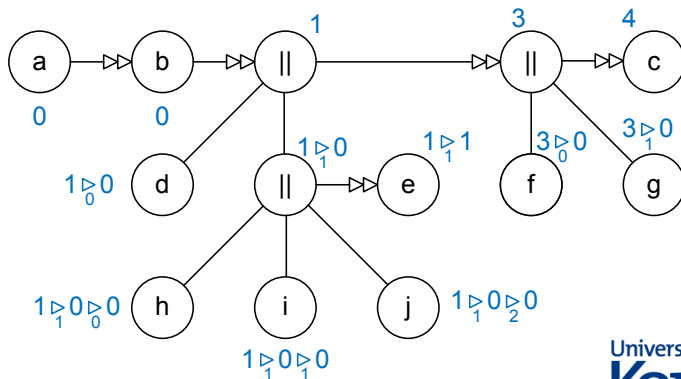


# Implementing VCR Traces

$(a; b; (d \parallel ((h \parallel i \parallel j); e)); (f \parallel g); c$

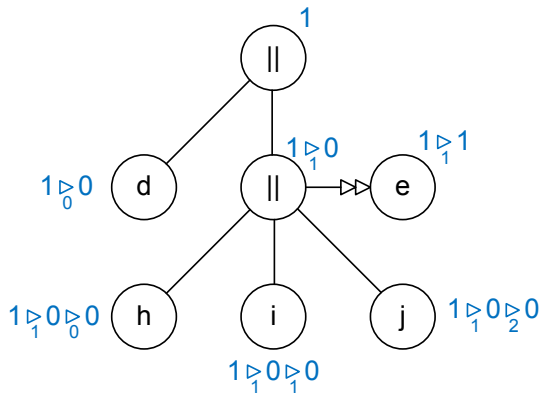


# Implementing VCR Traces

$$(a; b; (d \parallel ((h \parallel i \parallel j); e)); (f \parallel g); c$$


# Implementing VCR Traces

... $(d \parallel ((h \parallel i \parallel j); e)); \dots$



# Implementing VCR Traces

- Maintain hierarchical process identifiers of the form:

$$2 \triangleright_1 4 \triangleright_0 1 \dots$$

- Large numbers are sequence identifiers
  - Triangle subscripts are parallel identifiers
- To determine independence of processes:
    - Compare identifiers left-to-right
    - If a process identifier differs in sequence identifier, can deduce time-ordering
    - If a process identifier differs in parallel identifier, know independent



# Implementing Structural Traces

Surprisingly straightforward due to correspondence between program and trace:

- Shape of the program determines the shape of the trace.
- Append each event to process-local list.
- After parallel composition, store trace of sub-processes in parent.



# Traces could be large

Traces could reasonably reach gigabytes in an hour

- Could just keep a rolling trace (e.g., most recent thousand events)
- Or we could compress them during program execution (i.e., recursive processes have repeating behaviour  $\Rightarrow$  very natural to compress)



# Conclusions

Made theoretical models of tracing available to programmers who are not familiar with formal methods

- Tracing allows you to observe your program's behaviour without changing any of the code (e.g., adding print statements)
- Choice of trace representations (CSP/VCR/Structural)
- Implemented in CHP (see tomorrow), but easy to replicate in any other framework
- We still need ways to visualise/manipulate traces

