The Edinburgh Concurrency Workbench user manual (Version 7.1)

Original by Faron Moller, updated since CWB v7.0 by Perdita Stevens
Laboratory for Foundations of Computer Science
University of Edinburgh

$Date : 1999/07/1816 : 55 : 43$

Contents

1 Introduction **3**

2 Using the Concurrency Workbench **3**

3 Using the CWB from emacs **4**

4 Basic format and syntax of input **4**
4.1 The very basics of CWB format 4
4.2 Variables and identifiers 5

5 The languages understood by the CWB **5**
5.1 (T)CCS 5
5.2 SCCS 10
5.3 The modal mu calculus 12

6 Environments **14**

7 Another Example Session **15**

8 The Commands of the CWB **16**
8.1 Using on-line help 16
8.2 Quitting 16
8.3 Defining things and maintaining the environments 16
8.4 Working with files 18
8.5 Working with a single agent 18
8.6 Comparing two agents 22
8.7 Working with logic 26
8.8 Simulating the behaviour of an agent. 27

9 References **28**

1 Introduction

The Edinburgh Concurrency Workbench (CWB) is an automated tool which caters for the manipulation and analysis of concurrent systems. In particular, the CWB allows for various equivalence, preorder and model checking using a variety of different process semantics.

With the Workbench it is possible to:

- define *behaviours* given either in the syntax of the Temporal CCS (TCCS) or SCCS, and perform various analyses on these behaviours, such as analysing the state space of a given process, or checking various semantic equivalences and preorders;
- define *propositions* in a powerful modal logic and to check whether a given process satisfies a specification formulated in this logic;
- derive automatically logical formulae which distinguish nonequivalent processes;
- interactively simulate the behaviour of an agent, thus guiding it through its state space in a controlled fashion.

This document is intended to serve as an introduction only to the CWB. Some amount of description is contained within this document on the semantic interpretation of terms representing concurrent systems. However, the document is intended for readers already familiar with Process Algebra in some form, such as the Calculus of Communicating Systems (CCS), as well as the operational semantic definition of the terms in such an algebra. Those unfamiliar with these concepts are referred to the list of references which follows the main body of the text.

Section 8 provides a reference guide to the commands of the CWB.

2 Using the Concurrency Workbench

The CWB is an interactive system; when invoked, the user may issue commands which typically will bind identifiers, analyse derivations of processes, or check for equivalences between processes. The default input language for the Workbench is TCCS, but if you invoke it with the command line argument "sccs", then the input language will be SCCS.

In Figure 1 we have a sample session which demonstrates the implementation in CCS of a three-place buffer by linking up three single-element cells. The only commands used in this sample session are `agent` (bind an identifier to an agent), `eq` (observational equivalence, also known as weak bisimilarity), and `quit`. Lines 1 through 8 simply bind agent identifiers to process expressions. Line 9 tests the two agents identified by Buff3 and Spec for observational equivalence. The system responds with the result (line 9_a), in this case `true` signifying that the two agents in question are indeed observationally equivalent. Finally, in line 10, the user requests that the session terminate.

Hitting $\langle CTRL \rangle$-*c* at any point immediately returns you to the top-level command prompt. This is useful for aborting commands. Also, any line whose first printable character is an asterisk `*` will be interpreted as a comment line and will be ignored, as will any blank line when the CWB is not expecting the user to input a (possibly empty) list. This is useful for commenting and formatting code which can be read into the CWB using the `input` command as described later.

```
Edinburgh Concurrency Workbench, version 7.0alpha3,
Tue Sep 20 20:42:57 BST 1994
 (1)   Command: agent  Cell  = a.'b.Cell;
 (2)   Command: agent  C0  = Cell[c/b];
 (3)   Command: agent  C1  = Cell[c/a,d/b];
 (4)   Command: agent  C2  = Cell[d/a];
 (5)   Command: agent  Buff3  = (C0 | C1 | C2)\{c,d};
 (6)   Command: agent  Spec  = a.Spec';
 (7)   Command: agent  Spec'  = 'b.Spec + a.Spec'';
 (8)   Command: agent  Spec''  = 'b.Spec' + a.'b.Spec'';
 (9)   Command: eq (Buff3, Spec);
 (9a)  true
 (10)  Command: quit;
```

Figure 1: Sample session 1

3 Using the CWB from emacs

If you have an Emacs 19 (FSF or XEmacs) you can run the CWB from inside emacs, and this is the recommended way of running the CWB. See the installation for instructions on how to install the CWB emacs mode. Once you are set up, you can run the CWB using M-x cwb. You get a set of facilities which will be familiar to users of shell-mode and other comint-based modes. For example,

- M-p and M-n allow you to cycle through previous commands, so that you can repeat or edit them without retyping.
- There is completion on command names using TAB.
- If you have installed the graph viewer daVinci, C-c C-x starts up a graph viewer for use with the CWB. You can then interact with the viewer using the emacs menu. Look at the top of your emacs window towards the right hand end of the menu. Choose CWB, then Graph display, to get a list of possibilities. (There are also a few options on the daVinci Edit menu.)

  (Warning: think about how large the statespace is before you try displaying it!)
- Most importantly, C-c C-b sets you up to send a bug report or comment to the maintainer of the CWB.

For more information, see the online help, accessed by C-h m. (Or later versions of this manual!)

4 Basic format and syntax of input

4.1 The very basics of CWB format

All commands end with a semi-colon ; followed by a newline.

Commands, agent definitions etc can spread over several lines; the Version 6 continuation character is no longer required (or even permitted).

If you get asked a question expecting the answer y or n, the semi-colon is optional.

Anything after a semi-colon will be ignored. Therefore you cannot put more than one command on a line, even if you separate them by semi-colons.

* is the comment character. Anything between a * and the next newline will be ignored. This is now true even if * is not the first character on the line.

Note that * and ; cannot be used for anything other than starting a comment or finishing a command respectively. In particular, they cannot be used in strings, including filenames and arguments to echo.

4.2 Variables and identifiers

Names of actions, and formal parameters standing for actions, begin with a lower case letter a - z.

Identifiers for sets, relabelling functions, agents or propositions, and formal parameters standing for these things, begin with an upper case letter A - Z.

The second and subsequent characters of an identifier may be:

- lower case letters a - z
- upper case letters A - Z
- the digits 0 - 9
- the characters ? ! _ ‘ ’ -
- the characters **# ^ except** in SCCS action names, where these characters have a special meaning
- anything you like between a pair of % signs, at the very end of an identifier (this feature was added for use by Glenn Bruns' value-passing front end).

Also for obvious reasons:

- eps and tau may not be used as action names or formal action parameters
- T and F may not be used as proposition identifiers or formal parameters to propositions.

Please note that it is possible that the set of permitted characters may be restricted further in future; if you want to write future-proof scripts you would be well advised to stick to alphanumeric characters.

5 The languages understood by the CWB

5.1 (T)CCS

ACT	**a**	**tau**	**internal action**
		n	**name: an identifier beginning with a lower case letter**
		'n	**coname. This is an ASCIIfication of normal CCS's overbarred n.**
		eps	empty observation, not allowed in agents (because it isn't really an action!)

AGENT	**A**	**0**	**deadlock (nil)**
		a.A	**action prefix**
		A + A	**(weak) choice**
		A | A	**parallel composition**
		A \ a	**restriction of a single action**
		A \ S	**restriction of a set**
		A[R]	**relabelling**
		(A)	**you can use brackets just as you'd expect**
		X	**agent variables begin with upper case**
		@	divergence (for Walker-style divergence preorders etc.)
		$ 0	delayed nil (for TCCS)
		$ a.A	delayed action prefix (for TCCS)
		A ++ A	strong choice (for TCCS)
		A1 : S1 | | A2 : S2	synchronisation merge (CSP style) (S1 and S2 are action sets)
		X(P,...)	parameterised agent

The order of precedence is as usual: Summation binds the weakest, followed by Strong Summation, Parallel Composition, Prefixing, and finally Restriction and Relabelling. These precedence rules can be overcome by using parentheses.

Parameterised agents

Agents can be parameterised on other agents, on actions or on sets of actions. This does not simulate value-passing CCS, but it does provide some more flexibility.

Some examples of such definitions:

```
Command: agent A(B,x) = B + x.0;
Command: agent C = A(a.0, error);
Command: eq(C, A(a.0, y)); * false, of course: illustrating usage!
false
Command: agent D(S) = a.0\S;
Command: print;

** Agents **
agent A(B : agent,x : act) = B + x.0;
agent C = A(a.0,error);
agent D(S : set) = a.0\S;
Command:
```

Types of agent parameters

A formal parameter used in the definition of a parameterised agent has a certain (monomorphic!) type.

```
agentParamType := act | set | agent
```

As before:

- Names of formal parameters of type act begin with a lower case letter.
- Names of formal parameters of type set or agent begin with an upper case letter.

The CWB will infer types where it's possible to do so from the definition; if it can't, it will ask for clarification, or signal an error if the CWB session is not interactive.

To specify a type:

```
agent A(X : set) = a.0\X;
```

Specifications are legal only on the LHS of such a definition.

Details

- You can only leave the specification out if it can be inferred from this definition in isolation. For example in this case the specification is essential:

  ```
  agent A(X) = a.X;
  agent B(X : agent) = A(X);
  ```

  even though the type of X could be inferred by looking at the type of A. The justification for this is that the binding of A might change.

- More generally, notice that for the same reason (late binding) the following is perfectly legal:

  ```
  agent A(X : set) = ...
  agent B(Y : agent) = ... A(Y)...
  ```

  since each definition looks fine in isolation. Of course if you attempt to use B without first changing the binding of A, an error is likely to result. Slightly less excusable is that the following unusable definition will be accepted (and we will infer X : agent)

  ```
  agent A(X) = a.X + A({a});
  ```

  but I think if you write this kind of thing you deserve what you get.

- *Parameterised* agents used in the body of a definition are not bound to formal parameters. For example, in this case the X on the RHS is not the same X as on the LHS; this makes sense only if there is a parameterised X in the environment.

  ```
  agent A(X) = X(a.0);
  ```

- Note for TCCS users: a formal parameter standing for a time has type act!

Sets can be used in agent definitions (restriction or synchronous merge), and in propositions (modalities). The situation becomes slightly complicated, because agent definitions notionally mention sets of *names*, whereas modalities genuinely contain sets of *actions*.

If you give an action set in an agent definition explicitly, by listing its members, the CWB will prevent you from listing actions other than names and conames. However, if you use a set identifier, it cannot do this. (Sets, like other CWB entities, are bound dynamically, so at any stage you might change the contents of the set mentioned in the agent definition.) Therefore if you do use actions other than names and conames in an action set used to restrict an agent, **they will simply be ignored**.

It wouldn't make (much) sense for a set used to restrict an agent to contain the special actions eps or tau; really it's better to think of restricting a set of *names*. Therefore in the CWB you may not define a set identifier to contain eps or tau. If you want to use tau in a modality, you have to write the modality using an explicit list of actions, rather than using a set identifier.

SET	S	{ a,... } V	action set (not including eps or tau) set variables begin with upper case

RELABEL	**R**	**[a/a,...]** **V**	**relabelling function, given as a set of pairs** **relabelling variables begin with upper case**

Interpreting TCCS Agents We give here only a brief explanation of the transitional semantics of TCCS, and urge the reader to become more familiar with the theory of processes by consulting the references in order to seriously apply the CWB.

Terms of the process algebra are defined with respect to their behaviour, as stipulated by the transitions which they can perform. For the basic calculus, these transitions are described as follows.

- **0** can do no transitions, and represents the deadlocked process.
- $\perp$ can do no transitions, and represents the divergent or undefined process.
- $a.P$ can do the action a and evolve into P:

  $a.P \xrightarrow{a} P.$
- A sum can behave as any of its summands:

  $A_1 + \cdots + A_n \xrightarrow{a} A'$ whenever $A_i \xrightarrow{a} A'$ for some i.
- A parallel composition can perform the actions of any of its components, or perform a synchronisation between any two of them:

  $A_1|\cdots|A_i|\cdots|A_n \xrightarrow{a} A_1|\cdots|A_i'|\cdots|A_n$ whenever $A_i \xrightarrow{a} A_i'$ for some i; and

  $A_1|\cdots|A_i|\cdots|A_j|\cdots|A_n \xrightarrow{\tau} A_1|\cdots|A_i'|\cdots|A_j'|\cdots|A_n$ whenever $A_i \xrightarrow{a} A_i'$ and $A_j \xrightarrow{\overline{a}} A_j'$ for some i and j.
- A synchronisation merge synchronises on common actions:

$$A_1 : \{a_{11}, \ldots, a_{1k_1}\} \;\|\; \cdots \;\|\; A_n : \{a_{n1}, \ldots, a_{nk_n}\}$$
$$\xrightarrow{a} \; A'_1 : \{a_{11}, \ldots, a_{1k_1}\} \;\|\; \cdots \;\|\; A'_n : \{a_{n1}, \ldots, a_{nk_n}\}$$

whenever $a = a_{ij}$ for some i and j, and $A'_i \equiv A_i$ whenever $a \neq a_{ij}$ for all j, and $A_i \xrightarrow{a} A'_i$ whenever $a = a_{ij}$ for some j.

- $A\backslash L$ can do the transitions of A which are not prohibited by the set L:

  $A\backslash L \xrightarrow{a} A'\backslash L$ whenever $a, \overline{a} \notin L$ and $A \xrightarrow{a} A'$.

- $A[f]$ can do the transitions of A suitably relabelled:

  $A[f] \xrightarrow{f(a)} A'[f]$ whenever $A \xrightarrow{a} A'$.

  τ cannot be relabelled, but $f(a) = \tau$ is allowed, resulting in a and $\overline{a}$ being hidden.

Timed features

Here we describe the idling transitions for the original calculus TCCS.

Health warning You may reasonably find the relationship between what is said here and the circumstances under which the CWB reports a transition with label 1 obscure. This is not core CWB functionality, and it is not likely to be developed further: but it is left in to accommodate current users. The CWB does have the property that if your agent definitions do not use the timing constructs, the CWB will never give a response that does.

Since only discrete times are allowed in this version of the CWB, we only need consider the unlabelled transition, representing a unit idling step.

- $\underline{\mathbf{0}}$ can idle indefinitely:

  $\underline{\mathbf{0}} \rightsquigarrow \underline{\mathbf{0}}$.

- $\underline{a}.P$ can idle or do the action a and evolve into P:

  $\underline{a}.P \rightsquigarrow \underline{a}.P$ and $a.P \xrightarrow{a} P$.

- $t.P$ (for $t > 0$) idles for t steps before evolving into P:

  $1.P \rightsquigarrow P$; and
  $(t+1).P \rightsquigarrow t.P$.

- A (weak) sum can idle as long as at least one of its summands can idle, and will evolve according to how its summands evolve:

  $\sum A_i \rightsquigarrow \sum\{A'_i \;:\; A_i \rightsquigarrow A'_i\}$, as long as this set is nonempty. (And yes, the order in which the summands are printed by the CWB is potentially arbitrary. This is probably a bug: but if so it's an extremely old and documented bug...)

- A strong sum can idle as long as each of its summands can idle:

  $A_1 \texttt{++} \cdots \texttt{++} A_n \rightsquigarrow A'_1 \texttt{++} \cdots \texttt{++} A'_n$ whenever $A_i \rightsquigarrow A'_i$ for each i.

- A parallel composition can idle as long as each of its summands can idle:

$$A_1|\cdots|A_n \leadsto A_1'|\cdots|A_n' \text{ whenever } A_i \leadsto A_i' \text{ for each } i.$$

Typically one is interested not in these transition relations, but in ones which abstract away from the internal communication action τ, giving the following relations.

$$\begin{aligned} &\stackrel{\epsilon}{\Longrightarrow} = (\stackrel{\tau}{\longrightarrow})^* \qquad \text{(also written } \Longrightarrow) \\ &\stackrel{a}{\Longrightarrow} = \Longrightarrow\stackrel{a}{\longrightarrow}\Longrightarrow \\ &\approx> = \Longrightarrow\leadsto\Longrightarrow \end{aligned}$$

An important equivalence relation is that of observational equivalence. This is the most basic equivalence defined over CCS agents, and is the one we used in the sample session of Figure 1. Informally, observational equivalence deems that two terms are equivalent if whenever one can perform an observable transition (including the empty observation), then the second can perform that same observation and evolve into an equivalent state.

There are many other equivalence and preorder relations defined over agents which the CWB is capable of deciding. We leave any explanation of these to the references given.

5.2 SCCS

The workbench can use process terms styled after the calculus SCCS, *Synchronous Calculus of Communicating Processes*. These again are built up from *actions* and *identifiers* using a collection of *process constructors*. The Workbench uses a similar syntactic convention for identifiers as with TCCS, but now actions are built up from more primitive *particles*.

PARTICLE	**p**	**n**	**name: an identifier beginning with a lower case letter**
		'n	**coname: the inverse of n, usually written with a superscript -1.**

ACT	**a**	**1**	**internal action (tick)**
		p	**particle**
		p#q	**product of particles**
		p^n	**exponentiation: particle p, positive integer n times**
		eps	empty observation, not allowed in agents (because it isn't really an action!)

Positive integers other than 1 on their own do not count as actions, for example in logical modalities and action sets, even though they can be used as agent prefixes: think of the latter as syntactic sugar.

Action sets are just as in CCS.

AGENT	**A**	**0**	**deadlock (nil)**
		a:A	**action prefix**
		n:A	**n a positive integer: A prefixed with n ticks**
		A + A	**choice**
		A | A	**parallel composition**
		A \ p	**permission for a single particle**
		A \ S	**permission for a particle set**
		A[R]	**relabelling**
		(A)	**you can use brackets just as you'd expect**
		X	**agent variables begin with upper case**
		@	divergence (for Walker-style divergence preorders etc.)
		$ A	A with arbitrary delay

The order of precedence is as usual: Summation binds the weakest, followed by Strong Summation, Parallel Composition, Prefixing, and finally Permission and Relabelling. These precedence rules can be overcome by using parentheses.

If set used as a permission contains a non-particle action, this will cause a run-time error.

RELABEL	**R**	**[a/p,...]**	**relabelling function, given as a set of pairs.**
		V	**relabelling variables begin with upper case**

Relabel a particle to an action: this extends to a relabelling from actions to actions.

Interpreting SCCS Agents Again we give here only a brief explanation of the transitional semantics of SCCS and urge the reader to become more familiar with the theory of processes by consulting the references in order to seriously apply the CWB.

Terms of the process algebra are defined with respect to their behaviour, as stipulated by the transitions which they can perform. These transitions are described as follows. Notice especially that SCCS has a quite different style of communication from the two-agent handshaking of CCS.

- **0** can do no transitions, and represents the deadlocked process.
- $\perp$ can do no transitions, and represents the divergent or undefined process.
- $a : P$ can do the action a and evolve into P:

  $a : P \xrightarrow{a} P.$
- A sum can behave as any of its summands:

  $A_1 + \cdots + A_n \xrightarrow{a} A'$ whenever $A_i \xrightarrow{a} A'$ for some i.
- A parallel composition performs the action which is the product of the actions of the components of the composition:

  $A_1 | \cdots | A_n \xrightarrow{a_1 \# \cdots \# a_n} A'_1 | \cdots | A'_n$ whenever $A_i \xrightarrow{a_i} A'_i$ for all i.
- $A \backslash L$ can do the transitions of A which are in the subgroup generated by L:

  $A \backslash L \xrightarrow{p_1 \# \cdots \# p_n} A' \backslash L$ whenever $p_1, \cdots, p_n$ (or their inverses) are in L and $A \xrightarrow{p_1 \# \cdots \# p_n} A'$.

- $A[f]$ can do the transitions of A suitably relabelled:

  $A[f] \xrightarrow{f(a)} A'[f]$ whenever $A \xrightarrow{a} A'$.

Again, one is typically interested not in this transition relation, but in that which abstracts away from the internal communication action 1, giving the following relation:

$$\stackrel{\epsilon}{\Longrightarrow} \; = \; (\xrightarrow{1})^* \qquad \text{(also written } \Longrightarrow\text{)}$$
$$\stackrel{a}{\Longrightarrow} \; = \; \Longrightarrow\xrightarrow{a}\Longrightarrow$$

This transition relation again generates observational equivalence.

5.3 The modal mu calculus

The CWB has a model checker for determining the satisfiability with respect to a given agent of propositions expressed in the propositional modal μ-calculus. The command which invokes the model checker is `checkprop` (*check proposition*), and requires two parameters: an agent to analyse, and a proposition of which to check the satisfiability.

PROP	**P**	**T**	**true**
		F	**false**
		Q	**agent variables (which may not be T or F) begin with upper case**
		~P	negation
		P & P	**conjunction**
		P | P	**disjunction**
		(P)	**you can use brackets just as you'd expect**
		P =⟩ P	implication
		[a,...]P	**strong necessity**
		[-a,...]P	**strong complement necessity**
		[[a,...]]P	weak necessity
		[[-a,...]]P	weak complement necessity
		⟨a,...⟩P	**strong possibility**
		⟨-a,...⟩P	**strong complement possibility**
		⟨⟨a,...⟩⟩P	weak possibility
		⟨⟨-a,...⟩⟩P	weak complement possibility
		[S]P	strong necessity using a set identifier S
		[-S]P	strong complement necessity using a set identifier S
		[[S]]P	weak necessity using a set identifier S
		[[-S]]P	weak complement necessity using a set identifier S
		⟨S⟩P	strong possibility using a set identifier S
		⟨-S⟩P	strong complement possibility using a set identifier S
		⟨⟨S⟩⟩P	weak possibility using a set identifier S
		⟨⟨-S⟩⟩P	weak complement possibility using a set identifier S
		min(X.P)	**least fixpoint temporal formula**
		max(X.P)	**greatest fixpoint temporal formula**
		V(Q,...)	parameterised agent

In the absence of parentheses, the built-in unary operators bind more tightly than binary ones, with `=>` having a lower precedence than `&` and `|` (which have equal precedence).

Interpreting logical formulas The meaning of propositions is defined with respect to agents. Intuitively, an agent is said to *satisfy* a proposition if the proposition is true of the agent. Thus, every agent satisfies `T`, and no agent satisfies `F`. The meanings of the propositional connectives and macro terms are as one would expect; we will not go into them but instead concentrate on the modal and fixed point operators.

- An agent A satisfies $\texttt{[}K\texttt{]}P$ if every K-derivative of A satisfies P; that is, if A' satisfies P whenever there is an $a \in K$ such that $A \xrightarrow{a} A'$.
- An agent A satisfies $\texttt{<}K\texttt{>}P$ if it has a K-derivative satisfying P, that is, if there is some $a \in K$ such that $A \xrightarrow{a} A'$ with A' satisfying P.
- An agent A satisfies $\texttt{[[}K\texttt{]]}P$ if every K-observation derivative of A satisfies P; that is, if A' satisfies P whenever there is an $a \in K$ such that $A \overset{a}{\Longrightarrow} A'$.
- An agent A satisfies $\texttt{<<}K\texttt{>>}P$ if it has a K-observation derivative satisfying P, that is, if there is some $a \in K$ such that $A \overset{a}{\Longrightarrow} A'$ with A' satisfying P.

Empty sets of actions are allowed as modalities, as are their complements, so that for instance `[-]F` represents a strongly deadlocked property: upon performing *any* action (in the complement of the empty set), you arrive at a state satisfying `F`, or in other words, it is not possible to do any action whatsoever. In such cases, the complement of the empty set, written simply as "`-`", should be read as a *wild card* action.

The maximal fixed point operator $\texttt{max(X.}P\texttt{)}$, can be interpreted as the infinite conjunction

$$P^0 \;\texttt{\&}\; P^1 \;\texttt{\&}\; P^2 \;\texttt{\&}\; \cdots, \qquad \begin{array}{rl} \text{where} & P^0 \;=\; \texttt{T}, \\ \text{and} & P^{i+1} = P\left[{}^{P^i}\!/\texttt{X}\right]. \end{array}$$

This constructor is useful for expressing invariance properties such as the branching time temporal logic operator $\Box$, which can be defined by

$$\Box P \;=\; \texttt{max(X.}P \;\texttt{\& [-]X)}$$

This property states of an agent that the property P holds of the agent, and recursively continues to hold upon doing any derivation. We can also use maximal fixpoints to express weak eventuality properties, such as the *weak until* operator

$$\text{WeakUntil}\; P\, Q \;=\; \texttt{max(X.}Q \;\texttt{|}\; \texttt{(}P \;\texttt{\& [-]X)).}$$

This property states of an agent that the property P continues to holds of the agent throughout its derivation until such time as the property Q holds. This is a *weak* eventuality property in the sense that Q need never hold at any time.

The minimal fixed point operator `min` is the *dual* of `max`. That is, $\texttt{min(X.}P\texttt{)}$ can be interpreted as the infinite disjunction

$$P_0 \;\texttt{|}\; P_1 \;\texttt{|}\; P_2 \;\texttt{|}\; \cdots, \qquad \begin{array}{rl} \text{where} & P_0 \;=\; \texttt{F}, \\ \text{and} & P_{i+1} = P\left[{}^{P_i}\!/\texttt{X}\right]. \end{array}$$

It can be defined in terms of the maximal fixpoint operator as follows.

$$\texttt{min(X.}P\texttt{)} = \texttt{-max(X.-}P\left[{}^{-\texttt{X}}\!/\texttt{X}\right]\texttt{)}$$

This constructor is useful for expressing strong eventuality properties such as the *strong until* operator

$$\text{StrongUntil}\, P\, Q \;=\; \texttt{min(X.}Q\texttt{ | (}P\texttt{ \& <->T \& [-]X)).}$$

This property states of an agent that the property P continues to holds of the agent throughout its derivation until such time as the property Q holds, and furthermore Q is guaranteed to hold at some point. This final clause makes this a *strong* eventuality property.

6 Environments

An *environment* is a (partial) mapping from identifiers to objects, which allows for the objects to be globally referred to by an identifier to which the object is bound. There are several separate environments built into the CWB, for different objects, namely *agents*, *action sets*, *particle sets* (for SCCS) and *propositions*. We begin by discussing the environment which we have already seen, that for *agents*.

We saw that we can bind identifiers to agents within an environment by using the command `agent`. For example, the effect of the command

```
agent   Cell = a.'b.Cell;
```

as used in the example session of Figure 1 is to associate with the identifier `Cell` the agent which can perform the actions `a` and `'b` in a cyclic fashion. The association allows us to define the agent recursively, and allows us to define other agents in terms of this agent, simply by referring to the identifier `Cell`. Thus we were able in our example session to define the three individual cell agents `C0`, `C1` and `C2`, and finally the three-place buffer agent `Buff3`.

Note that the bindings are "dynamic" rather than "static", so that changing the binding of some identifier may change completely the behaviour of another agent defined in terms of it. For example, suppose that `A` is bound to `a.B`, and `B` is bound to `b.0` in the agent environment. If `B` is rebound to `c.0`, then the `B` in the definition of `A` is the "new" `B`. Thus after the rebinding, `A` can perform the transition sequence $\xrightarrow{\texttt{a}}\xrightarrow{\texttt{c}}$. The reason for dynamic binding is that it affords one the ability to modify interactively agents and propositions.

The next type of environment of interest is that of *action sets*. Notice that according to the syntax of CCS agents given above, we could write a restriction as $A\backslash L$, where L is assumed to be bound to some set of actions. We can bind sets to identifiers by using the command `set` (*bind set identifier*). For example, in the sample session of Figure 1, we could have replaced the command on line 5 by the following commands.

```
set  L = {c, d};
agent Buff3 = (C0 | C1 | C2)\L;
```

When listing the actions to include in the action set, we are restricted from using the actions `tau` (for CCS) or `1` (for SCCS), or `eps`, which represent the CCS event τ or the SCCS event *tick*, and ϵ which cannot be prevented from occurring through the use of the action restriction operator. The actions to be included in the set must be listed wholly on one line, separated by spaces or commas. We will discuss the format of commands and their parameters more fully in Section 8.

There is one further environment in the CWB, pertaining to logical *propositions*. We postpone discussion of this environment until after discussing the logical aspect of the CWB, apart from noting that the binding of propositional identifiers is accomplished using the command `prop`.

```
Edinburgh Concurrency Workbench, version 7.0alpha3,
Tue Sep 20 20:42:57 BST 1994
          * Assume we have the agents from Session 1
 (1)      Command:   sort  (Buff3);
 (1a)     {a,'b}
 (2)      Command:   size  (Buff3);
 (2a)     Buff3 has 12 states.
 (3)      Command:   min  (Buff3Min, Buff3);
 (3b)     Buff3Min has 4 states.
 (4)      Command:   vs (3, Buff3Min);
 (4a)     === a a a ==>
          === a a 'b ==>
          === a 'b a ==>
 (5)      Command:   random (16, Buff3Min);
 (5a)     a,'b,a,a,'b,'b,a,a,a,'b,'b,a,'b,a,a,'b
 (6)      Command:   quit;
```

Figure 2: Sample session 2

7 Another Example Session

In this section, we extend the sample session of Figure 1 to demonstrate a few useful commands. The session appears in Figure 2.

We first compute the syntactic *sort* of the agent `Buff3` (line 1), and discover that it has sort $\{a, \overline{b}\}$ (line 1_a). Computing the sort can be a useful check to see that one has typed in an agent correctly.

Next we compute the size (of the state space) of the agent (line 2), and discover that it has 11 states (line 2_a). This can give us an impression of how big our analysis problem is, and how feasible the analysis of the agent is. Notice, however, that because there are several different notions of what it means for two states to be the same, and since there are several different CWB commands that tell you, among other things, how many states an agent has, it's possible that you may get slightly different answers depending on exactly how you ask the question! Do not be disturbed by this; just remember that the number of states is only supposed to be a rough guide.

Many of these states are weakly bisimilar, and so a useful procedure to apply before performing many analyses on an agent is to minimise (the state space of) the agent (line 3). After minimisation, you are informed of the resulting size of the minimised agent (line 3_b); in this case, the agent Buff3 is minimised down to 4 states.

A first analysis of an agent might be to witness its behaviour. One possibility is to look at all possible sequences of visible actions of a given length which the agent can perform, using the `vs` (*visible sequence* command (lines 4 and 4_a). Another possibility, which is suitable for analysis of *depth* of behaviour rather than *breadth* of behaviour, is to ask for a random sequence of visible actions of a given length which the agent can perform using the command `random` (lines 5 and 5_a).

There are many other analysis routines which can be used. These are described in the rest of this document, and summarised in Section 8.

8 The Commands of the CWB

In this section we present the full list of commands available in the system, describing their parameters and results.

8.1 Using on-line help

From basic module.

help : provide on-line help

Usage: `help;` brief help on what help is available

Usage: `help commands;` list all commands, with one-line descriptions

Usage: `help ⟨command⟩;` give help on the particular command

Syntax of the process algebra, which used to be under help syntax, is now given by a command with the name of the process algebra, ccs or sccs. Similarly, to get help on the syntax of the logic, use command logic. The old help brief has been removed, since it didn't seem very useful. (Let me know what would be useful!)

8.2 Quitting

From basic module.

quit : terminates the workbench session

Usage: `quit;`

Synonyms: exit, bye

8.3 Defining things and maintaining the environments

From basic module

agent : change (or show) the definition of an agent identifier

Usage: `agent X = A;` binds agent A to identifier X

Usage: `agent X(FP) = A;` binds a parameterised agent ((T)CCS only)

Usage: `agent X;` prints the definition of X

An agent identifier must begin with an upper case letter.

(T)CCS only:

If P is a formal parameter, it is an error to use P(anything) in the body of the definition.

If a formal parameter name starts with a lower case letter, it stands for an action. If it starts with an upper case letter, it stands for an agent or for an action set, depending on context.

Synonyms: bi,pi

set : change (or show) the definition of a set identifier

Usage: `set S = a,b,...;` binds a set of actions

Usage: `set S;` prints set bound to S

A set identifier must begin with an upper case letter. A named set cannot (currently) contain eps or tau.

Synonyms: basi,pasi

relabel : change (or show) the definition of a relabelling identifier

Usage: `relabel R = [a/p,...];` binds a relabelling function

Usage: `relabel R;` prints function bound to R

a/p means that p is relabelled to a. You cannot relabel tau. If p is relabelled to a, then 'p is also relabelled to 'a. You cannot define a relabelling function which violates this rule.

print : show the definitions of all identifiers

Usage: `print;`

To print just one binding, say that of agent A, use 'agent A;'. Etc.

Synonyms: pe,pae,pase,ppe

See also: agent, set, relabel, prop, save

clear : removes all bindings (a fresh start)

Usage: `clear;`

From optional module Logic

prop : change (or show) the definition of a proposition identifier

Usage: `prop P = F;` binds proposition F to identifier P

Usage: `prop P(FP) = Q;` binds a parameterised proposition

Usage: `prop P;` prints the definition of P

A proposition identifier must begin with an upper case letter.

If a formal parameter starts with an upper case letter, it stands for a proposition. For obvious reasons T and F are not valid formal parameter names!

If a formal parameter name starts with a lower case letter, it stands for a modality. The corresponding actual parameter can be positive or negative and can be an action set identifier, or an explicit list, e.g. Set, -Set, a,b, -a,b.

If the actual parameter is a modality (positive or negative) containing a single action, you may omit the , writing for example

prop Can(a) = $\langle$a$\rangle$T;

but **WARNING**: it is still a modality parameter, not an action parameter. So if you write

prop P(a) = $\langle$a,b$\rangle$Q,

the action a on the right hand side has NOTHING TO DO with the modality a on the left hand side. You have been warned!

Synonyms: bpi,ppi

See also: logic

8.4 Working with files

All from basic module.

input : execute the CWB commands in the given file

Usage: `input "file";`

The file may contain any CWB commands (provided they do not require user interaction: e.g. you cannot use sim in an input file!) In particular, the file can itself contain input commands. If an error is found at any level, the CWB will stop processing input files and return to standard terminal input mode.

There must be quotes around the file name. The file name can be absolute or relative, but must not contain metacharacters like ..

output : control where CWB output is written

Usage: `output "file";` sends output to the given file in future

Usage: `output;` sends output to stdout (e.g. the terminal) in future

The CWB keeps a stack of output (and input) files; the two forms of the output command push a new file onto the stack and pop the stack, respectively. So nested output commands will do what you probably expect.

See also: save, savemeije, savefc2, input

save : save the current environment in a file

Usage: `save "file";`

This can be read as a short way of writing:

output "file";

print;

output;

To get a CWB session to read such a file, use the input command.

See also: savemeije, savefc2, input

8.5 Working with a single agent

From basic module

diverges : does the agent contain an unguarded occurrence of @?

Usage: `diverges A;`

Returns true iff there is an unguarded occurrence of bottom (@). Notice especially that an agent which can just do taus for ever does not diverge in this sense!

Synonyms: div

prefixform : print an agent in prefix form

Usage: `prefixform A;`

This produces an agent which is strongly bisimilar to A and which is a sum of prefixes, e.g. a.A + b.B + c.C.

Warning: identifiers appearing in the definition of A may get replaced by their current definitions, so if you later change the definition of such an identifier, A may cease to be bisimilar to what prefixform returned.

Synonyms: pf

See also: normalform, sim, transitions

From option module AgentExtra (TCCS only)

transitions : list the (single-step) transitions of an agent

Usage: `transitions A;`

Synonyms: tr

See also: sim, derivatives, closure, obs, vs, random, init

graph : list the transition graph of an agent

Usage: `graph A;`

Simple-mindedly, and not particularly efficiently, calculates the whole transition graph of the agent. You do not want to do this for large agents, let alone infinite ones!

Because of a feature of how graphs are calculated and stored, internal actions don't show what communication they come from (in CCS, you see tau, never tau⟨a⟩). Use "transitions" to get this information when you want it.

size : find the number of states of a finite-state agent

Usage: `size A;`

The number of states of an agent is not as clear a concept as you might think: treat the number as a rough indication of size, only.

See also: states, statesexp, statesobs

closure : find the weak derivatives of an agent via a trace

Usage: `closure(a,A);` lists the agents reachable from A via action a

Usage: `closure(a,b,...,A);` ditto, but via the observation a,b,...

The first case lists all B such that A =a=⟩ B, where a may be an observable action, tau, or eps.

In the second case, tau or eps may be included in the trace, but will be ignored: if the trace is l, you get all B such that A =^l=⟩ B.

(Strange? yes, I think so!)

Synonyms: cl,actcl

See also: sim, transitions, derivatives, obs, vs, random, init

deadlocks : find dead- or live-locked states and traces leading to them

Usage: `deadlocks A;`

A state is deadlocked if after reaching it no observable action will ever be possible. This definition follows Milner's book, but includes what is often known as livelock.

For each such state, we give one sequence of actions by which it may be reached (just one, even if there are many ways of reaching that state).

Synonyms: fd

See also: deadlocksobs, findinit, findinitobs

deadlocksobs : find dead- or live-locked states with observations

Usage: `deadlocksobs A;`

A state is deadlocked if after reaching it no observable action will ever be possible. This definition follows Milner's book, but includes what is often known as livelock.

For each such state, we give one sequence of observable actions by which it may be reached (just one, even if there are many ways of reaching that state).

Synonyms: fdobs

See also: deadlocks, findinit, findinitobs

derivatives : find the derivatives of an agent via a given action

Usage: `derivatives(a,A);`

If there are several derivations of A -a-⟩ B, B will still only be listed once.

Synonyms: dr, actders

See also: sim, transitions, closure, obs, vs, random, init

findinit : find states with a given set of next observable actions

Usage: `findinit(a,b,...,A);`

So, for example, findinit (a,b,A) is equivalent to findinit (b,a,A), and will give states reachable from A from which a and b are the only possible next observable actions. That is, a state D reachable from A should appear if D =a=⟩ and D =b=⟩ and for no other observable c does D =c=⟩. Note the weak transitions here! It's not true that init(D) will be { a,b }. Of course only observable actions are legal arguments.

For each such state, we give a trace leading to it.

See also: findinitobs, deadlocks

findinitobs : find states with a given set of next observable actions

Usage: `findinitobs(a,b,...,A);`

Exactly like findinit, except that for each state with the given set of next observable actions, we give an observable trace leading to the state. That is, we take the the trace produced by findinit and delete all taus!

See also: findinit, deadlocks

freevars : list the free agent variables of an agent

Usage: `freevars A;`

Tells you nothing about free set or relabelling variables! However, may be useful for checking that you've got recursive definitions right.

Synonyms: fv,freevariables

init : find the observable actions an agent can perform immediately

Usage: `init A;`

See also: sim, transitions, derivatives, closure, obs, vs, random

normalform : print an agent in normal form

Usage: `normalform A;`

This is a slight misnomer, and this command is not really intended for the end user. It takes what's returned by prefixform and simplifies further, still returning something strongly bisimilar to A.

See also: prefixform

obs : find observations of a given length, and their results

Usage: `obs(n,A);`

Lists all observations of length n, and the final state of each. If there are several states which can be reached by the same observation, all will be listed.

See also: sim, transitions, derivatives, closure, vs, random, init

random : give a pseudo-random observation of at most a given length

Usage: `random(n,A);`

Selects a pseudo-random observation of length $\langle$= n.

See also: sim, transitions, derivatives, closure, obs, vs, init

sort : find the syntactic sort of the agent

Usage: `sort A;`

See also: freevars

stable : is the agent stable?

Usage: `stable A;`

An agent is stable iff it can not initially perform an unobservable action. (tau for TCCS, 1 for SCCS)

states : list the state-space of a finite-state agent

Usage: `states A;`

See also: size, statesexp, statesobs

statesexp : list the state-space, and a trace leading to each state

Usage: `statesexp A;`

Lists the state space of an agent, if it is finite. For each state a sequence of actions by which it may be reached from the initial state will be printed.

See also: size, statesobs

statesobs : list the state-space and an observation leading to each state

Usage: `statesobs A;`

Lists the state space of an agent, if it is finite. For each state a sequence of observable actions by which it may be reached from the initial state will be printed.

See also: size, statesexp

vs : find observations of a given length

Usage: `vs(n,A);`

Lists all observations (visible sequences) of length n from A. This is exactly like obs(n,A), except that we don't care what the final state is.

See also: sim, transitions, derivatives, closure, obs, random, init

From optional module Equivalences

min : minimise an agent with respect to weak bisimulation

Usage: `min(X,A);` minimise A, binding the result to X

If your variable is X, the CWB will use identifiers like XminStateNNN for the states of the minimised agent. Any existing bindings will be overridden.

See also: eq

8.6 Comparing two agents

Each of these commands takes two agents. None of them will terminate if either agent given as a parameter has an infinite state space.

From optional module Equivalences

branchingeq : are agents branching bisimilar?

Usage: `branchingeq(A,B);`

cong : are two agents observationally congruent (equal)?

Usage: `cong(A,B);`

Returns true iff

whenever A -a-⟩ A' there exists B' st B =a=⟩ B' and A' ∼∼ B' and

whenever B -a-⟩ B' there exists A' st A =a=⟩ A' and A' ∼∼ B'

(where A' ∼∼ B' means that A' and B' are weakly bisimilar)

See also: eq, strongeq

diveq : are two agents divergence equivalent?

Usage: `diveq(A,B);`

This is observational equivalence which respects divergence. A state is divergent in this sense if it can perform an infinite sequence of unobservable actions (tau for CCS or 1 for SCCS), or if it can reach an unguarded occurrence of the divergent agent, @, through some sequence of unobservable actions.

eq : are two agents observationally equivalent (weakly bisimilar)?

Usage: `eq(A,B);`

Returns true iff the agents are related by some weak bisimulation. R is a weak bisimulation iff PRQ implies

whenever P -a-⟩ P' there exists Q' st Q =^a=⟩ Q' and P' R Q' and

whenever Q -a-⟩ Q' there exists P' st P =^a=⟩ P' and P' R Q'.

See also: cong, strongeq

pb : print largest weak bisimulation over the state-space of two agents

Usage: `pb(A,B);`

If the agents are weakly bisimilar, then they will appear in the same bisimulation class.

See also: eq

strongeq : are two agents strongly bisimilar?

Usage: `strongeq(A,B);`

Returns true iff A and B are related by some strong bisimulation. R is a strong bisimulation iff PRQ implies

whenever P -a-⟩ P' there exists Q' st Q -a-⟩ Q' and P' R Q' and

whenever Q -a-⟩ Q' there exists P' st P -a-⟩ P' and P' R Q'.

See also: eq, cong

From optional module Testing

mayeq : are two agents may equivalent, i.e. trace equivalent?

Usage: `mayeq(A,B);`

maypre : are two agents related by the may preorder?

Usage: `maypre(A,B);`

Returns true iff the language of A is contained in the language of B.

See Hennessy: Algebraic Theory of Processes, MIT Press.

See also: mayeq, mustpre, testpre

musteq : are two agents must equivalent?

Usage: `musteq(A,B);`

Returns true iff A and B are related both ways round by mustpre.

See Hennessy: Algebraic Theory of Processes, MIT Press.

mustpre : are two agents related by the must preorder?

Usage: `mustpre(A,B);`

Returns true iff for every s in the language of B (including the empty s) if A converges at s (i.e. you can't get to a divergent state by following observable path s) then B converges at s and every acceptance set of B after s contains some acceptance set of A after s.

(Here "divergence" includes performing an infinite sequence of tau moves, as well as @.)

See Hennessy: Algebraic Theory of Processes, MIT Press.

See also: musteq, maypre, testpre

testeq : are two agents testing equivalent (i.e. failures equivalent)?

Usage: `testeq(A,B);`

Returns true iff A and B are both must and may equivalent, that is, if they are related both ways round by both mustpre and maypre.

See Hennessy: Algebraic Theory of Processes, MIT Press.

testpre : are two agents related by the testing preorder?

Usage: `testpre(A,B);`

Returns true iff maypre (A, B) and mustpre (A, B).

See Hennessy: Algebraic Theory of Processes, MIT Press

See also: testeq, maypre, mustpre

From optional module Contraction

contraction : are agents related by the contraction pre-order?

Usage: `contraction(A,B);`

Returns true iff A and B are related by some contraction. C is a contraction iff PCQ implies

whenever P -a-⟩ P' there exists Q' s.t. Q =^a=⟩ Q' and P' C Q' and

whenever Q -a-⟩ Q' there exists P' s.t. (P -a-⟩ P' or P -^a-⟩ P') and P' C Q'

(P and Q are weakly bisimilar, and P need never do more tau moves than Q)

See Milner "Contractions", RM 13, Handwritten notes, dated 23 March 1990.

From optional module Divergence

pre : are two agents related by the (weak) bisimulation preorder?

Usage: `pre(A,B);`

This is NOT one half of a weak bisimulation relation!

Returns true iff A and B are related by some weak bisimulation preorder. R is a weak bisimulation preorder iff PRQ implies

Q is no more divergent than P and

P=^a=⟩P' implies Q=^a=⟩Q' and P' R Q' and

either P diverges globally or Q=^a=⟩Q' implies P=^a=⟩P' and P' R Q' (except that we allow Q=^a=⟩Q' to go unmatched if P is locally divergent at a)

We set divergence by maximal fixpoint: a thing gets to be marked not globally divergent by not having @ as a summand or component (even under restriction/relabelling), and not having any tau-successors, or having every tau-successor satisfy that. Then P diverges at a iff P=a=⟩P' globally divergent. (So, e.g., @ does not diverge at tau.)

See also: strongpre, precong

precong : are two agents related by the bisimulation precongruence?

Usage: `precong(A,B);`

This is NOT one half of a bisimulation congruence relation.

This is like pre (q.v.) except that we fiddle the =^a=⟩ graph such that the roots P and Q don't have tau arrows to themselves but Q does have an eps arrow to itself, which is deemed to match a tau-arrow from P to a divergent state P' which is related to Q. (We copy the roots, in fact, so if P or Q occur as derivatives of themselves then in that context they will have tau loops like all other nodes. The effect is simply to treat the first transition specially, just as in equality vs w. bisim.)

See also: strongpre, pre

strongpre : are two agents related by the strong bisimulation preorder?

Usage: `strongpre(A,B);`

This is NOT one half of a strong bisimulation relation!

Returns true iff A and B are related by some strong bisimulation preorder. R is a strong bisimulation preorder iff PRQ implies

Q is no more divergent than P and

P-a-⟩P' implies Q-a-⟩Q' and P' R Q' and

either P diverges globally or Q-a-⟩Q' implies P-a-⟩P' and P' R Q' (except that we allow Q-a-⟩Q' to go unmatched if P -a-⟩ some globally divergent state.)

Here "P diverges globally" means P has @ as a summand or component, (possibly after variable lookup, possible under restriction/relabelling) "P diverges locally at a" means P -a-⟩ P' which diverges globally. "Q is no more divergent than P" means if Q diverges (globally, at a) so does P.

So if there are no @s in sight, this is the same as strongeq.

See also: pre, precong

twothirdseq : are agents related both ways by 2/3 bisimulation preorder?

Usage: `twothirdseq(A,B);`

twothirdspre : are two agents related by the 2/3 bisimulation preorder?

Usage: `twothirdspre(A,B);`

Returns true iff A and B are related by some two-thirds preorder. R is a two-thirds preorder iff PRQ implies

whenever P -a-⟩ P' there exists Q' such that Q -a-⟩ Q' and P'RQ' and

initial actions of P = initial actions of Q

Note that twothirdseq (P,Q) iff twothirdspre (P,Q) and twothirdspre (Q,P)

See also: twothirdseq

From optional module Logic

dftrace : find a trace distinguishing two agents

Usage: `dftrace(A,B);`

Produces a sequence of observable actions which can be performed by one agent but not by the other, unless the agents are trace (may) equivalent.

See also: dfstrong, dfweak, mayeq

8.7 Working with logic

All from optional module Logic.

checkprop : model-checking: does an agent satisfy a formula?

Usage: `checkprop(A,P);`

Uses the new games-based local model-checking algorithm. Generates a winning strategy for the corresponding Stirling model checking game, and if you have the toggle "hateGames" set to false (the default) it will offer to play this strategy against you. We hope you will find this useful, especially when the CWB gives you an answer you did not expect. Feedback especially welcome!

See also: logic,toggle,checkpropold

checkpropold : DEPRECATED modelchecking using old algorithm

Usage: `checkpropold(A,P);`

This command uses the old (v7.0 and earlier) implementation of tableau-based local model-checking. This is always(?) slower than the new games based algorithm; it is included so that if you suspect the new algorithm of giving the wrong answer or being too slow you have something to compare with. Please let me know if there are circumstances where this algorithm is better than the old one; by default it will disappear from the next release.

See also: checkprop,logic

dfstrong : find a strong HML formula distinguishing two agents

Usage: `dfstrong(A,B);`

Produces a strong HML formula which distinguishes between the two agents if they are not strongly bisimilar. The formula is true of the first agent and false of the second.

Synonyms: df

See also: dfweak, dftrace

dfweak : find a weak HML formula distinguishing two agents

Usage: `dfweak(A,B);`

Produces a weak HML formula which distinguishes between the two agents if they are not weakly bisimilar. The formula is true of the first agent and false of the second.

See also: dfstrong, dftrace

8.8 Simulating the behaviour of an agent.

From optional module Simulation.

sim : simulate an agent interactively

Usage: `sim A;`

(If you are running under Solaris 2.x using X, you might like to consider using the prototype graphical simulator instead. See the CWB WWW home page for more information.)

The following commands are available in the simulator:

sim B; Stops current simulation, deletes breakpoints, starts simulating B instead.

menu; Lists the (one-step) transitions from the current state.

n; Follows the transition labelled with the integer n in the menu.

random n; Simulates no more than n steps, choosing transitions at random. Halts if it reaches a deadlocked agent, or one which can perform an action which is on the list of breakpoints (see the break command).

; Short for random 1;

history; Lists the states and transitions by which the current state was reached.

return n; Stops current simulation, simulates the agent reached at step n of the current simulation instead. (E.g. return 0; starts again at the beginning.)

break a,b...; Sets breakpoints on actions a, b etc. See random.

lb; Lists all the breakpoints.

db a,b,...; Deletes a,b etc. from the list of breakpoints.

bind A; Bind the current state to identifier A: if the current state is B, this is equivalent to agent A = B; at the CWB main prompt.

help; Prints out a summary of these simulation commands.

quit; Stops simulation, returns to the CWB main prompt.

Acknowledgement

This document is a (thorough) reworking of the document *"The Concurrency Workbench: Operating Instructions"*, Rance Cleaveland, Joachim Parrow & Bernhard Steffen, University of Edinburgh Computer Science Technical Note LFCS-TN-10, 1988. Any similarities between the present document and this earlier document are purely intentional. (Though probably few remain.)

9 References

This section contains an incomplete list of references for those wishing to learn more about the theory underlying the Workbench.

Firstly, the standard reference for CCS is the following.

R. Milner, **Communication and Concurrency**, Prentice Hall International, 1989.

Two further references on Process Algebras which describe the theory of the testing equivalences and preorders (defined in the case of the second reference here in terms of failures) are as follows.

M. Hennessy, **Algebraic Theory of Processes**, MIT Press, 1988.

C.A.R. Hoare, **Communicating Sequential Processes**, Prentice Hall International, 1985.

The theory behind the timing constructs used in the CWB is provided by the following.

F. Moller, C. Tofts, *"A Temporal Calculus of Communicating Systems"*, Proceedings of CONCUR'90, pp401-415, Lecture Notes in Computer Science 458, Springer-Verlag, 1990.

The theory behind the prebisimulation preorders and divergence equivalence computed in the Workbench are given in the following.

D. Walker, *"Bisimulations and Divergence in CCS"*, Proceedings of the Third Annual Symposium on Logic in Computer Science, pp186-192, Computer Society Press, 1988.

The modal μ-calculus is due to Pratt and Kozen, as described in the following.

D. Kozen, *"Results on the Propositional μ-Calculus"*, Theoretical Computer Science 27, pp333-354, 1983.

The utility of modal logics for CCS is discussed in the following.

C. Stirling, *"An Introduction to Modal and Temporal Logics for CCS"*, Proceedings of the 1989 Joint UK/Japan workshop on Concurrency, pp2-20, Lecture Notes in Computer Science 491, Springer-Verlag, 1991.

Details of the original design of the Workbench can be found in the following. (Warning: a lot has changed since this was written!)

R. Cleaveland, J. Parrow, B. Steffen, *"The Concurrency Workbench: A Semantics-based Verification Tool for Finite-state Systems"*, Proceedings of the Workshop on Automated Verification Methods for Finite-state Systems, Lecture Notes in Computer Science 407, Springer-Verlag, 1989.

Contact

If you have any queries, comments or difficulties with the CWB, please get in touch with:

Perdita Stevens
Division of Informatics
University of Edinburgh
The King's Buildings
Mayfield Road
Edinburgh EH9 3JZ
SCOTLAND

email: Perdita.Stevens@dcs.ed.ac.uk