Concurrency Workbench and Process Algebras

- Process algebra CCS
- μ-calculus and its relationship with CTL
- Verification approaches
- Example: simple protocol
- Doing this in CWB

Other: plan for the rest of the semester; projects; assignments

Concurrency Workbench

- Specify: a set of (communicating) concurrent processes (using CCS or SCCS)
- Use various verification methods to check that the processes meet their specification.
- The system is designed to be easy to extend (so CW in Manual $\neq$ CW in Paper)

Overview of CCS

Processes are called *agents*, built from a set of *actions*. Actions can be:
- Observable (or *communication*), marked by letters a, b, etc., and
- Unobservable (or silent), marked by τ.

Observable actions:
- a, b, ... - input actions
- $\overline{a}$, $\overline{b}$, ... - output actions

Input action a and output action $\overline{a}$ are *complimentary*, reflecting input and output on the "port" a (used to represent synchronization).

Some standard operators

Nil - Terminated process

$\perp$ - Undefined process. Its behavior is unknown ("don't care").

$a.P$ - Process performs action a and then acts exactly like P.

More on that:

$\stackrel{a}{\rightarrow}$ - transition relation.

$p \stackrel{a}{\rightarrow} p\prime$ holds when p can evolve into $p\prime$ by performing action a. $p\prime$ is called an *a-derivative* of p.

$a.p \stackrel{a}{\rightarrow} p$ holds for any p.

Standard Operators (Cont'd)

$+$ - Choice. $p+q$ - either p or q will get performed. $p+q \stackrel{a}{\rightarrow} p'$ if either $p \stackrel{a}{\rightarrow} p'$ or $q \stackrel{a}{\rightarrow} p'$.

$|$ - Parallel composition. The agent $p\,|\,q$ behaves like the "interleaving" of p and q with the possibility of complementary actions synchronizing to produce a τ action.

Example:

$\backslash L$ - Restriction on (finite) set L of actions. $p\backslash L$ behaves like p with the exception that no actions in L are allowed.

$[f]$ - Relabeling of f, which maps actions to actions. $p[f]$ behaves like p with actions renamed by function f.

CCS Operators - Formal Semantics

$$
\begin{array}{rcl}
a.p \stackrel{a}{\rightarrow} p & & \\
p \stackrel{a}{\rightarrow} p' & \Rightarrow & p+q \stackrel{a}{\rightarrow} p' \\
q \stackrel{a}{\rightarrow} q' & \Rightarrow & q+q \stackrel{a}{\rightarrow} q' \\
p \stackrel{a}{\rightarrow} p' & \Rightarrow & p|q \stackrel{a}{\rightarrow} p'|q \\
q \stackrel{a}{\rightarrow} q' & \Rightarrow & p|q \stackrel{a}{\rightarrow} p|q' \\
p \stackrel{a}{\rightarrow} p', q \stackrel{\bar{a}}{\rightarrow} q' & \Rightarrow & p|q \stackrel{\tau}{\rightarrow} p'|q' \\
p \stackrel{a}{\rightarrow} p',\ a, \bar{a} \notin L & \Rightarrow & p\backslash L \stackrel{a}{\rightarrow} p'\backslash L \\
p \stackrel{a}{\rightarrow} p' & \Rightarrow & p[f] \stackrel{f(a)}{\rightarrow} p'[f] \\
p_A \stackrel{a}{\rightarrow} p' & \Rightarrow & A \stackrel{a}{\rightarrow} p'
\end{array}
$$

Here p_A is the agent expression bound to identifier A.

Specification examples - Buffers

BUF_n - a buffer of capacity n.

$$\begin{aligned}
\mathsf{BUF}_n &= \mathsf{BUF}_n^0 \\
\mathsf{BUF}_n^0 &= \mathit{in}.\mathsf{BUF}_n^1 \\
\mathsf{BUF}_n^i &= \mathit{in}.\mathsf{BUF}_n^{i+1} + \overline{out}.\mathsf{BUF}_n^{i-1} \\
&\quad \text{for } i = 1, \ldots, n-1 \\
\mathsf{BUF}_n^n &= \overline{out}.\mathsf{BUF}_n^{n-1}
\end{aligned}$$

Transition graph for BUF_n.

Specification examples - Buffers

CBUFn - a compositional buffer of capacity n.

$$\mathsf{CBUF}_n = (\mathsf{BUF}_1[x_1/\mathit{out}] \mid \underbrace{\ldots \mid \mathsf{BUF}_1[x_i/in, x_{i+1}/out] \mid \ldots}_{i=1,\ldots,n-2} \mathsf{BUF}_1[x_{n-1}/in]) \setminus \{x_1, \ldots, x_{n-1}\}$$

Transition graph for CBUF_2.

Notion of Observation

Transition graphs make a transition on every time "tick" (even if it is just τ). If timing is removed, we might be interested in just *observable* transitions.

Definition:
- $p \stackrel{\epsilon}{\Rightarrow} p\prime$ iff $p \stackrel{\tau *}{\rightarrow} p\prime$

(transitive and reflexive closure of $\stackrel{\tau}{\rightarrow}$)
- $p \stackrel{a}{\Rightarrow} p\prime$ iff $p \stackrel{\epsilon}{\Rightarrow}\stackrel{a}{\rightarrow}\stackrel{\epsilon}{\Rightarrow} p\prime$

(relational products of $\stackrel{\epsilon}{\Rightarrow}$ and $\stackrel{a}{\rightarrow}$)

Can compute observation graphs, which takes $O(n^3)$, where n - number of nodes in the graph.

Observation Graph for $CBUF_2$

For clarity, ϵ-loops - one self-looping edge from each node - are omitted from the following graph.

Figure 5, page 8

μ-calculus

Specifications can be written in a modal logic based on the *propositional* μ-calculus.

Syntax of formulas:

$$
\begin{array}{rcl}
\Phi & ::= & tt \mid ff \mid X \\
 & \mid & \neg\Phi \;\mid\; \Phi \vee \Phi \;\mid\; \Phi \wedge \Phi \;\mid\; \Phi \Rightarrow \Phi \\
 & \mid & < a > \Phi \;\mid\; [a]\Phi \;\mid < . > \Phi \;\mid\; [.]\Phi \\
 & \mid & B \textit{ arg-list} \\
 & \mid & \nu X.\Phi \;\mid\; \mu X.\Phi
\end{array}
$$

X ranges over variables,
a - over actions,
B - over user-defined macro identifiers,
arg-list - over lists of actions and/or formulas that B requires in order to produce a proposition,
tt and ff hold on every node and no node, respectively.

Semantics of μ-calculus formulas

Constructors $< a >$, $[a]$, $< . >$ and $[.]$ - to reason about edges leaving a node.

A node n satisfies:
- $< a > \Phi$ if it has an a-derivative satisfying Φ
- $[a]\Phi$ if *all* of its a-derivatives satisfy Φ

In the case that n has no a-derivatives, n trivially satisfies $[a]\Phi$

. acts like a "wild-card" action in $[.]$, $< . >$.
n satisfies:
- $< . > \Phi$ if it satisfies $< a > \Phi$ for some a
- $[.]\Phi$ if it satisfies $[a]\Phi$ for all a

Semantics of μ-calculus formulas (cont'd)

Formulas of type $\nu X.\Phi$ and $\mu X.\Phi$ are recursive formulas, representing the greatest- and least-fixpoints, respectively.

- $\nu X.\Phi = \bigwedge_{i=0}^{\infty} \Phi_i$, where
  Φ_0 is tt and
  $\Phi_{i+1} = \Phi[\Phi_i/X]$ (substitute Φ_i for all free occurrences of X in Φ)
- $\mu X.\Phi = \bigvee_{i=0}^{\infty} \hat{\Phi}_i$, where
  $\hat{\Phi}_0$ is ff and
  $\hat{\Phi}_{i+1} = \Phi[\hat{\Phi}_i/X]$

Restriction: Φ should be such that any free occurrences of X appear positively.

μ-calculus and CTL

Formulas in general are unintuitive and difficult to understand. But using macros facility, they can be "coded up" into better-understood operators like CTL (its logic is a subset of μ-calculus). For example,

$$\begin{aligned} AG\Phi &= \nu X.(\Phi \wedge [.]X) \\ AF\Phi &= \mu X.(\Phi \vee (< . > tt \wedge [.]X)) \\ AU1\Phi\Psi &= \nu X.(\Phi \vee (\Psi \wedge [.]X)) \\ AU2\Phi\Psi &= \mu X.(\Phi \vee (\Psi \wedge < . > tt \wedge [.]X)) \end{aligned}$$

You can do similar encoding for Assignment 3.

How is Model-Checking Done?

- Semantics of propositions
- Tableau-based checking:
  - Notion of tableau
  - Rules
  - Example
- Implementation of model-checking
- Running times

Terminology

$\mathcal{A}$ - a set of *atomic formulas A* ...
$\mathcal{V}$ (disjoint from $\mathcal{A}$) - a set of *propositional variables X* ...
Act - a set of *actions a* ...
Formulas are Φ ...
$\mathcal{T}$ - a set of states

- $\Gamma \prec \Phi$ if Γ is a strict subformula of Φ.
- *environment* - a mapping of variables to sets of states as a means of interpreting free propositional variables.
- $e[X \mapsto S]$ - the environment e with X "updated" to S.
- Use *sequents* of the form $H \vdash s \in \Phi$, where s is a state, Φ is a formula, and H is a set of *hypotheses* of the form $s\prime : \Gamma$, for $s\prime$ a state and Γ a *closed recursive formula*.

Semantics of propositions

$$
\begin{array}{rcl}
[\![A]\!]e & = & V(A) \\
[\![X]\!]e & = & e(X) \\
[\![\neg\Phi]\!]e & = & \mathcal{T} - [\![\Phi]\!]e \\
[\![\Phi_1 \vee \Phi_2]\!]e & = & [\![\Phi_1]\!]e \cup [\![\Phi_2]\!]e \\
[\![< a > \Phi]\!]e & = & \pi_a([\![\Phi]\!]e), \text{ where} \\
 & & \pi_a(S) = \{s\prime \mid \exists s \in S. s\prime \stackrel{a}{\rightarrow} s\} \\
[\![\nu X.\Phi]\!]e & = & \bigcup\{S \subseteq \mathcal{T} \mid S \subseteq [\![\Phi]\!]e[X \mapsto S]\}
\end{array}
$$

Idea of tableau

• Theorem: $H \vdash s \in \Phi$ has a successful tableau if and only if $H \vdash s \in \neg\Phi$ has no successful tableau.

• Idea: start with property (or negated property), apply rules R1-R8 and DR1-DR3 (below) in top-down fashion until *all* leaves are successful. A leaf is successful if and only if one of the following holds:

1. $\Phi \in \mathcal{A}$ and $s \in (V(\Phi))$.
2. Φ is $\neg A$ for some $A \in \mathcal{A}$ and $s \notin V(A)$.
3. Φ is $\neg < a > \Phi\prime$ for some a and $\Phi\prime$.
4. Φ is $\nu X.\Phi\prime$ for some X and $\Phi\prime$.
5. Sequents of form $H \vdash s \in \mathit{True}$ are successful.
6. Leaves of the form $H \vdash s \in [a]\Phi$ are successful.

Note: $H \vdash s \in \neg < a > \Phi$ is a leaf only when s has no a-derivatives, while $H \vdash s \in \nu X.\Phi$ is a leaf only when $s : \nu X.\Phi \in H$.

Rules

see Figure 3 on p. 730 of Acta Informatica paper

Rules (Cont'd)

see Figure 4 on p. 732 of Acta Informatica paper

Rules, Etc.

R7 and R8 require that in order to establish that a state enjoys a (negated) recursive property, it is sufficient to establish that it enjoys the (negated) unrolling of the property, provided that the assumptions involving the formulas having the recursive formula as a subformula are removed or *discharged* from the hypothesis list.

Other results:
1. (Finiteness) If models are finite, their tableaux are finite.
2. (Soundness and completeness) $H \vdash s \in \Phi$ has a successful tableau if and only if $s \in [\![\Phi]\!]^H$.

Example

See Figure 5 on p. 732 of Acta Informatica paper

Simple Implementation of model-checking

fun *check1'*$(H \vdash s \in \Phi) =$
  case Φ is
  $A \in \mathcal{A} \rightarrow$ return $(s \in \mathcal{V}(\mathcal{A}))$
  $X \in \mathcal{V} \rightarrow$ error
  $\neg\Phi' \rightarrow$ return not (*check1'*$(H \vdash s \in \Phi')$)
  $\Phi_1 \vee \Phi_2 \rightarrow$ return (*check1'*$(H \vdash s \in \Phi_1)$
    or *check1'*$(H \vdash s \in \Phi_2)$)
  $< a > \Phi' \rightarrow$ for each $s' \in \{s'$, s.t. $s \xrightarrow{a} s'\}$ do
    if *check1'*$(H \vdash s' \in \Phi')$ then
      return true;
    else return false
  $\nu X.\Phi' \rightarrow$ let $H' = \{s' : \Gamma \mid \Phi\neg \prec \Gamma\}$ in
    return (*check1'*$(H \cup \{s : \Phi\} \vdash$
      $s \in \Phi'[\Phi/X])$)
end
fun *check1*$(s \in \Phi) =$ *check1'*$(\emptyset \vdash s \in \Phi)$

Running times

- Algorithm has exponential running time even for formulas having no recursive subformulas, owing to the possibility of nested modal operators.
- Possible optimization: store results of sequents whose truth has already been determined
- Running time is $O((|S| \times |\Phi|)^{id(\Phi)+1})$:
  - $id(\Phi)$ – *interconnection depth* of Φ, measure of the degree of mutual recursion in Φ
  - Φ – formula under verification
  - S – number of states in transition system

Verification example

Want to prove that CBUF$_n$, for a particular n, is deadlock free.
- Define a macro $Deadlock = \neg. < . > tt$ (true in states that cannot perform any actions)
- Using model-checker, check $AG\neg Deadlock$

Want to prove liveness property - buffer will eventually get engaged in an *in* or an $\overline{out}$
- Check $(AG((AF < in > tt) \vee (AF < \overline{out} > tt))$

Equivalence Checking

Idea - node matchig. Two transition graphs are equivalent if their nodes can be matched such that
1. two matched nodes have compatible information fields
2. if two nodes are matched and one has an a-derivative, then the other must have a matching a-derivative
3. the root nodes of two transition graphs are matched.

Equivalence Checking - formal definition

Let G_1 and G_2 be transition graphs with node sets N_1 and N_2, respectively. Let $N = N_1 \cup N_2$, and let $\mathcal{C} \subseteq N \times N$ be an equivalence relation reflecting a notion of "compatibility" between information fields. A *$\mathcal{C}$-bisimulation* on G_1 and G_2 is a relation $\mathcal{R} \subseteq N \times N$ such that $< m, n > \in \mathcal{R}$ implies that:
1. if $m \stackrel{a}{\to} m\prime$ then $\exists n\prime : n \stackrel{a}{\to} n\prime$ and $< m\prime, n\prime > \in \mathcal{R}$, and
2. if $n \stackrel{a}{\to} n\prime$ then $\exists m\prime : m \stackrel{a}{\to} m\prime$ and $< m\prime, n\prime > \in \mathcal{R}$, and
3. $< m, n > \in \mathcal{C}$

If root nodes can be related by $\mathcal{C}$-bisimulation, then two transition graphs are $\mathcal{C}$-equivalent.

Equivalence Checking - Cont'd

Many equivalences are instances of $\mathcal{C}$-equivalence combined with graph transformations. For example, *observation equivalence* corresponds to equivalence on observation graphs where $\mathcal{C}$ is replaced by $U = N \times N$.

Similarly, can define *testing equivalence* for acceptance graphs (see paper).

BUF$_n$ and CBUF$_n$ are observationally equivalent for each n. Notation:

$$\text{BUF}_n \approx \text{CBUF}_n, \forall n$$

Preorder Checking

A process A is "more defined than" a process B if A has the same behavior as B except for the holes in B. The preorder algorithm determines if a process is more defined than its specification. One transition graph is less than another if the states of first can be matched to the second such that:

1. the information field of the "lesser" node must be "less" than the "greater".
2. if the "greater" node has "valid" a-transitions, then each a-transition of the "lesser" must be matched by some a-transitions of the "greater".
3. if the "lesser" node has "viable" a-transitions, then each transition of the "greater" must be matched by some a-transition of the "lesser".
4. start state of the "lesser" and the "greater" must be matched.

Preorder Checking

Weak divergence preorder, $\underset{\approx}{\sqsubset}$, is obtained from the observation graph where
- "viable" holds for all nodes
- "valid" stands for $\{n \mid n \Downarrow a\}$. $n \Downarrow a$ holds if n is not globally divergent and cannot be triggered by means of an a-action to reach a globally divergent state.

This interpretation is based upon regarding divergent states as being *underspecified*. So, $\perp$ allows any process as a correct implementation. Preorder $\underset{\approx}{\sqsubset}$ coincides with $\approx$ for complete specifications.
Actually, when left-hand side process is completely specified, then so is the right-hand side process.

Other preorders can be defined similarly (see paper).

A Simple Protocol

Service specification of the protocol requires that any message sent must be received before a second message may be sent:

$$SRV = s.\bar{r}.\mathrm{SRV}$$

Graph:

Protocol specification - two processes, a sender and a receiver, and a medium connecting them.

Protocol Design

Create the following (logical) design:

Define:

$$\begin{array}{rcl} SND & = & s.\overline{from}.ack_{to}.SND \\ MDM & = & from.\overline{to}.MDM + ack_{from}.\overline{ack_{to}}.MDM \\ RCV & = & to.\bar{r}.\overline{ack_{from}}.RCV \\ PROT & = & SND \mid MDM \mid RCV \\ & & \backslash\{from, to, ack_{from}, ack_{to}\} \end{array}$$

Restriction operator means that these actions are internal.

- Can show $PROT \approx SRV$.

Another Protocol Design

Produce a partial definition, reflecting the fact that there may be different implementations for the medium still leading to a correct overall implementation of the service specification.

$$
\begin{aligned}
PM &= from.(\overline{to}.PM + ack_{from}.\bot) + \\
   &\quad ack_{from}.(\overline{ack_{to}}.PM + from.\bot) \\
PP &= SND \mid PM \mid RCV \\
   &\quad \backslash\{from, to, ack_{from}, ack_{to}\}
\end{aligned}
$$

Now define an implementation, consisting of two one-piece buffers, running in parallel: one for messages, one for acknowledgments.

$$
\begin{aligned}
NM &= MB \mid AB \\
MB &= from.\overline{to}.MB \\
AB &= ack_{from}.\overline{ack_{to}}.AB \\
N\_PROT &= SND \mid NM \mid RCV \\
   &\quad \backslash\{from, to, ack_{from}, ack_{to}\}
\end{aligned}
$$

Verification for this Example

- Can show that $N\_PROT \approx SRV$:
  - $PP \approx SRV$
  - $PM \overset{\sqsubseteq}{\approx} NM$
  - PP never reaches an underspecified state (via model-checking)

Therefore, $N\_PROT \approx PP$ and hence $N\_PROT \approx SRV$

Model-checking

Define the following macros:

$$\begin{aligned} AG\Phi &= \nu X.(\Phi \wedge [.]X) \\ \text{Can } \Phi &= \mu X.(X. < \Phi > tt \mid < \tau > X) \\ \text{Can't } \Phi &= \neg \text{Can } \Phi \end{aligned}$$

Now, check:

- $S_1 = AG((\text{Can } s) \mid (\text{Can}\bar{r}))$
either a s or a $\bar{r}$ can always happen
- $S_2 = AG([s](\text{Can } \bar{r})\&[\bar{r}](\text{Can } s))$
after a s, a process can $\bar{r}$ and vice versa
- $S_3 = AG([s](\text{Can't } s)\&[\bar{r}](\text{Can't } \bar{r}))$
two consecutive s's or $\bar{r}$'s cannot happen
- $S_4 = \text{Can } s$
s must eventually be possible

Working with CWB

To run CWB:

From command line using

`cwb`

or from emacs (see `man cwb` for installation instructions).

Examples of CWB specifications:

`/local/share/cwb/examples/ccs`

CWB Syntax

Identifiers - `(A-Z)(A-Z, a-z, 0-9, ?, !, _, ', `, -, #)*`
Actions - `[']` `(a-z)(A-Z, a-z, 0-9, ?, !, _, ', `, -, #)*`

- Action τ is represented as `tau`
- Inverse actions like $\overline{a}$ are represented as `'a`
- Constant 0 is represented as `0`
- Constant `@` represents agent $\perp$ (divergence)

The rest is identical to CCS. Operations include action prefixing, summation, parallel composition, restriction, relabelig.

Concurrency Workbench - Design

Design of CW - 3 layers
- First layer manages interaction with the user and contains the basic definition of process semantics in terms of *labeled transition graphs*
- Second layer provides transformations that may be applied to transition graphs (so we can change the semantic model of processes under consideration)
- Third layer includes basic algorithms to establish whether the process meets its specification. Depending on the verification method used, a specification may either be another process (describing the desired behavior) or a formula in a modal logic expressing a relevant property.

Example Session

```
eddie% cwb
Edinburgh Concurrency Workbench, version 7.0,
Fri Oct  6 11:36:58 BST 1995
Command: agent Cell = a.'b.Cell;
Command: agent C0 = Cell[c/b];
Command: agent C1 = Cell[c/a,d/b];
Command: agent C2 = Cell[d/a];
Command: agent Buff3 = (C0 | C1 | C2)\{c,d};
Command: agent Spec = a.Spec';
Command: agent Spec' = 'b.Spec + a.Spec'';
Command: agent Spec'' = 'b.Spec' + a.'b.Spec'';
Command: save "spec1";
Command: eq (Buff3, Spec);
true
Command: quit
eddie%
```


Environments

- CWB has several separate environments.
- All bindings are dynamic:

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

- Environments are: agents, action sets, and propositions.
- Identifiers do not clash between environments.

```
set Cell = {c, d};
agent Buff3 = (C0 | C1 | C2)\Cell;
print;
** Agents **
...
agent Cell = a.'b.Cell
** Action Sets **
set Cell = {c, d}
```

Another Example Session

```
eddie% cwb
Edinburgh Concurrency Workbench, version 7.0,
Fri Oct  6 11:36:58 BST 1995
Command: input "junk";
Command: sort Buff3;
{a,'b}
Command: size Buff3;
Buff3 has 12 states.
Command: min (Buff3Min, Buff3);
Resetting tables...
Buff3Min has 4 states.
Command: vs (3, Buff3Min);
=== a a a ===>
=== a a 'b ===>
=== a 'b a ===>
Command: random (16, Buff3Min);
a,a,a,'b,a,'b,a,'b,'b,'b,a,a,a,'b,'b,'b
```


Formatting μ-calculus formulae. Modal Operators

If P is a proposition, $a_1, ...a_n$ are actions, and L is a set identifier, then the following are propositions:

- $[a_1, ...a_n]$P and [L]P - strong necessity

Agent A satisfies $[K]P$ if every K-derivative of A satisfies P; that is, there is an $a \in K$ such that $A \xrightarrow{a} A\prime$ and $A\prime$ satisfies P.

- $[[a_1, ...a_n]]$P and [[L]]P - weak necessity

Agent A satisfies $[K]P$ if every K-observation derivative of A satisfies P; that is, there is an $a \in K$ such that $A \overset{a}{\Rightarrow} A\prime$ and $A\prime$ satisfies P.

- $< a_1, ...a_n >$P and <L>P - strong necessity

Agent A satisfies $[K]P$ if it has a K-derivative of A satisfies P; that is, there is an $a \in K$ such that $A \xrightarrow{a} A\prime$ and $A\prime$ satisfies P.

- $<< a_1, ...a_n >>$P and <<L>>P - weak necessity

Formatting μ-calculus formulae (Cont'd)

- `-` indicates any transition (e.g. `[-]`). In μ-calculus, use [.].
- For strong modalities, the action sets must not include the empty action `eps`. For weak modalities, they must not include the unobservable action `tau`.

- *Propositional Connectives*: if P and Q are propositions, then so are `T`(true), `F`(false), `P` (negation), `P&Q` (conjunction), `P|Q` (disjunction) and `P=>Q` (implication).

- *Fixed Point Operators*: greatest fixpoint $\nu X.P$ is `max(X.P)`; least fixpoint $\mu X.P$ is `min(X.P)`.

To check a property, use command

```
checkprop (A, P);
```


Macros for Conversion between μ-calculus and CTL

`/local/share/cwb/examples/ccs/tl.macro`:

```
*  "Bx" is the "Box" ("always") operator. AG.
prop Bx(P) =  max(Z.P & [-]Z);
*  "Poss" is the "possibility" operator. EF.
prop Poss(P) = min(Z.P | <->Z);
* "Ev" is the "eventuality" operator.  AF and
* future states exist.
prop Ev(P) = min(Z.P | ([-]Z & <->T));
prop StrongUntil(P,Q) =
          min(Z.Q | (P & [-]Z & <->T));
prop WeakUntil(P,Q) = max(Z. Q | (P & [-]Z));
```

Some Useful CWB Commands

- **help**, **quit**
- **agent**, **set**, **relabel**, **prop**, **print**, **clear**
- **input "file"**, **output "file"** - send CWB output to a file rather than terminal, **save "file"**
- **sim** - simulate behavior of an agent using interactive simulation (see manual)
- **checkprop**
- **transitions** - list single-step transitions of an agent, **min**, **init** - observable actions that agent can perform immediately, **vs** - visible sequences of length n, **random**, **sort**, **size**, **states** - list the state-space of finite-state agent, **deadlocks** - find deadlocks and list traces leading to them
- **eq** - two agents are observationally equivalent, **pre** two agents are related by the weak divergence (bisimulation) preorder.

For More Information...

- See `man` pages for CWB
- See CWB Manual (Version 7)
- See examples in `/local/share/cwb/examples/ccs`
- See R. Milner, **Communication and Concurrency**, Prentice Hall International, 1989.
- See D. Kozen, "*Results on the Propositional μ-Calculus*", Theoretical Computer Science 27, p. 333-354, 1983.

What Other (Untimed) Process Algebras Are Out There?

- CCS (Calculus of Communicating Systems) - Milner
- CSP (Communicating Sequential Processes) - C.A.R. Hoare, **Communicating Sequential Processes**, Prentice Hall, 1985.
- ASP (Algebra of Communicating Processes) - J. Begstra and J. Klop. "*Algebra of Communicating Processes with Abstraction*". Journal of Theoretical Computer Science, 37:77-121, 1985.
- SCCS (Synchronous CCS) - used in CWB. Reference?

What Other (Timed) Process Algebras Are Out There?

- CSR (Communicating Shared Resources) - R. Gerber, Ph.D. Thesis, University of Pennsylvania, 1991.
- ACSR (Asynchronous CSR) - P. Bremond-Gregoire, J.Y. Choi and I. Lee, "*The Soundness and Completeness of ACSR*", Technical Report MS-CIS-93-59, Univ. of Pennsylvania, June 1993.
- Timed CSP - G. Reed and A. Roscoe. "*Metric Spaces as Models for Real-Time Concurrency*", in **Proceedings of Mathematical Foundations of Computer Science, LNCS**, volume 298, Springer-Verlag, 1987.
- and many others.