The Concurrency Workbench of North Carolina

Version 1.0

September, 1996

User's Manual

Rance Cleaveland
rance@csc.ncsu.edu

Steve Sims
stsims@eos.ncsu.edu

Dept. of Computer Science
N.C. State University
Raleigh, N.C. 27695-8206

Contents

Chapter 1

Introduction

The difficulty of building concurrent software systems has sparked a large research effort to develop formal approaches to the design and analysis of these systems. To reason formally about real-world systems, tool support is necessary; consequently, a number of tools embodying various analyses have been developed. Some of these tools include capabilities for performing *automatic verification*. In general these tools offer decision procedures that answer the verification question: does system *sys* satisfy specification *spec*? The tools differ in the formal notation used to describe systems and give specifications and in the decision procedures used to determine whether or not a system meets its specification. This document describes how to use a particular automatic verification tool: The Concurrency Workbench of North Carolina (CWB-NC) [9, 11].

The CWB-NC provides support for automatically answering the verification question: does system *sys* satisfy specification *spec*? To implement such a tool, the verification question must be formulated more carefully by fixing the following:

- a precise notation for defining systems
- a precise notation for defining specifications
- what it means for a system to satisfy a specification

Like its predecessor, the Edinburgh Concurrency Workbench [8, 14], the CWB-NC supports several formulations of the verification question.

One formulation supported by the CWB-NC and its predecessor involves using a process algebra such as CCS [13] for defining systems and specifications and using a behavioral equivalence to relate systems and specifications. A number of useful behavioral equivalences have been proposed. A behavioral equivalence is an equivalence relation which relates a system and a specification only when the system *behaves the same* as the specification. Different notions of *behaving the same* yield different behavioral equivalences. A similar approach involves using a behavioral preorder to relate a system and a specification when the system *behaves the same or better than* the specification. Efficient algorithms have been developed for equivalence and preorder checking [15, 2] and routines for performing these types of verification have been implemented in the CWB-NC.

A second formulation of the verification question involves using a design language (possibly a process algebra) for describing systems and temporal logic formulas as specifications. In this framework, referred to as model checking , one writes a logic formula encoding a desired property of the system and the tool determines whether a system satisfies the formula. The CWB-NC includes a model checker for determining whether systems satisfy formulas written in an expressive temporal logic, the modal mu-calculus.

In contrast with the Edinburgh CWB, and with other tools, the CWB-NC also supports the following.

- **Diagnostic information.** The CWB-NC provides appropriate diagnostic information for explaining why two systems fail to be related by a given semantic equivalence or preorder.

- **Language flexibility.** The design of the system exploits the language-independence of its analysis routines by localizing language-specific procedures (syntactic analyzers, semantic functions) in one module. This enables users to change the system description language of the CWB-NC using the Process Algebra Compiler of North Carolina (PAC-NC) [10, 7].

In order to enable the tool to handle large "real-world" systems we have also paid great attention to issues of time- and space-efficiency.

Chapter 2

Preliminaries

2.1 Obtaining the CWB-NC

The tool is available free of charge and may be obtained from the following URL:

`http://www4.ncsu.edu/ rance/WWW/cwb-nc.html`.

This web page also includes the directions that follow.

2.2 Installing the CWB-NC

2.2.1 Installing Binaries

Pre-compiled binaries for the system on the following architecture- operating system combinations are now available:

Architecture	**Operating System**
RS6000	Aix 3.x
Sparc	SunOS 4.x
Sparc	Solaris 2.x

Obviously, installing the binary distribution is much easier than compiling the sources (and does not require that you have SML-NJ Version 109 installed on your system), so if one of the above combinations meets your needs this is probably the option you should choose.

To install a binary distribution do the following:

1. Create a directory on your system in which the CWB-NC will reside.

   ```
   % mkdir cwb-nc
   ```

2. You will need to obtain the CWB-NC installation script and at least two compressed tar files. The first tar file contains the CWB-NC standard distribution containing

examples, documentation, and other common files. The other compressed tar file(s) you must obtain are pre-compiled binaries for each architecture-operating system combination you wish to install. You may pick up the files you need from the web page which includes the following:

- CWB-NC installation script:
  - install-cwb-nc (17Kbyte)
- Standard distribution:
  - cwb-nc.tar.Z (233Kbyte)
- Pre-compiled binaries:
  - cwb-nc-rs6000-aix.tar.Z (2.5 Mbyte)
  - cwb-nc-sparc-sunos.tar.Z (2.3 Mbyte)
  - cwb-nc-sparc-solaris.tar.Z (2.3 Mbyte)

Place these files in the newly created CWB-NC directory on your system. If you have multiple platforms sharing the same file system, you only need to obtain one copy of the standard distribution, along with the pre-compiled binaries for each architecture-operating system combination you wish to install. Place all files in the same cwb-nc home directory. When the files are extracted from the tar files, they will be placed in the proper subdirectories. The CWB-NC is configured to automatically determine at run time the architecture and operating system on which the CWB-NC is being run and load the appropriate binaries.

3. If you wish the use the CWB-NC graphical user interface, then expectk (version 5.20) must be available on your system. This program combines tcl-tk with expect and allows a graphical interface to communicate with a text-based user interface. Expectk is free and available at `ftp.cme.nist.gov` as `pub/expect/beta.tar.Z`. Expectk 5.20 requires Tcl 7.5 and Tk 4.1 which are also free and available at ftp.sunlabs.com as `pub/tcl/tcl7.5p1.tar.Z` and `pub/tcl/tk4.1p1.tar.Z`. The expect home page may be found at `URL http://www.cme.nist.gov/pub/expect/`. and the Tcl home page may be found at `URL http://www.smli.com/research/tcl/`. If you wish to use the CWB-NC graphical user interface, make sure expectk is installed on your system and be prepared when running the CWB-NC installation script to give the absolute path to the directory in which expectk resides. If you wish to use only the text-based user interface, then expectk is not necessary. If you wish to add support for the GUI at a later time, simply install expectk, then re-run the CWB-NC installation script.

4. Run the installation script.

```
% install-cwb-nc
```

You will be queried for a few pieces of information and then the distribution will be unpacked and installed. If you wish to support different architecture-operating system combinations, simply re-run the installation script on the different platforms you wish to support.

5. The system is now ready to go! The script for running the system is .../cwb-nc/bin/cwb-nc. See Chapter 3 for details on how to invoke and use the system. Have fun.

2.2.2 Compiling Sources

If pre-compiled binaries are not available for your platform, then you may compile the system from the source distribution.

Compiling the CWB-NC requires SML of New Jersey Version 109. This is available free of charge from :

`URL ftp://ftp.research.bell-labs.com/dist/smlnj/working/`

This is not a general release, but rather an intermediate working release; however, our experience has been that it is fairly stable. In addition to the compiler you will also need ml-lex, ml-yacc, and the Compiliation Manager all of which are included with the distribution located at the above URL. SML-NJ 109 is available on the following platforms:

Architecture	**Operating System**
Alpha	OSF/1 V 2.0 or later
HPPA	HPUX 9.x
MIPS	Irix 4.x
MIPS	Irix 5.x
Sparc	SunOS 4.x
Sparc	Solaris 2.x
RS6000	Aix
x86	Linux
x86	Solaris 2.x

The CWB-NC graphical user interface requires Expectk Version 5.20. See description under installing pre-compiled binaries for details.

To install from the source distribution do the following:

1. Create a directory on your system in which the CWB-NC will reside.

   ```
   % mkdir cwb-nc
   ```

2. You will need to obtain the CWB-NC installation script and two compressed tar files. The first tar file contains the CWB-NC standard distribution containing examples, documentation, and other common files. The other tar file contains the CWB-NC sources. You may pick up the files you need from the web page which includes the following:

- CWB-NC installation script:
  - install-cwb-nc (17Kbyte)
- Standard distribution:
  - cwb-nc.tar.Z (233Kbyte)
- Source distribution:
  - cwb-nc-src.tar.Z (344Kbyte)

Place these files in the newly created CWB-NC directory on your system.

3. Run the installation script.

```
% install-cwb-nc
```

If the script does not run, the most likely problem is that that the Bourne Shell is not located in /bin/sh on your system. If this is the problem, locate the Borne Shell on your system and change the first line of install-cwb-nc. You will be queried for a few pieces of information and then the distribution will be unpacked, compiled, and installed. If you wish to support different architecture-operating system combinations, simply re-run the installation script on the different platforms you wish to support (of course SML-NJ 109 must be installed on each platform).

4. The system is now ready to go! The script for running the system is .../cwb-nc/bin/cwb-nc. For details on how to invoke and use the system, see Chapter 3 of the User's manual located in the newly created doc directory. Have fun.

2.3 Feedback and Bug Reports

Please send comments and bug reports to `cwb-nc@csc.ncsu.edu`. All feedback is greatly appreciated.

Chapter 3

Using the CWB-NC

This chapter explains via a number of simple examples how to use the CWB-NC. The general procedure for using the tool is for the the user to first write a *program* in the desired design language (ccs in the following examples) describing the behavior of a concurrent system. This program is stored in a file as a sequence of declarations and then loaded into the CWB-NC. Once the file is loaded the user may then invoke various analysis routines to better understand the behavior of the concurrent system at hand.

3.1 Invoking the CWB-NC

The tool is invoked with the command:

`cwb-nc` *design-language* `[-gui]`

which takes one required and one optional argument. The required argument indicates the design language to be used for describing systems. Currently available languages are:

Design language	*Command line argument*
Milner's Calculus of Communicating Systems (CCS) [13]	`ccs`
Version of CCS with prioritized actions [5]	`pccs`
Basic Lotos [1]	`lotos`

Versions for CSP and Timed CCS will be released in the near future. The examples in this document use CCS as the design language. The tool is used in exactly the same fashion for other languages. If the `-gui` flag is given then a graphical user interface will be invoked, otherwise a text-based interface is initiated. Both of the user interfaces consist of a read-eval-print loop in which the CWB-NC repeatedly accepts a user request, executes it, and displays a result.

3.2 The CWB-NC Text-Based User Interface

The CWB-NC text-based user interface offers a set of unix-like commands for invoking the various analysis routines offered by the tool. Personal taste might lead some users to prefer the text interface over the gui; consequently, we have decided to support both versions of the tool. It may also be the case that it is easier to get the text based version working on a wider range of platforms. This section introduces some of the CWB-NC commands and Chapter 5 includes man pages for all CWB-NC commands.

3.2.1 Command Line Conventions

The CWB-NC text interface is started as follows:

```
% cwb-nc ccs

The Concurrency Workbench of North Carolina
(Version 1.0 --- September, 1996)

cwb-nc>
```

When the prompt (`cwb-nc>`) appears, the user may enter a CWB-NC command. The CWB-NC adopts a C-shell-like philosophy toward commands. That is, commands are interpreted as space-separated sequences of *words*. Some commands may also be given flags. As an example, the following command checks for may-equivalence between agents A1 and A2.

```
cwb-nc> eq -S may A1 A2
```

The `-S` flag indicates that the *semantics* to be used is *may* semantics.

Just as in the C-shell, if an argument to a command contains an embedded space then the argument needs to be surrounded with quotation marks. For example,

```
cwb-nc> eq "a.nil + b.nil" A2
```

compares agent `a.nil + b.nil` with agent `A2`.

The \ may be used at the end of a line as a continuation character to allow commands to extend over one line. For example

```
cwb-nc> eq A1 \
           A2
```

compares `A1` and `A2` for equivalence. When no S flag is given, observational equivalence is computed.

3.2.2 A Simple Example

We now demonstrate some of the primary CWB-NC commands by using the tool to examine a simple version of the Alternating Bit Protocol which is shown in Figure 3.1 and is included in the examples of the CWB-NC distribution. The file includes two versions of the protocol; the first employs a safe medium which never looses any messages and the second uses a lossy medium which may arbitrarily drop messages. The sequence of CWB-NC commands and responses described in the following appears in Figure 3.2.

After invoking the ccs-text version of the CWB-NC, the `load` command is used to load the system description contained in the file `abp.ccs`. The `.ccs` suffix is required of system designs written in CCS. The `ls` command may be used to list the agents currently loaded in the system and available for analysis. This list now includes the two versions of the protocol `ABP-safe` and `ABP-lossy`, their specification `Spec`, as well as the various sub-components of the two systems.

We now compute whether `ABP-safe` and `Spec` are observationally equivalent. The CWB-NC equivalence checking routines are invoked with the `eq` command and the semantics flag `-S` *semantics-type* indicates the particular equivalence relation to be computed. The `obseq` argument given here indicates observational equivalence. When this `eq` command is issued, the result is displayed instantly; however, for larger systems which take longer to analyze, the messages such as `Building automaton...` and `Done building automaton` report the progress of the computation to the user. The numbers of states and transitions in the automaton constructed to model to behavior of `Spec` and `ABP-lossy` are also reported. [1] The `FALSE` indicates that the two agents are not observationally equivalent and the ensuing *distinguishing formula* (`[[send]]<<'receive>>tt`) explains *why* they are inequivalent. See Section 3.4 for an explanation of distinguishing formulas. Note that the next `eq` command issued has no `-S` flag; therefore, the default case applies and the CWB-NC checks whether `ABP-lossy` is observationally equivalent to `Spec` and we see that the two agents are equivalent.

The next two commands show how to use the CWB-NC model checker to determine if a system has a property specified as a mu-calculus formula. The file abp.mu contains a mu-calculus formula, `can_deadlock`, which holds of a system if it is capable of reaching a state where no further actions are possible. We see that the property holds of ABP-safe, thus explaining why it is not equivalent to `spec`.

3.3 The CWB-NC Graphical User Interface

Figure 3.3 contains a diagram of the CWB-NC graphical user interface and labels its various parts. To demonstrate how to use the gui, we describe the sequence of interactions which lead to the gui state shown in the diagram. In the following we write click(n) to mean

[1] The first step in performing the various types of analysis is to construct an automaton which models the behavior of the systems at hand. This means the analysis routines do not depend on the particular design language used to describe systems and makes it possible to re-target the CWB-NC to different languages.

```
*********************************************************************
*  abp.ccs   (written by Rance Cleaveland)
*
*  This file contains a version of the Alternating Bit Protocol.
*
*  The specification of the protocol is represented by the agent
*  "Spec", and there are two implementations, "ABP-lossy", which
*  uses a lossy medium "Mlossy", and "ABP-safe, which uses a
*  reliable medium "Msafe".
*
*  "ABP-lossy" is correct in the sense that it is observationally
*  equivalent to "Spec", whereas "ABP-safe" is not.
*
*********************************************************************

proc Spec = send.'receive.Spec

***  The definition of the sender

proc S0  = send.S0'
proc S0' = 's0.(rack0.S1 + rack1.S0' + t.S0')
proc S1  = send.S1'
proc S1' = 's1.(rack1.S0 + rack0.S1' + t.S1')

***  The definition of the receiver

proc R0 = r0.'receive.'sack0.R1 + r1.'sack1.R0 + t.'sack1.R0
proc R1 = r1.'receive.'sack1.R0 + r0.'sack0.R1 + t.'sack0.R1

***  The definition of the reliable medium "Msafe"

proc Msafe = s0.'r0.Msafe + s1.'r1.Msafe +
             sack0.'rack0.Msafe + sack1.'rack1.Msafe

***  The definition of the lossy medium "Mlossy"

proc Mlossy =
    s0.('r0.Mlossy + Mlossy) + s1.('r1.Mlossy + Mlossy)
       + sack0.('rack0.Mlossy + Mlossy) + sack1.('rack1.Mlossy + Mlossy)

***  The definition of the safe implementation "ABP-safe"

proc ABP-safe = (R0 | Msafe | S0)\Internals

***  The definition of the lossy implementation "ABP-lossy"

proc ABP-lossy = (R0 | Mlossy | S0)\Internals

set Internals = {r0,r1,s0,s1,rack0,rack1,sack0,sack1}
```

Figure 3.1: The file abp.ccs includes a CCS description of two versions of the Alternating Bit Protocol.

```
% cwb-nc ccs
The Concurrency Workbench of North Carolina
(Version 1.0 --- September, 1996)

cwb-nc> load abp.ccs
Execution time (user,system,gc,real):(0.060,0.040,0.000,0.162)
cwb-nc> ls
===Agent===

Spec
S0
S0'
S1
S1'
R0
R1
Msafe
Mlossy
ABP-safe
ABP-lossy

===Set===

Internals

===Formula===

Execution time (user,system,gc,real):(0.050,0.010,0.000,0.075)
cwb-nc> eq -S obseq Spec ABP-safe
Building automaton...
States: 51
Transitions: 76
Done building automaton.
Transforming automaton...
Done transforming automaton.
FALSE...
Spec satisfies:
        [[send]]<<'receive>>tt
ABP-safe does not.
Execution time (user,system,gc,real):(0.240,0.040,0.000,0.311)
cwb-nc> eq Spec ABP-lossy
Building automaton...
States: 59
Transitions: 132
Done building automaton.
Transforming automaton...
Done transforming automaton.
TRUE
Execution time (user,system,gc,real):(0.290,0.030,0.000,0.406)
cwb-nc> load abp.mu
Execution time (user,system,gc,real):(0.030,0.000,0.000,0.160)
cwb-nc> chk ABP-safe can_deadlock
Invoking alternation-free model checker.
Building automaton...
States: 49
Transitions: 74
Done building automaton.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.150,0.000,0.000,0.161)
```

Figure 3.2: Sample CWB-NC session using Alternating Bit Protocol example

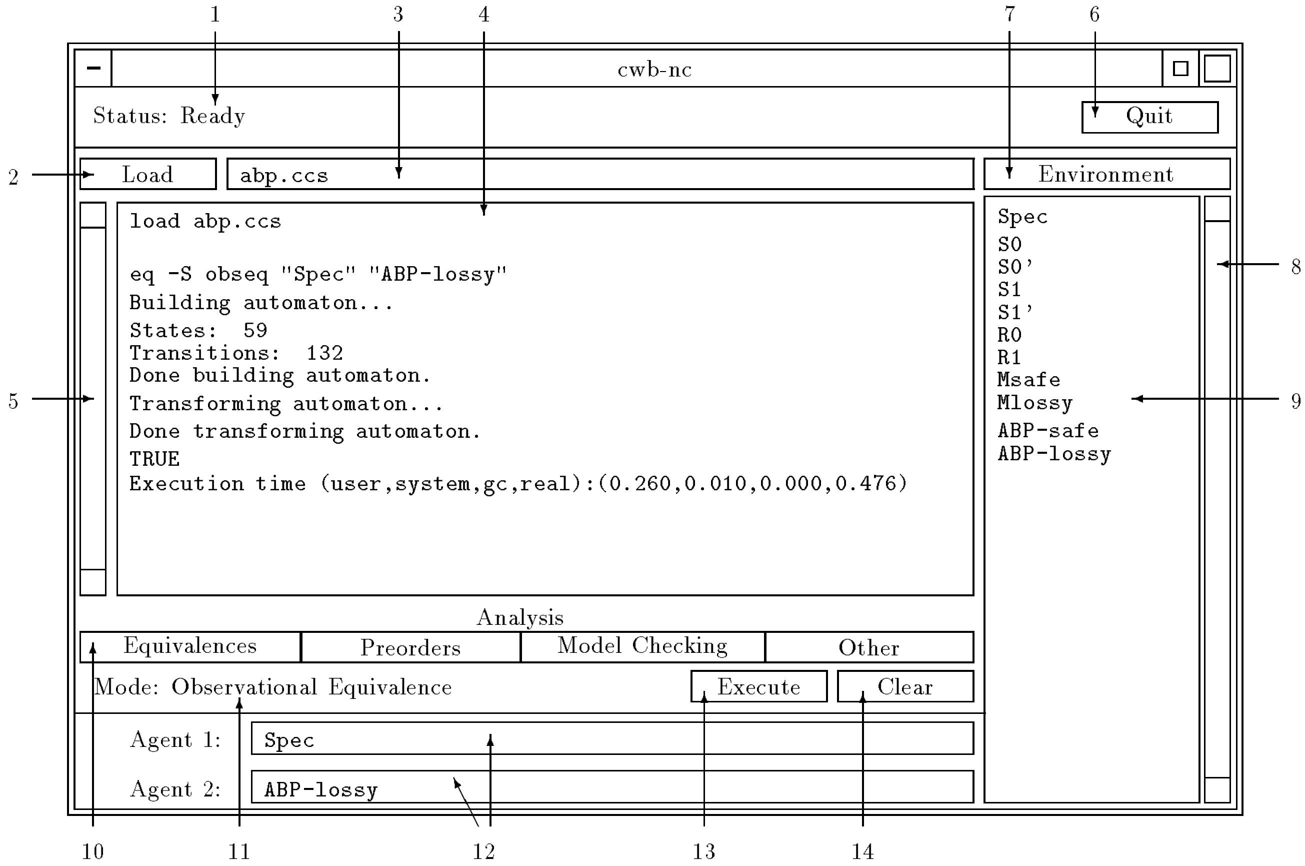
clicking with mouse button n.

- Invoke CWB-NC with `-gui` flag (`cwb-nc ccs -gui`) in the directory containing the Alternating Bit Protocol example (abp.ccs).
- The tool status message will display `Initializing system...`, then `Ready`.
- Click(1) in the load entry box bringing it in focus, then type: `abp.ccs` to load the Alternating Bit Protocol example. Hitting return in the load entry box or clicking(1) the load button causes the file to be loaded and a list of variables bound in the file to be displayed in the variable list. These items are now loaded in the cwb-nc and may be passed as arguments to the various analysis routines. The initial list displays *agent* variables. Select an item from the environment menu to change the type of variables displayed. For example, selecting the `Set` menu item displays a list of the variables bound to sets of actions in the `abp.ccs` file. Also notice that after initiating the load, the message `load abp.ccs` appears in the results window to indicate the action the cwb-nc is taking. This is the command that would be issued to the text interface.
- Click(1) on the first item in the variable list, `Spec`. This causes `Spec` to be displayed in reverse video indicating that it is selected. Click(2) in the top analysis entry box (now labeled Agent 1). This drops `Spec` in the entry box. Alternatively, `Spec` could have been simply typed in the entry box. Repeat the process to enter ABP-lossy in the lower entry box.
- Click(1) on the execute button. This causes the tool to check whether or not Spec and ABP-lossy are observationally equivalent and display the answer in the results window. The gui should now appear as it does in Figure 3.3.

To perform a different type of analysis, change the analysis mode by selecting an item from a menu on the analysis menu bar. For example, select `Strong` from the `Equivalences` menu indicating the desire to check whether two agents are related by the strong bisimulation equivalence relation [13]. When this menu item is selected the current analysis mode changes from observational equivalence to strong equivalence and the CWB-NC is ready to compute strong equivalence. Clicking(1) the execute button (with Spec and ABP-lossy still in the analysis entry boxes) begins the computation. The `FALSE` displayed in the results window indicates that the agents are not strongly equivalent and a *distinguishing formula* (`[t]ff`) satisfied by `Spec` but not by `ABP-lossy` is displayed. See Section 3.4 for explanation of how to interpret distinguishing formulas. Intuitively this formula indicates that `Spec` is unable to perform a `t` action (the CCS internal action τ), but ABP-lossy is capable of performing a `t` action.

3.4 Diagnostic Information

When two systems fail to be related by a given behavioral relation, the CWB-NC returns diagnostic information to explain *why* the systems are not related [4]. The information



1. Tool status message
2. Load button
3. Load entry box
4. Results window
5. Results window scrollbar
6. Quit button
7. Environment menu button
8. Variable list scrollbar
9. Variable list
10. Analysis menu bar
11. Current analysis mode
12. Analysis entry boxes
13. Execute button
14. Clear button

Figure 3.3: The GUI after loading a system description and running a check for observational equivalence.

consists of a property that one system has but the other does not. The way the property is formulated varies depending on the relation being computed.

3.4.1 Strong Bisimulation Equivalence

When two systems are not strongly bisimulation equivalent, the CWB-NC returns a Hennessy-Milner Logic (HML) formula [12] satisfied by one system but not the other. The syntax of HML formulas is defined by the following grammar, where a is an action.

$$\Phi := \mathtt{tt} \mid \mathtt{ff} \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \langle a \rangle \Phi \mid [a]\Phi$$

The formula tt holds of every state, while the formula ff holds of no state. The formula $\Phi_1 \wedge \Phi_2$ holds of a state if both Φ_1 and Φ_2 hold of the state, likewise $\Phi_1 \vee \Phi_2$ holds of a state if either Φ_1 or Φ_2 hold of the state. The modal formula $\langle a \rangle \Phi$ holds of a state if the state has *some* a-derivative at which Φ holds, and $[a]\Phi$ holds at a state if *all* a-derivatives of the state satisfy Φ.

The following example demonstrates the use of HML formulas as diagnostic information.

```
cwb-nc> eq -S bisim "a.b.nil + a.c.nil" "a.(b.nil + c.nil)"
Building automaton...
States: 9
Transitions: 9
Done building automaton.
FALSE...
a.b.nil + a.c.nil satisfies:
        <a>[b]ff
a.(b.nil + c.nil) does not.
Execution time (user,system,gc,real):(0.040,0.000,0.000,0.085)
```

The HML formula here indicates that the first agent has an a-transition to a state that has no b-transition; however, the second agent does not satisfy the formula, so after every a-transition it always has an enabled b-transition.

3.4.2 Weak Bisimulation Equivalence

To explain weak bisimulation inequivalence, a variation of HML is used that employs weak modal operators which abstract away from internal computation. The modified syntax is given by the following.

$$\Phi := \mathtt{tt} \mid \mathtt{ff} \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \langle\langle a \rangle\rangle \Phi \mid [[a]]\Phi$$

We say that q is a weak a-derivative of p if $\exists p' : p'' : p \xrightarrow{\tau *} p' \xrightarrow{a} p'' \xrightarrow{\tau *} q$. The modal formula $\langle\langle a \rangle\rangle \Phi$ holds of a state if the state has *some* weak a-derivative at which Φ holds, and $[[a]]\Phi$ holds at a state if *all* weak a-derivatives of the state satisfy Φ.

The following example demonstrates diagnostic information for observational inequivalence.

```
cwb-nc> eq -S obseq "a.t.(b.nil + c.nil)" "a.b.nil + a.c.nil"
Building automaton...
States: 7
Transitions: 8
Done building automaton.
Transforming automaton...
Done transforming automaton.
FALSE...
a.t.(b.nil + c.nil) satisfies:
        [[a]]<<b>>tt
a.b.nil + a.c.nil does not.
Execution time (user,system,gc,real):(0.030,0.030,0.010,0.105)
```

3.4.3 May Equivalence (Preorder)

When two systems are not may equivalent (or not related by the may preorder), the CWB-NC returns an action trace which one may perform, but the other may not. The returned traces are simply space delimited sequences of actions.

The following shows the use of traces as diagnostic information.

```
cwb-nc> eq -S may "a.b.nil + a.c.nil" "a.b.nil"
Building automaton...
States: 5
Transitions: 5
Done building automaton.
Transforming automaton...
Done transforming automaton.
FALSE...
a.b.nil + a.c.nil has trace:
        a c
a.b.nil does not.
Execution time (user,system,gc,real):(0.030,0.010,0.000,0.083)
```

3.4.4 Must Equivalence (Preorder)

When two systems are not must equivalent (or not related by the must preorder), the CWB-NC returns a test one must pass, but the other must not. The returned tests have the following form, where a, a_i are actions.

$$
\begin{array}{rcl}
\mathit{test} & := & \mathit{actSequence}\ (\ \mathit{acceptanceSet}\ ) \\
 & | & \mathit{actSequence}\ \mathtt{CONV} \\
 & & \\
\mathit{actSequence} & := & \\
 & | & a \rightarrow \mathit{actSequence} \\
 & & \\
\mathit{acceptanceSet} & := & a_1.\mathtt{PASS} + a_2.\mathtt{PASS} + \ldots + a_n.\mathtt{PASS}
\end{array}
$$

A system must pass the test *actSequence* (*acceptanceSet*), if it is always the case that after performing the sequence of actions in *actSequence*, the system is in a state where it is capable of performing some action in *acceptanceSet*. A system must pass the test *actSequence* `CONV`, if it is always the case that after performing the sequence of actions in *actSequence*, the system is in a convergent state. A convergent state is one in which the system is not capable of performing an infinite sequence of internal actions.

The use of tests as diagnostic information is demonstrated below.

```
cwb-nc> eq -S must "a.b.(c.nil + d.nil)" "a.(b.c.nil + b.d.nil)"
Building automaton...
States: 8
Transitions: 9
Done building automaton.
Transforming automaton...
Done transforming automaton.
FALSE...
a.b.(c.nil + d.nil) must pass:
        a->b->(d.PASS)
a.(b.c.nil + b.d.nil) may fail it.
Execution time (user,system,gc,real):(0.030,0.020,0.000,0.116)
```

Chapter 4

Mu-Calculus Syntax and Semantics

The CWB-NC includes facilities for determining whether or not a process satisfies properties expressed in an expressive temporal logic called the modal mu-calculus. To use the model checker, one first formulates properties that a process should have as mu-calculus formulas and includes the formulas in a file with the `.mu` suffix. One then loads this file into a CWB-NC session (with the `load` command) and invokes the model checker with the `chk` command.

Computation Tree Logic (CTL) [3] has proven to be an intuitive and easy to use logic for specifying temporal properties of systems, so we have extended the logic supported by the CWB-NC to include the CTL operators. This does not increase the expressive power of the logic; however, some users may prefer CTL over the mu-calculus.

This chapter explains how to formulate properties in the supported logic.

4.1 Syntax

The syntax of a `.mu` file is shown in Figure 4.1 and consists of a list of proposition declarations separated by white space. Each declaration binds a mu-calculus proposition to a proposition variable. Currently, true and false (**tt** and **ff**) are the only atomic propositions; although we plan to add others such as a convergence predicate that is true for states not capable of performing an infinite sequence of internal actions. Proposition variables may appear within propositions. Negation, disjunction, and conjunction are represented with standard notation: **not**, `\/`, and `/\`. The modal operators <*actSet*>, <<*actSet*>>, [*actSet*], and [[*actSet*]] are each parameterized by an action set. An action set may either be a single action, a comma separated list of actions preceded by a { and followed by a }, or a complemented set. Note that the - character is used for set complementation. Recursive formulas are given with the least (**min**) and greatest (**max**) fixpoint operators. Parentheses may be used in the standard fashion to group subterms of a proposition. The CTL operators use the standard notation as indicated in Figure 4.1.

tokens

`"[a-zA-Z][a-zA-Z0-9'_-]*"` => *Ident*

grammar

muFile	:	*propDeclList*	
propDecl	:	**prop** *ident* = *prop*	
prop	:	**tt**	(True)
	|	**ff**	(False)
	|	*Ident*	(Proposition variable)
	|	**not** *prop*	(Negation)
	|	*prop* `\/` *prop*	(Disjunction)
	|	*prop* `/\` *prop*	(Conjunciton)
	|	`<` *actSet* `>` *prop*	(Possibility)
	|	`<<` *actSet* `>>` *prop*	(Weak possibility)
	|	`[` *actSet* `]` *prop*	(Necessity)
	|	`[[` *actSet* `]]` *prop*	(Weak necessity)
	|	**min** *ident* = *prop*	(Least fix point)
	|	**max** *ident* = *prop*	(Greatest fix point)
	|	(*prop*)	(Parentheses)
	|	**AG** *prop*	(CTL operators)
	|	**AF** *prop*	
	|	**A** (*prop* **U** *prop*)	
	|	**A** (*prop* **W** *prop*)	
	|	**EG** *prop*	
	|	**EF** *prop*	
	|	**E** (*prop* **U** *prop*)	
	|	**E** (*prop* **W** *prop*)	
actSet	:	*act*	
	|	{ actList }	
	|	- actSet	

lists

List name	May be Empty?	Open Delimiter	Separator	Close Delimiter	Items
actList	yes	EMPTY_STR	,	EMPTY_STR	*act*
propDeclList	no	EMPTY_STR	EMPTY_STR	EMPTY_STR	*propDecl*

priorities

tt,**ff** $>$ `[`,`[[`,`]`,`]]`,`<`,`<<`,`>`,`>>` $>$ **not, AG, AF, EG, EF** $>$

`/\` $>$ `\/` $>$ =

Figure 4.1: Syntax of a `.mu` file

4.2 Semantics

We first explain the semantics of the pure mu-calculus and then describe how it is extended to include the CTL operators.

4.2.1 The Basic Mu-Calculus Operators

We now explain what it means for a process to satisfy a mu-calculus formula, which intuitively means that the formula is true for the process. The simplest formulas have obvious meanings; **tt** holds for all processes, whereas **ff** holds for no process. A proposition variable holds for a process if the proposition bound to the variable holds for the process. The formula **not** *prop* holds of a process if *prop* is not true for the process. A process satisfies $prop_1$ `\/` $prop_2$ if it satisfies either $prop_1$ or $prop_2$; similarly, $prop_1$ `/\` $prop_2$ holds for a process if both $prop_1$ and $prop_2$ hold for the process.

The modal operators $\texttt{<}\mathit{actSet}\texttt{>}$, $\texttt{<<}\mathit{actSet}\texttt{>>}$, $\texttt{[}\mathit{actSet}\texttt{]}$, and $\texttt{[[}\mathit{actSet}\texttt{]]}$ allow one to make statements about the initial actions a process is able to take. If we view a process as a particular state in an automaton modeling the global behavior of the process, then the modal operators allow us to make statements about the edges leaving the state and the states to which these edges lead. In the following, note that q is an a-derivative of p if $p \xrightarrow{a} q$, that is if process p can perform and action a and subsequently behave as the process q. Similarly, we say that q is a weak a-derivative of p if p is capable of performing zero or more internal computation steps, then an a action, then zero or more internal computation steps and subsequently behaving as q. If τ represents an internal computation step and $\tau*$ indicates zero or more internal computation steps, then q is a weak a-derivative of p if $\exists p' : p'' : p \xrightarrow{\tau*} p' \xrightarrow{a} p'' \xrightarrow{\tau*} q$.

The *possibility* operator allows one to express that some a-derivative of a process should satisfy a property. The formula $\texttt{<}a\texttt{>}\mathit{prop}$ holds for a process if the process has an a-derivative that satisfies *prop*. Here the argument to the modal operator is implicitly a singleton set containing the action a. The operator extends to arbitrary sets as follows. A process p satisfies $\texttt{<}\mathit{actSet}\texttt{>}\mathit{prop}$ if *actSet* contains an action b such that p has a b-derivative that satisfies *prop*. The *weak possibility* operator allows one to abstract away from internal computation; $\texttt{<<}a\texttt{>>}\mathit{prop}$ holds for a process if the process has a *weak* a-derivative that satisfies *prop*.

The *necessity* operator enables one to stipulate that all a-derivatives of a process satisfy a property. The formula $\texttt{[}a\texttt{]}\mathit{prop}$ holds for a process if all a-derivatives of the process satisfy *prop*. This operator also extends to arbitrary sets. A process p satisfies $\texttt{[}\mathit{actSet}\texttt{]}\mathit{prop}$ if $\texttt{[}b\texttt{]}\mathit{prop}$ holds of p for every action b in *actSet*. The *weak necessity* operator allows one to abstract away from internal computation by expressing that all *weak* a-derivatives of a process satisfies a property.

Finally, the least and greatest fixpoint operators (**min** and **max**) are used to define recursive formulas. Their meanings are defined in terms of infinite disjunctions and conjunctions as follows. Let $prop_0$ be the proposition **ff** and define $prop_{i+1}$ to be the proposition $prop[prop_i/X]$. Then $\mathbf{min}\ X = prop$ corresponds to the infinite disjunction $\bigvee_{i=0}^{\infty} prop_i$.

Dually, if $prop_0$ is defined as **tt**, then **max** $X = prop$ may be interpreted as the infinite conjunction $\bigwedge_{i=0}^{\infty} prop_i$

4.2.2 The CTL Operators

The meaning of each CTL operator is defined in terms of the *computations* of a process, which are the maximal execution sequences in which the process is capable of engaging. The elements of a computation sequence are processes which may be viewed as the intermediate states a process passes through during execution. In the following *Act* is a set of actions. A computation of a process p is a sequence of processes $p_0, p_1, ..., p_n$ such that:

- $p = p_0$
- $n = \infty$ or ($n \geq 0$ and $\forall a \in Act : p_n \stackrel{a}{\not\longrightarrow}$)
- $\forall i : 0 \leq i < n : \exists a \in Act : p_i \stackrel{a}{\longrightarrow} p_{i+1}$

Note that, unlike the traditional models for CTL, computations may be finite.

Each CTL operator has two components, the first of which is a quantifier indicating which computations of a process the property being formulated should hold for. If the first component of a CTL formula is **A**, then the property should hold for all computations of a process; whereas, if the first component is **E**, then there should exist a computation of the process for which the property holds.

The second component of a CTL operator, formulates a property of a computation. Let $C = p_0, p_1, ..., p_n$ be a computation, then:

- *Always.*
  G *prop* holds of C if *prop* holds for all $p_i, 0 \leq i \leq n$.
- *Eventually.*
  F *prop* holds of C if $\exists i, 0 \leq i \leq n$ and p_i satisfies *prop*.
- *Until.*
  $prop_1$ **U** $prop_2$ holds of C if $\exists i : 0 \leq i \leq n : p_i$ satisfies $prop_2$ and $\forall j : 0 \leq j < i :$ p_j satisfies $prop_1$.
- *Weak Until.*
  $prop_1$ **W** $prop_2$ holds of C if ($\exists i : 0 \leq i \leq n : p_i$ satisfies $prop_2$ and $\forall j : 0 \leq j < i :$ p_j satisfies $prop_1$)
  or
  ($\forall i : 0 \leq i \leq n : p_i$ satisfies $prop_1$).

The two components of a CTL formula combine to allow one to express properties of the computations of a process. For example, **AG** *prop* holds of a process if for all computations of the process G *prop* holds.

CTL Formula	*Mu-Calculus Translation*
AG *prop*	**max** X = *prop* `/\ [-{}]`X
AF *prop*	**min** X = *prop* `\/` (`[-{}]`X `/\ <-{}>`**tt**)
A ($prop_1$ **U** $prop_2$)	**min** X = $prop_2$ `\/` ($prop_1$ `/\ [-{}]`X `/\ <-{}>`**tt**)
A ($prop_1$ **W** $prop_2$)	**max** X = $prop_2$ `\/` ($prop_1$ `/\ [-{}]`X)
EG *prop*	**max** X = *prop* `/\` (`<-{}>`X `\/ [-{}]`**ff**)
EF *prop*	**min** X = *prop* `\/ <-{}>`X
E ($prop_1$ **U** $prop_2$)	**min** X = $prop_2$ `\/` ($prop_1$ `/\ <-{}>`X)
E ($prop_1$ **W** $prop_2$)	**max** X = $prop_2$ `\/` ($prop_1$ `/\` (`<-{}>`X `\/ [-{}]`**ff**))

Table 4.1: Mu-calculus translation of the CTL operators

We mentioned above that the CTL operators do not add expressive power to the mu-calculus; therefore, each CTL formula may be translated to an equivalent mu-calculus formula. This is in fact how the CWB-NC model checker is implemented, CTL formulas are translated into mu-calculus formulas and then fed into a mu-calculus model checker. The translation used for each operator is given in Table 4.1. Note in particular the treatment of finite (terminating) behaviors. Also recall that the set indicated in the modal operators (`-{}`) is the complement of the empty set, which of course is the set of all actions *Act*.

4.3 Examples

Figure 4.2 shows several mu-calculus propositions which express properties that the Alternating Bit Protocol model should (or should not) have.

- *can_deadlock* is true of a system if it may reach a deadlocked state.
- *can_send* is true of a system if, after a finite number (possibly 0) of internal execution steps, it may perform the `send` action.
- *can_receive* is true of a system if, after a finite number (possibly 0) of internal execution steps, it may perform the `'receive` action.
- *can_send_or_receive* is true of a system if it is always the case that either *can_send* or *can_receive* are true.
- *no_repeat, no_repeat'* are true of a system if it may never perform two consecutive `send` actions or two consecutive `'receive` actions.

```
prop can_deadlock =
  min X = [-{}]ff \/ <-{}>X

prop can_send =
  min Y = <send>tt \/ <t>Y

prop can_receive =
  min Y = <'receive>tt \/ <t>Y

prop always_can_send_or_receive =
  AG (can_send \/ can_receive)

prop no_repeat =
  AG ([send](not can_send) /\ ['receive](not can_receive))

prop no_repeat' =
  AG ([send] A([send]ff W <'receive>tt) /\
      ['receive] A(['receive]ff W <send>tt))
```

Figure 4.2: Example mu-calculus formulas

Chapter 5

CWB-NC Commands

This chapter includes man pages describing CWB-NC commands which are partitioned into two groups: top-level commands and simulator commands. Top-level commands are issued at the main prompt; whereas, simulator commands are entered when the tool is in simulator mode. See Section 3.2.1 for a description of command line conventions.

5.1 Top-Level Commands

NAME

cat

SYNOPSIS

cat identifier

DESCRIPTION

cat displays the item bound to identifier. If identifier is bound in more than one environment, each binding is displayed.

SEE ALSO

ls

NAME

cd

SYNOPSIS

cd directory

DESCRIPTION

cd sets the present working directory (the unix directory in which the CWB-NC searches when the load command is executed). The initial working directory is the one from which the CWB-NC was invoked.

EXAMPLE

```
cwb-nc> cd /usr/local/src/cwb-nc/examples/ccs
Execution time (user,system,gc,real):(0.000,0.000,0.000,0.000)
cwb-nc>
```

SEE ALSO

load

NAME

chk

SYNOPSIS

chk agent formula

DESCRIPTION

chk invokes the model checker to determine whether or not agent satisfies formula. If the formula is alternation- free then a global model checker is invoked; otherwise, if the formula is in the L2 fragment of the mu-calculus, then an on-the-fly model checker is invoked. Formulas in neither of these fragments are not yet handled.

EXAMPLE

```
cwb-nc> load abp.ccs
Execution time (user,system,gc,real):(0.120,0.010,0.000,0.161)
cwb-nc> load abp.mu
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.014)
cwb-nc> chk ABP-safe can_deadlock
Building automaton...
States: 49
Transitions: 74
Done building automaton.
True, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.130,0.010,0.000,0.216)
```

SEE ALSO

load

NAME

compile

SYNOPSIS

compile agent1 [agent2 ...]

DESCRIPTION

compile builds an automaton for the agents given as arguments and displays a textual representation of the constructed automaton. In the example below, the start state is 0 and this state has an "a" transition to state 1 and a "c" transition to state 2.

EXAMPLE

```
cwb-nc> compile "a.b.nil | c.nil"
Building automaton...
States: 6
Transitions: 7
Done building automaton.
0:      a       {1}
        c       {2}

1:      b       {3}
        c       {4}

2:      a       {4}

3:      c       {5}

4:      b       {5}

5:

Start States:  [0]
Execution time (user,system,gc,real):(0.010,0.020,0.000,0.082)
```

NAME

eq

SYNOPSIS

eq [-S semantic-type] agent1 agent2

DESCRIPTION

eq computes whether agent1 and agent2 are related by the equivalence relation indicated by semantic-type (obseq by default). The currently available semantic-type arguments are obseq (observational equivalence), bisim (strong bisimulation equivalence), trace (trace equivalence), may (may equivalence), and must (must equivalence).

EXAMPLE

```
cwb-nc> load abp.ccs
Execution time (user,system,gc,real):(0.080,0.070,0.000,0.198)
cwb-nc> eq Spec ABP-lossy
Building automaton...
States: 59
Transitions: 132
Done building automaton.
Transforming automaton...
Done transforming automaton.
TRUE
Execution time (user,system,gc,real):(0.250,0.010,0.000,0.423)
cwb-nc> eq -S bisim Spec ABP-lossy
Building automaton...
States: 59
Transitions: 132
Done building automaton.
FALSE...
Spec satisfies:
        [t]ff
ABP-lossy does not.
Execution time (user,system,gc,real):(0.190,0.050,0.010,0.343)
```

NAME

es

SYNOPSIS

es filename1 [filename2]

DESCRIPTION

es executes the sequence of CWB-NC commands contained in filename1 and directs the output to filename2 (stdout if second argument is not given). Note that each command in filename1 (including the final command) must end with a newline character.

EXAMPLE

```
Let the file abp.cws contain:

  load abp.ccs
  eq -S obseq Spec ABP-safe
  load abp.mu
  chk ABP-safe can_deadlock

After,

  cwb-nc> es abp.cws abp.cws.output
  Executing CWB script file abp.cws, directing output
    to abp.cws.output.
  June 17, 1996   10:03
  ....................
  cwb-nc>

the file abp.cws.output contains:

  Executing CWB script file abp.cws, directing output
    to abp.cws.output.
  June 17, 1996   10:01
  Execution time (user,system,gc,real):(0.010,0.000,0.000,0.120)
  cwb-nc> load abp.ccs
  Execution time (user,system,gc,real):(0.110,0.030,0.000,0.179)
  cwb-nc> eq -S obseq Spec ABP-safe
  Building automaton...
  States: 51
  Transitions: 76
```

```
Done building automaton.
Transforming automaton...
Done transforming automaton.
FALSE...
Spec satisfies:
  [[send]]<<'receive>>tt
ABP-safe does not.
Execution time (user,system,gc,real):(0.240,0.010,0.010,0.259)
cwb-nc> load abp.mu
Execution time (user,system,gc,real):(0.020,0.000,0.000,0.018)
cwb-nc> chk ABP-safe can_deadlock
Building automaton...
States: 49
Transitions: 74
Done building automaton.
True, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.110,0.010,0.000,0.121)
cwb-nc>
```

NAME

help

SYNOPSIS

help [command name]

DESCRIPTION

help displays information about CWB-NC commands. If executed with no arguments, then a list of available commands is displayed. If given a command name as an argument, then information about the specific command is displayed.

NAME

le

SYNOPSIS

le [-S semantic-type] agent1 agent2

DESCRIPTION

le computes whether agent1 and agent2 are related by the behavioral preorder indicated by semantic-type. The currently available semantic-type arguments are may (may preorder) and must (must preorder).

NAME

load

SYNOPSIS

load filename

DESCRIPTION

This command loads a file into the current CWB-NC session. The file suffix indicates the type of file to be loaded: .ccs files contain a sequence of ccs process definitions. .mu files contain a sequence of mu-calculus formulas.

NAME

ls

SYNOPSIS

ls

DESCRIPTION

ls lists the names of variables bound in the various environments, i.e. agent variables, set variables, ...

NAME

min

SYNOPSIS

min [-S semantic-type] agent identifier

DESCRIPTION

min minimizes agent with respect to the equivalence relation indicated by semantic-type (obseq by default) and binds the resulting agent to identifier. The currently available semantic type arguments are obseq (observational equivalence) and bisim (strong bisimulation equivalence).

EXAMPLE

```
cwb-nc> load abp.ccs
Execution time (user,system,gc,real):(0.090,0.040,0.000,0.163)
cwb-nc> min -S bisim ABP-lossy ABP-lossy-min
Building automaton...
States: 57
Transitions: 130
Done building automaton.
Minimizing automaton...
Computing bisimulation...
Done computing bisimulation.
Done minimizing automaton.
Execution time (user,system,gc,real):(0.210,0.060,0.000,0.489)
cwb-nc> size ABP-lossy-min
Building automaton...
States: 15
Transitions: 32
Done building automaton.
        15      states
        32      transitions
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.013)
```

NAME

quit

SYNOPSIS

quit

DESCRIPTION

quit ends the current CWB-NC session.

NAME

sim

SYNOPSIS

sim agent

DESCRIPTION

sim allows user-directed or random simulation of agent. After invoking the simulator the following commands are available: help [command], current, transition-num, random [num], break [-a][-d][-l] [act list], back [num], history, trace, goto num, !!, quit, More information on each of these commands is available via the simulator's help command.

EXAMPLE

```
cwb-nc> sim "a.b.nil + c.nil"
a.b.nil + c.nil
1: -- c --> nil
2: -- a --> b.nil
sim: 2
b.nil
1: -- b --> nil
sim: 1
nil
The agent has no transitions
sim: history
1: a.b.nil + c.nil -- a -->
2: b.nil -- b -->
3: nil
sim: trace
 a b
sim: goto 2
Current state moved 1 step back.
sim: current
b.nil
1: -- b --> nil
```

NAME

size

SYNOPSIS

size agent

DESCRIPTION

size builds an automaton which models the behavior of agent and displays the number of states and transitions in the constructed automaton.

EXAMPLE

```
cwb-nc> size "a.b.nil | 'a.c.nil"
Building automaton...
States: 9
Transitions: 13
Done building automaton.
Execution time (user,system,gc,real):(0.010,0.010,0.000,0.052)
```

NAME

sort

SYNOPSIS

sort agent

DESCRIPTION

sort displays the set of visible actions that agent may engage in.

EXAMPLE

```
cwb-nc> sort "a.b.nil | 'a.c.nil"
{ a, b, c, 'a }
Execution time (user,system,gc,real):(0.000,0.020,0.000,0.016)
```

NAME

trans

SYNOPSIS

trans agent

DESCRIPTION

trans displays the single step transitions which agent may perform.

EXAMPLE

```
Command: trans "a.nil + b.nil"
-- b --> nil
-- a --> nil
Execution took 0.000 seconds.
```

5.2 Simulator Commands

NAME

!!

SYNOPSIS

!!

DESCRIPTION

Simulator command which repeats the previous command.

NAME

back

SYNOPSIS

back [num]

DESCRIPTION

Simulator command which erases the previous num execution steps.

EXAMPLE

```
cwb-nc> sim a.b.c.d.nil
a.b.c.d.nil
1: -- a --> b.c.d.nil
cwb-nc-sim> 1
b.c.d.nil
1: -- b --> c.d.nil
cwb-nc-sim> 1
c.d.nil
1: -- c --> d.nil
cwb-nc-sim> 1
d.nil
1: -- d --> nil
cwb-nc-sim> back 2
Current state moved 2 steps back.
cwb-nc-sim> current
b.c.d.nil
1: -- b --> c.d.nil
```

NAME

break

SYNOPSIS

break [-a][-d][-l] [act list]

DESCRIPTION

Simulator command used to add (-a), delete (-d), and list (-l) break points. An agent is interpreted as a break point if one or more of its enabled transitions is labeled by an action in the break point set. A message is displayed when a break point is reached and random execution is halted. If no flag is given then each item in the list of actions is added to the set of break points. The act list argument is a list of actions separated by spaces.

EXAMPLE

```
Command: sim "a.b.nil + a.c.nil"
a.b.nil + a.c.nil
1: -- a --> c.nil
2: -- a --> b.nil
cwb-nc-sim> quit
Execution time (user,system,gc,real):(0.010,0.010,0.000,30.509)
Command: sim "a.b.nil + a.c.nil"
a.b.nil + a.c.nil
1: -- a --> c.nil
2: -- a --> b.nil
cwb-nc-sim> break -a a b c
cwb-nc-sim> break -d a
cwb-nc-sim> break -l
 b c
cwb-nc-sim> random 3
Break point
b.nil
1: -- b --> nil
```

NAME

current

SYNOPSIS

current

DESCRIPTION

Simulator command that displays the current agent and lists its available transitions.

EXAMPLE

```
cwb-nc> sim a.b.c.nil
a.b.c.nil
1: -- a --> b.c.nil
cwb-nc-sim> 1
b.c.nil
1: -- b --> c.nil
cwb-nc-sim> 1
c.nil
1: -- c --> nil
cwb-nc-sim> current
c.nil
1: -- c --> nil
cwb-nc-sim> back 1
Current state moved 1 step back.
cwb-nc-sim> current
b.c.nil
1: -- b --> c.nil
```

NAME

goto

SYNOPSIS

goto num

DESCRIPTION

Simulator command which sets the current state to state num of the current session, where 1 ¡= num ¡= current state number. Note that state numbers are displayed by the history command for each state on the path from the start state to the current state.

EXAMPLE

```
cwb-nc> sim a.b.c.d.nil
a.b.c.d.nil
1: -- a --> b.c.d.nil
cwb-nc-sim> 1
b.c.d.nil
1: -- b --> c.d.nil
cwb-nc-sim> 1
c.d.nil
1: -- c --> d.nil
cwb-nc-sim> 1
d.nil
1: -- d --> nil
cwb-nc-sim> history
1: a.b.c.d.nil -- a -->
2: b.c.d.nil -- b -->
3: c.d.nil -- c -->
4: d.nil
cwb-nc-sim> goto 2
Current state moved 2 steps back.
cwb-nc-sim> current
b.c.d.nil
1: -- b --> c.d.nil
```

SEE ALSO

history

NAME

help

SYNOPSIS

help [command name]

DESCRIPTION

help displays information about CWB-NC commands. If executed with no arguments, then a list of available commands is displayed. If given a command name as an argument, then information about the specific command is displayed.

NAME

history

SYNOPSIS

history

DESCRIPTION

Simulator command that prints the history of the current simulator session. Note that backtracking with the back command removes transitions from the history; thus, the path displayed by the history command represents a path from the initial agent to the current agent. The states through which the path progresses are numbered starting with 1.

EXAMPLE

```
cwb-nc> sim "a.b.c.nil | d.e.nil"
a.b.c.nil | d.e.nil
1: -- d --> a.b.c.nil | e.nil
2: -- a --> b.c.nil | d.e.nil
cwb-nc-sim> 2
b.c.nil | d.e.nil
1: -- d --> b.c.nil | e.nil
2: -- b --> c.nil | d.e.nil
cwb-nc-sim> 1
b.c.nil | e.nil
1: -- e --> b.c.nil | nil
2: -- b --> c.nil | e.nil
cwb-nc-sim> 1
b.c.nil | nil
1: -- b --> c.nil | nil
cwb-nc-sim> history
1: a.b.c.nil | d.e.nil -- a -->
2: b.c.nil | d.e.nil -- d -->
3: b.c.nil | e.nil -- e -->
4: b.c.nil | nil
```

SEE ALSO

trace, goto

NAME

quit

SYNOPSIS

quit

DESCRIPTION

Simulator command that ends the current simulator session.

NAME

random

SYNOPSIS

random [num]

DESCRIPTION

Simulator command that simulates num random execution steps of an agent. The default value for num is 1.

EXAMPLE

```
cwb-nc>  sim "a.b.c.nil | d.e.nil"
a.b.c.nil | d.e.nil
1: -- d --> a.b.c.nil | e.nil
2: -- a --> b.c.nil | d.e.nil
cwb-nc-sim> random
b.c.nil | d.e.nil
1: -- d --> b.c.nil | e.nil
2: -- b --> c.nil | d.e.nil
cwb-nc-sim> random 3
c.nil | d.e.nil
1: -- d --> c.nil | e.nil
2: -- c --> nil | d.e.nil
nil | d.e.nil
1: -- d --> nil | e.nil
nil | e.nil
1: -- e --> nil | nil
cwb-nc-sim> history
1: a.b.c.nil | d.e.nil -- a -->
2: b.c.nil | d.e.nil -- b -->
3: c.nil | d.e.nil -- c -->
4: nil | d.e.nil -- d -->
5: nil | e.nil
```

NAME

trace

SYNOPSIS

trace

DESCRIPTION

Simulator command that prints the trace of the current simulator session. Note that backtracking with the back command removes transitions from the trace; thus, the sequence displayed by the trace command represents a path from the initial agent to the current agent.

EXAMPLE

```
cwb-nc> sim "a.b.c.nil | d.e.nil"
a.b.c.nil | d.e.nil
1: -- d --> a.b.c.nil | e.nil
2: -- a --> b.c.nil | d.e.nil
cwb-nc-sim> 2
b.c.nil | d.e.nil
1: -- d --> b.c.nil | e.nil
2: -- b --> c.nil | d.e.nil
cwb-nc-sim> 1
b.c.nil | e.nil
1: -- e --> b.c.nil | nil
2: -- b --> c.nil | e.nil
cwb-nc-sim> 1
b.c.nil | nil
1: -- b --> c.nil | nil
cwb-nc-sim> trace
 a d e
```

SEE ALSO

history

Appendix A

Design Language Syntax and Semantics

We mentioned previously that the CWB-NC currently supports several different design languages for specifying system behavior: CCS, a version of CCS in which actions may be assigned different priorities, and Basic Lotos. Versions for CSP and Timed CCS will be released soon. In this appendix we briefly describe the syntax and semantics employed by the CWB-NC for each of the supported languages. The goal of each overview is to explain to users already familiar with a particular design language the details of how to use that language with the CWB-NC.

For each language, we give a Yacc-like grammar to define the syntax and provide a set of Plotkin-style SOS rules [16] to define the semantics. In the syntax description, we also list the Lex regular expressions for any non-trivial tokens as well as the associativity and priority assigned to tokens. These are also the notations used by the PAC-NC [10, 7] to generate language interfaces for the CWB-NC. The grammars and SOS rule sets shown here are elided versions of the PAC-NC specifications for the corresponding languages. The complete versions are included with the CWB-NC distributions.

For each language, the given set of SOS rules defines a *transitions relation* which determines the set of single step transitions that a term in the design language is capable of performing. SOS rules have the following general form.

$$\frac{premises\ (side\ condition)}{conclusion}$$

The intuitive reading of the rule is that if one is able to establish the premises, which typically involve statements about the execution behavior of subprograms of the one mentioned in the conclusion, and the side condition holds, then one may infer the conclusion. As an example, the following describes the synchronizations allowed by the parallel composition operation in CCS.

$$\frac{p \xrightarrow{a} p' \quad q \xrightarrow{b} q' \quad (a, b\ inverses)}{p|q \xrightarrow{\tau} p'|q'}$$

The rule states that if p can engage in an action a and evolve to p' and q can engage in b and evolve to q', and a and b are inverses (i.e. constitute an input/output pair on the same communication channel), then $p|q$ can execute an internal action, τ, corresponding to the synchronized execution of a and b.

A.1 CCS

The standard reference for CCS is [13].

A.1.1 CCS Syntax

A sample CCS specification was seen in Figure 3.1. Readers familiar with CCS will recognize that the standard syntax is used for all operators. The comment character is `*`. All text between a `*` and the end of line character is ignored. A `.ccs` file consists of a sequence of declarations which define processes and identifier sets. The keyword `proc` precedes a process declaration.

$$\texttt{proc}\ \mathit{procName} = \mathit{ccsTerm}$$

Here the behavior of a system component is defined in CCS by *ccsTerm* and this expression is bound to the variable *procName*. Identifier set declarations are similar.

$$\texttt{set}\ \mathit{setName} = \mathit{idSet}$$

Process terms may contain free process and set variables which will be defined by other declarations. Variables are only instantiated when analysis is performed on a system so declarations may appear in any order or even in different files. The grammar shown in Figure A.1 and Figure A.2 defines the full syntax of a `.ccs` file.

```
tokens

  "[a-zA-Z][a-zA-Z0-9'_-]*" => NEWID of string
  "[1-9][0-9]{0,8} | 0" => INT of string

grammar

  ccs_file : binding_list

  binding : "proc" id "=" agent
          | "set" id "=" id_set

  agent : "nil"
        | "@"
        | id
        | act "." agent
        | agent "+" agent
        | agent "|" agent
        | agent "\" restriction
        | agent "[" relabeling "]"
        | agent "where" agent_frame "end"
        | automaton
        | "(" agent ")"

  act : "t"
      | id
      | "'" id

  restriction : id_set
              | id

  relabeling : rename_list

  rename : id "\" id

  ag_eqn : id "=" agent

  id_set : "{" id_list "}"

  id : NEWID

  agent_frame : ag_eqn_list

  automaton : "Aut" "(" "start" "=" int "," state_list ")"

  state : int ":" trans_list

  trans : act int_list

  int : INT
```

Figure A.1: Syntax of CCS - part 1

```
lists

  binding_list is non_empty_list EMPTY_STR EMPTY_STR EMPTY_STR of binding
  rename_list is non_empty_list EMPTY_STR "," EMPTY_STR of rename
  id_list is non_empty_list EMPTY_STR "," EMPTY_STR of id
  ag_eqn_list is non_empty_list EMPTY_STR "and" EMPTY_STR of ag_eqn
  state_list is non_empty_list EMPTY_STR EMPTY_STR EMPTY_STR of state
  trans_list is empty_list EMPTY_STR EMPTY_STR EMPTY_STR of trans
  int_list is empty_list LCBRACE COMMA RCBRACE of int

priorities

  noassoc 40 "where"
  right   50 "+"
  right   60 "|"
  noassoc 70 "\"
  noassoc 80 "."
  noassoc 85 "["
  noassoc 90 "nil" "@"
```

Figure A.2: Syntax of CCS - part 2

A.1.2 CCS Semantics

The operational semantics of CCS is shown in Figure A.3 and Figure A.4. The rules define the relation:

$$transitions \in agent\ env \times set\ env \times agent \times act \times agent$$

The first two components are environments which map identifiers to agents and sets respectively. If $\langle ae, se, p, a, p' \rangle \in transitions$ then we write:

```
ae, se :  p -- a --> p'
```

and read this as "p may perform an action a and become p' when free agent and set variables are resolved by the bindings defined in ae and se."

```
RULE_SET transitions

vars

  a , b : act
  p , p' , p1 , p1' , p2 , p2' : agent
  s : id_set
  r, r1, r2 : relabeling
  ae : (agent env)
  se : (id_set env)
  A : id
  s_id : id
  ag_frame : (agent frame)
  aut, aut' : automaton

rules

prefix

 --------------------------
  ae, se : a.p -- a --> p

sum_1

     ae, se : p1 -- a --> p1'
 -------------------------------
  ae, se : p1 + p2 -- a --> p1'

sum_2

     ae, se : p2 -- a --> p2'
 -------------------------------
  ae, se : p1 + p2 -- a --> p2'

parallel_1

        ae, se : p1 -- a --> p1'
 ------------------------------------
  ae, se : p1 | p2 -- a --> p1' | p2

parallel_2

       ae, se : p2 -- a --> p2'
 ------------------------------------
  ae, se : p1 | p2 -- a --> p1 | p2'

parallel_3

  ae, se : p1 -- a --> p1' , ae, se : p2 -- b --> p2', inverses(a,b)
 ---------------------------------------------------------------------
               ae, se : p1 | p2 -- t --> p1' | p2'
```

Figure A.3: Semantics of CCS - part 1

```
relabel1

    ae, se : p[compose(r1 : r2)] -- a --> p'
  -------------------------------------------
       ae, se : p[r1][r2] -- a --> p'

relabel2

           ae, se : p -- a --> p'
 ------------------------------------------
    ae, se : p[r] -- apply(a,r) --> p'[r]

restrict_1

     ae, se : p -- a --> p', not (labelof(a) in s)
 --------------------------------------------------
            ae, se : p\s -- a --> p'\s

restrict_2

     ae, se : p -- a --> p', not (labelof(a) in lookup(s_id,se))
 ------------------------------------------------------------------
                ae, se : p\s_id -- a --> p'\s_id

aut_rule

        aut - a -> aut'
 ------------------------------
   ae, se : aut -- a --> aut'

process_constant

   ae, se : lookup(A,ae) -- a --> p'
 ------------------------------------
       ae, se : A -- a --> p'

fixpoint

           push_frame(ag_frame,ae), se : p -- a --> p'
 ---------------------------------------------------------------
   ae, se : p where ag_frame end -- a --> p' where ag_frame end

end
```

Figure A.4: Semantics of CCS - part 2

A.2 Prioritized CCS

For a description of this version of CCS with priorities, which we refer to as PCCS, see [5, 6]. PCCS extends CCS by allowing different priority values to be assinged to actions to give higher priority communications preemptive power. Priority values are natural numbers and we adopt the convention that 0 has the highest priority and increasing numbers imply decreasing priorities. To indicate that action a has priority n, we write `a:n`.

A.2.1 Syntax of Prioritized CCS

The structure of a .pccs file is exactly the same as that for a .ccs file as described above, namely it constists of a sequence of process and identifier set declarations. The comment character is `*`; all text between a `*` and the end of line character is ignored. The grammar shown in Figure A.1 and Figure A.2 defines the full syntax of a `.pccs` file.

```
tokens

  "[a-zA-Z][a-zA-Z0-9'_-]*" => ID of string
  "[0-9][0-9]{0,8}" => INT of string

grammar

  pccs_file : binding_list

  binding : "proc" id "=" agent
          | "set" id "=" id_set

  agent : "nil"
        | "@"
        | id
        | act "." agent
        | agent "+" agent
        | agent "|" agent
        | agent "\" restriction
        | agent "[" relabeling "]"
        | agent "[>" agent
        | agent "where" agent_frame "end"
        | automaton
        | "(" agent ")"

  act : basic_act ":" int

  basic_act : "tau"
            | id
            | "'" id
            | "^" id

  restriction : id_set
              | id

  relabeling : rename_list

  rename : id "/" id

  ag_eqn : id "=" agent

  id_set : "{" id_list "}"

  agent_frame : ag_eqn_list

  id : ID

  automaton : "Aut" "(" "start" "=" int "," state_list ")"

  state : int ":" trans_list

  trans : act int_list

  int : INT
```

Figure A.5: Syntax of PCCS

```
lists

  binding_list is non_empty_list EMPTY_STR EMPTY_STR EMPTY_STR of binding
  rename_list is non_empty_list EMPTY_STR "," EMPTY_STR of rename
  id_list is non_empty_list EMPTY_STR "," EMPTY_STR of id
  ag_eqn_list is non_empty_list EMPTY_STR EMPTY_STR EMPTY_STR of ag_eqn
  state_list is non_empty_list EMPTY_STR EMPTY_STR EMPTY_STR of state
  trans_list is empty_list EMPTY_STR EMPTY_STR EMPTY_STR of trans
  int_list is empty_list "{" "," "}" of int

priorities

  left    10 "[>"
  left    20 "|"
  noassoc 50 "\"
  noassoc 60 "[" "]" "{" "}" "where"
  left    70 "+"
  right   80 "."
  noassoc 90 "nil" "@"
  noassoc 100 ":"
```

Figure A.6: Syntax of PCCS - part 2

A.2.2 Semantics of Prioritized CCS

The operational semantics of PCCS is shown in Figures A.7, A.8, A.9, A.10, A.11, and A.12. The rules define two relations, a *transitions* relation and an auxiliary relation called *init_acts* which is used in defining the *transitions* relation.

The rules define the relation:

$$transitions \in agent\ env \times set\ env \times agent \times act \times agent$$

The first two components are environments which map identifiers to agents and sets respectively. If $\langle ae, se, p, a, p' \rangle \in transitions$ then we write:

```
ae, se :  p -- a --> p'
```

and read this as "p may perform an action a and become p' when free agent and set variables are resolved by the bindings defined in ae and se."

```
RULE_SET transitions

vars

  a, b, c : act
  p , p' , p1 , p1' , p2 , p2' : agent
  s  : id_set
  r  : relabeling
  ae : (agent env)
  se : (id_set env)
  A  : id
  s_id : id
  ag_frame : (agent frame)
  aut, aut' : automaton

rules

prefix

 --------------------------
  ae, se : a.p -- a --> p

sum1

    ae, se : p1 -- a --> p1', not (t in init(ae,se,prio(a),p2))
 ----------------------------------------------------------------
                 ae, se : p1 + p2 -- a --> p1'

sum2

    ae, se : p2 -- a --> p2', not (t in init(ae,se,prio(a),p1))
 ----------------------------------------------------------------
               ae, se : p1 + p2 -- a --> p2'

par1

  ae, se : p1 -- a --> p1', not (t in init(ae,se,prio(a),p1 | p2))
 -------------------------------------------------------------------
              ae, se : p1 | p2 -- a --> p1' | p2

par2

    ae, se : p2 -- a --> p2' , not (t in init(ae,se,prio(a),p1 | p2))
 ----------------------------------------------------------------------
              ae, se : p1 | p2 -- a --> p1 | p2'

par3

    ae, se : p1 -- a --> p1',   ae, se : p2 -- b --> p2',
    inverses(a,b) and not (t in init(ae,se,prio(b),p1 | p2))
 --------------------------------------------------------------
        ae, se : p1 | p2 -- t:prio(b) --> p1' | p2'
```

Figure A.7: Semantics of PCCS - (transitions relation, part 1)

```
par4

  ae, se : p1 -- a --> p1', ae, se : p2 -- b --> p2',
  poll_inverses(a,b) and not (t in init(ae,se,prio(b),p1 | p2))
 ---------------------------------------------------------------
       ae, se : p1 | p2 -- t:prio(b) --> p1 | p2'

par5

  ae, se : p1 -- a --> p1', ae, se : p2 -- b --> p2',
  poll_inverses(b,a) and not (t in init(ae,se,prio(b),p1 | p2))
 ---------------------------------------------------------------
      ae, se : p1 | p2 -- t:prio(a) --> p1' | p2

relabel

           ae, se : p -- a --> p'
 -----------------------------------------
    ae, se : p[r] -- apply(a,r) --> p'[r]

disabling1

   ae, se : p1 -- a --> p1', not (t in init(ae,se,prio(a),p2))
  -------------------------------------------------------------
            ae, se : p1 [> p2 -- a --> p1' [> p2

disabling2

    ae, se : p2 -- a --> p2', not (t in init(ae,se,prio(a),p1))
  -----------------------------------------------------------------
                  ae, se : p1 [> p2 -- a --> p2'

restrict_1

     ae, se : p -- a --> p', not (labelof(a) id_in s)
 ----------------------------------------------------
               ae, se : p\s -- a --> p'\s

restrict_2

     ae, se : p -- a --> p', not (labelof(a) id_in lookup(s_id,se))
 ---------------------------------------------------------------------
                ae, se : p\s_id -- a --> p'\s_id

aut_rule

      aut --- a ---> aut'
 ------------------------------
   ae, se : aut -- a --> aut'
```

Figure A.8: Semantics of PCCS - (transitions relation, part 2)

```
process_constant

   ae, se : lookup(A,ae) -- a --> p'
 -------------------------------------
       ae, se : A -- a --> p'

fixpoint

           push_frame(ag_frame,ae), se : p -- a --> p'
 ----------------------------------------------------------------
   ae, se : p where ag_frame end -- a --> p' where ag_frame end

end
```

Figure A.9: Semantics of PCCS - (transitions relation, part 3)

```
RULE_SET init_acts

vars

  ae : (agent env)
  se : (id_set env)
  p, p1, p2 : agent
  a, b : act
  c : id
  j,k : int
  n, s_id, A : id
  s : id_set
  r : relabeling
  ag_frame : (agent frame)

rules

prefix
        prio(a) < k
  --------------------------
    ae, se, k : a.p - a ->


sum1

      ae, se, k : p1 - a ->
  ------------------------------
    ae, se, k : p1 + p2 - a ->

sum2

      ae, se, k : p2 - a ->
  ------------------------------
    ae, se, k : p1 + p2 - a ->

par1

      ae, se, k : p1 - a ->
  ------------------------------
    ae, se, k : p1 | p2 - a ->

par2

      ae, se, k : p2 - a ->
  ------------------------------
    ae, se, k : p1 | p2 - a ->

par3

    ae, se, k : p1 - b -> , ae, se, k : p2 - a -> , inverses(b,a)
  -----------------------------------------------------------------
               ae, se, k : p1 | p2 - t:prio(b) ->
```

Figure A.10: Semantics of PCCS - (init_acts relation, part 1)

```
par4

    ae, se, k : p1 - b -> , ae, se, k : p2 - a -> , poll_inverses(a,b)
  --------------------------------------------------------------------
               ae, se, k : p1 | p2 - t:prio(b) ->


par5

    ae, se, k : p1 - b -> , ae, se, k : p2 - a -> , poll_inverses(b,a)
  --------------------------------------------------------------------
               ae, se, k : p1 | p2 - t:prio(b) ->

restrict_1

    ae, se, k : p - a -> , not (labelof(a) id_in s)
 ----------------------------------------------------
               ae, se, k : p\s - a ->

restrict_2

     ae, se, k : p - a -> , not (labelof(a) id_in lookup(s_id,se))
 -------------------------------------------------------------------
                   ae, se, k : p\s_id - a ->

relabel

         ae, se, k : p - a ->
 ------------------------------------
   ae, se, k : p[r] - apply(a,r) ->

disabling1

       ae, se, k : p1 - a ->
  -------------------------------
    ae, se, k : p1 [> p2 - a ->

disabling2

       ae, se, k : p2 - a ->
  ------------------------------
    ae, se, k : p1 [> p2 - a ->
```

Figure A.11: Semantics of PCCS - (init_acts relation, part 2)

```
process_constant

   ae, se, k : lookup(A,ae) - a ->
 -----------------------------------
        ae, se, k : A - a ->

fixpoint

   push_frame(ag_frame,ae), se, k : p - a ->
 ---------------------------------------------
    ae, se, k : p where ag_frame end - a ->

end
```

Figure A.12: Semantics of PCCS - (init_acts relation, part 3)

A.3 Basic Lotos

For an introduction to Basic Lotos, see [1]. We adopt the standard notation for all operators, but, for efficiency reasons, do not allow local process definitions.

A.3.1 Syntax of Basic Lotos

A .lotos file simply consists of a sequence of process declarations. The full syntax of a .lotos file is shown in Figures A.13 and A.14.

```
tokens

  "[a-zA-Z][a-zA-Z0-9'_-]*" => NEWID of String
  "[1-9][0-9]{0,8} | 0" => INT of String

grammar

  lotos_file : process_def_list

  process : "process" id "[" id_list "]" ":="
            bexp "endproc"

  bexp : "stop"
       | user_act ";" bexp
       | bexp "[]" bexp
       | bexp "|[" id_set "]|" bexp
       | bexp "||" bexp
       | bexp "|||" bexp
       | "hide" id_set "in" bexp
       | id "[" id_list "]"
       | process "(" id ")"
       | "exit"
       | bexp ">>" bexp
       | bexp "[>" bexp
       | bexp "[" relabeling "]"
       | automaton
       | "(" bexp ")"

  relabeling : rename_list

  rename : id "/" id

  user_act  : "i"
            | id

  act : user_act
      | "delta"

  id : NEWID

  id_set : id_list

  automaton : "Aut" "(" "start" "=" int "," state_list ")"

  state : int ":" trans_list

  trans : act int_list

  int : INT
```

Figure A.13: Syntax of Basic Lotos - part 1

```
lists

  id_list is non_empty_list EMPTY_STR "," EMPTY_STR of id
  process_def_list is non_empty_list EMPTY_STR EMPTY_STR EMPTY_STR of process
  rename_list is non_empty_list EMPTY_STR "," EMPTY_STR of rename
  state_list is non_empty_list EMPTY_STR EMPTY_STR EMPTY_STR of state
  trans_list is empty_list EMPTY_STR EMPTY_STR EMPTY_STR of trans
  int_list is empty_list "{" "," "}" of int

priorities

  left   10 ">>" "[>"
  right   20 "|[" "]|" "||" "|||"
  noassoc 30 "[" "]"
  noassoc 60 "where" "endproc" "in" "exit"
  right   70 "[]"
  noassoc 80 ";"
  noassoc 90 "stop"
```

Figure A.14: Syntax of Basic Lotos - part 2

A.3.2 Semantics of Basic Lotos

The operational semantics of Basic Lotos is shown in Figures A.15, A.16, A.17, and A.18.

```
RULE_SET transitions

vars

  a, b                    : user_act
  c, d                    : act
  pe                      : (proc env)
  p, p', p1, p1', p2, p2' : bexp
  S                       : id_set
  r, r1, r2               : relabeling
  actual_gates           : (id list)
  proc_name               : id
  aut,aut'                : automaton

rules

prefix

  ----------------------
    pe : a;p -- a --> p

choice1

       pe : p1 -- c --> p1'
  -----------------------------
    pe : p1 [] p2 -- c --> p1'

choice2

       pe : p2 -- c --> p2'
  -----------------------------
    pe : p1 [] p2 -- c --> p2'

par1_1

    pe : p1 -- a --> p1', not (labelof(a) in S)
  ------------------------------------------------
    pe : p1 |[ S ]| p2 -- a --> p1' |[ S ]| p2

par1_2

    pe : p2 -- a --> p2', not (labelof(a) in S)
  ----------------------------------------------
    pe : p1 |[ S ]| p2 -- a --> p1 |[ S ]| p2'

par1_3

    pe : p1 -- c --> p1' , pe : p2 -- d --> p2',
      ((c = d) and ((labelof(c) in S) or (c = delta)))
  ------------------------------------------------------
      pe : p1 |[ S ]| p2 -- c --> p1' |[ S ]| p2'
```

Figure A.15: Semantics of Basic Lotos - part 1

```
par2_1

        pe : p1 -- i --> p1'
  ------------------------------------
    pe : p1 || p2 -- i --> p1' || p2

par2_2

        pe : p2 -- i --> p2'
  ------------------------------------
    pe : p1 || p2 -- i --> p1 || p2'


par2_3

    pe : p1 -- c --> p1' , pe : p2 -- d --> p2', (c = d) and (not (c = i))
  -------------------------------------------------------------------------
                 pe : p1 || p2 -- c --> p1' || p2'

par3_1

         pe : p1 -- a --> p1'
  -------------------------------------
    pe : p1 ||| p2 -- a --> p1' ||| p2

par3_2

         pe : p2 -- a --> p2'
  -------------------------------------
    pe : p1 ||| p2 -- a --> p1 ||| p2'

par3_3

    pe : p1 -- delta --> p1' , pe : p2 -- delta --> p2'
  ------------------------------------------------------
        pe : p1 ||| p2 -- delta --> p1' ||| p2'

hide1

   pe : p -- c --> p', (not (labelof(c) in S))
  ----------------------------------------------
    pe : hide S in p -- c --> hide S in p'

hide2

    pe : p -- a --> p',labelof(a) in S
  -----------------------------------------
   pe : hide S in p -- i --> hide S in p'

termination

  -------------------------------
    pe : exit -- delta --> stop
```

Figure A.16: Semantics of Basic Lotos - part 2

```
enabling1

        pe : p1 -- a --> p1'
  ------------------------------------
    pe : p1 >> p2 -- a --> p1' >> p2

enabling2

     pe : p1 -- delta --> p1'
  ------------------------------
    pe : p1 >> p2 -- i --> p2

disabling1

        pe : p1 -- a --> p1'
  ------------------------------------
    pe : p1 [> p2 -- a --> p1' [> p2

disabling2

       pe : p1 -- delta --> p1'
  ----------------------------------
    pe : p1 [> p2 -- delta --> p1'

disabling3

       pe : p2 -- c --> p2'
  ------------------------------
    pe : p1 [> p2 -- c --> p2'

relabeling1

    pe : p[compose(r1 : r2)] -- c --> p'
  -----------------------------------------
       pe : p[r1][r2] -- c --> p'


relabeling2

    pe : p -- c --> p', not_relabeling(p)
  -----------------------------------------
    pe : p[r] -- apply(c,r) --> p'[r]
```

Figure A.17: Semantics of Basic Lotos - part 3

```
proc_instantiation1

  pe : get_bexp(lookup(proc_name,pe))[mk_relabeling
                                       (
                                         proc_name --
                                         actual_gates --
                                         get_formal_gates(lookup(proc_name,pe))
                                       )
                                      ] -- c --> p'
  ---------------------------------------------------------------------------
             pe : proc_name[actual_gates] -- c --> p'

proc_instantiation2

    pe : get_bexp(lookup(proc_name,pe)) -- c --> p',
  ------------------------------------------------------
        pe : process(proc_name) -- c --> p'

aut_rule

      aut - a -> aut'
 ---------------------------
   pe : aut -- a --> aut'


end
```

Figure A.18: Semantics of Basic Lotos - part 4

Bibliography

[1] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[2] U. Celikkan. *Semantic Preorders in the Automated Verification of Concurrent Systems.* PhD thesis, North Carolina State University, 1995.

[3] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[4] R. Cleaveland. On automatically explaining bisimulation inequivalence. In E.M. Clarke and R.P. Kurshan, editors, *Computer Aided Verification (CAV '90)*, volume 531 of *Lecture Notes in Computer Science*, pages 364–372, Piscataway, NJ, June 1990. Springer-Verlag.

[5] R. Cleaveland and M.C.B. Hennessy. Priorities in process algebra. *Information and Computation*, 87(1/2):58–77, July/August 1990.

[6] R. Cleaveland, G. Luettgen, V. Natarajan, and S. Sims. Modeling and verifying distributed systems using priorities: A case study. *Software Concepts and Tools*, 17:50–62, 1996.

[7] R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. In E. Brinksma, R. Cleaveland, K.G. Larsen, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 153–173, Aarhus, Denmark, May 1995. Springer-Verlag.

[8] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[9] R. Cleaveland and S. Sims. The concurrency workbench of north carolina. URL http://www4.ncsu.edu/ rance/WWW/cwb-nc.html.

[10] R. Cleaveland and S. Sims. The process algebra compiler of north carolina. URL http://www4.ncsu.edu/ stsims/WWW/pac/pac-nc.html.

[11] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. Henzinger, editors, *Computer Aided Verification (CAV '96)*, Lecture Notes in Computer Science, pages 394–397, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[12] M.C.B. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 32(1):137–161, January 1985.

[13] R. Milner. *Communication and Concurrency.* Prentice-Hall, London, 1989.

[14] F. Moller and P. Stevens. *The Edinburgh Concurrency Workbench (Version 7.0).* University of Edinburgh, November 1994.

[15] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, December 1987.

[16] G Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, Denmark, 1981.

Index