The Concurrency Workbench of the New Century

Version 1.2

July 6, 2000

User's Manual

Rance Cleaveland
rance@cs.sunysb.edu

Tan Li
tanli@cs.sunysb.edu

Steve Sims
sims@reactive-systems.com

Dept. of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400

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 the New Century (CWB-NC, previously called Concurrency Workbench of North Carolina) [23, 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

The CWB-NC supports several formulations of the verification question.

The simplest type of verification supported by the tool is reachability analysis. Here, as in each type of verification, the first step in using the tool is to write a description of the system at hand in one of the several design languages supported by the CWB-NC. The description is then parsed by the tool and checked for syntactic correctness. The user then gives a logical formula describing a "bad state" that the system should never reach. Given such a formula and system description, the CWB-NC explores every possible state the system may reach during execution and checks to see if a bad state is reachable. If a bad state is detected, a description of the execution sequence leading to the state is reported to the user. Many bugs such as deadlock and critical section violations may be found using this approach.

Reachability analysis is actually a special case of a more general type of verification called model checking. In the model checking approach a system is again described using a design language and a property the system should have is formulated as a logical formula. However, in model checking rather than specifying a "bad state" to be searched for among the reachable states of the system, the given formula defines a behavior the system should or should not have as it executes. The logic used for expressing formulas contains temporal operators enabling one to describe how a system behaves as time passes rather than simply a characteristic of the system at a particular point in time. Using such a *temporal logic* one can state properties such as the following:

- "eventually event a will occur"
- "it is always the case that after an a event the event b eventually occurs"

The CWB-NC includes a model checker for determining whether systems satisfy formulas written in two different temporal logics, the modal mu-calculus and GCTL* . The former is a very expressive, and many other logics may be efficiently translated into the mu-calculus; therefore, it may serve as the basis for model checking in a variety of different logics. The CWB-NC checks CTL [6] formulas using this approach. The drawback of the mu-calculus is that is is difficult to encode certain kinds of properties, such as those involving fairness constraints. GCTL* [1] captures such properties more easily.

A third type of verification supported by the CWB-NC involves using a design language for defining both systems and specifications. Here the specification describes a system behavior more abstractly than the system description. A binary relation over terms in the design language is defined such that a system satisfies a specification if the two terms in the design language (the system and specification) are related by the relation (called a behavioral relation). Two basic approaches have been advocated for defining behavioral relations. If a relation β is an equivalence relation (i.e. a relation that is reflexive, symmetric, and transitive), then two terms related by β *behave the same* according to some criteria. Different relations may be defined to capture different notions of behaving the same. In a second approach β is defined as a preorder (i.e. a relation that is reflexive and transitive) and $\langle Sys, Spec \rangle \in \beta$ indicates that *Sys behaves the same or better* than *Spec*. Efficient algorithms have been developed for equivalence and preorder checking [24, 5] and routines for performing these types of verification have been implemented in the CWB-NC. The CWB-NC provides appropriate diagnostic information for explaining why two systems fail to be related by a given semantic equivalence or preorder.

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) [22, 10]. Using this tool a number of front ends have been implemented to support a number of different design languages which are listed in Section 3.1.

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.

Acknowledgements. The authors of this manual would like to thank S. Arun-Kumar and Alex Groce for contributing the implementation of the SCCS front-end and GCTL* model checker, respectively.

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://www.cs.sunysb.edu/~cwb`

This web page also includes the directions that follow.

2.2 Revision History

2.2.1 Version 1.2

This is the current version of CWB-NC and was released in June 2000. The official name for the CWB-NC was changed to *the Concurrency Workbench of the New Century* in honor of the group's move from NCSU to SUNY at Stony Brook.

New Features

- Model-checking for the temporal logic GCTL* [1], a extension of traditional CTL* [12], is now supported. See the man page for `chk` for the details about how to check if a system satisfies a GCTL* formula. Files containing GCTL* formulas should use `.gctl` as the file extension.

- A model-checker based on Alternating Büchi Tableau Automata (ABTAs) [1, 2] checker has been implemented. The GCTL* model checker works by translating GCTL* formulas into ABTAs; other temporal logics may be checked similarly. Currently the ABTA model checker is activated if the property is written in GCTL* formula, while CTL and mu-calculus still use the existing mu-calculus model checkers.

- Support for a new designed language, S(ynchronous)CCS, has been included in this release.

- The CWB-NC now runs on win32 platforms (Windows 95/98/NT).

Improvements

- τ-actions in TCCS may now have labels attached. This permits the "cause" of a synchronization to be observed.
- A Red Hat Linux RPM package has been included in this release.
- Pre-compiled distributions are now stand-alone executable binaries rather than heap-image binaries. Users can run these stand-alone executables without installing the SML/NJ run-time environment. Pre-compiled heap image binaries will be discontinued from this release.
- The installation procedure has been upgraded. Users may now reconfigure the CWB-NC by re-running the installation script appropriately.

Changes

- The standard distribution has been renamed as the "base distribution."

2.2.2 Version 1.11

This version was released in May 1998.

Improvements

- Slight performance improvements were made to the reachability analysis routine (`search`).
- The tool now compiles under SML of New Jersey Version 110, the latest general release. If you are obtaining a binary distribution, this will not affect you.

Bug Fixes

1. An overflow error in a hash function was corrected.
2. Several bugs in the `es` command were fixed.

2.2.3 Version 1.1

Version 1.1 became available in September, 1997. This release provides several bug fixes, several new features, and some minor improvements to the documentation.

New Features

- A general purpose reachability analysis routine searches a system's state space for a state satisfying a given formula. When such a state is found, the simulator is invoked with the path to the state pre-loaded. See the man page for the `search` command for details.
- A find deadlock `fd` command has been implemented as a special case of the reachability routine.
- A version of the simulator for the graphical user interface has been implemented. This is particularly useful for examining sequences returned by the `search` and `fd` commands.
- The `save` command was added to save an automaton to a file after it has been compiled from a system description.

- The `transitions` command and simulator may now be invoked in "observational equivalence" mode meaning that each transition displayed may in fact be a sequence of transitions such that zero or more internal actions are followed by an action (either visible or internal) which is followed by zero or more additional internal actions.

- Support for two new design languages (Timed CCS and CSP) is included in this release; however, these interfaces have not been tested extensively, so they should be considered an alpha release. Please report any bugs encountered.

Changes

- In the syntax of the mu-calculus, braces ({ and }) are no longer used for the action set arguments to the modal operators. See Section 5.1 for details.

Bug Fixes

1. In alternation free model checker, nested variables were handled incorrectly for some formulas.

2. The translator to L2 format worked incorrectly in some cases.

3. The strongly connected components routine used in the graph transformation to compute observational equivalence failed in some cases.

4. In the Basic LOTOS command line, "min foo[a,b] bar[c,d]" was accepted rather than reporting an error since the second argument to min must be an identifier.

2.2.4 Version 1.0

The original release of the tool was labeled Version 1.0 and occurred in September, 1996.

2.3 Installing the CWB-NC

2.3.1 Installing Binaries

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

Architecture	Operating System
SPARC	Solaris 2.x
x86	Linux
x86	win32 (Windows 95/98/NT)

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

Generic Solaris/Linux Binary Installation

To install a binary distribution under Solaris/Linux 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 gzipped tar files. The first tar file contains the CWB-NC base distribution comprising examples, documentation, and other common files. The other compressed tar file(s) you must obtain are pre-compiled stand-alone executable 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:

   - The CWB-NC installation script:
     - install-cwb-nc.sh
   - Base distribution:
     - cwb-nc.tar.gz
   - Pre-compiled binaries:
     - cwb-nc-sparc-solaris-bin.tar.gz
     - cwb-nc-x86-linux-bin.tar.gz

   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 (of the cwb-nc home directory). 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 to use the CWB-NC graphical user interface, then Expectk (version 5.20 or later) 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/expect.tar.gz`. 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. Make sure install-cwb-nc.sh is executable.

```
% chmod +x install-cwb-nc.sh
```

5. Run the installation script.

```
% install-cwb-nc.sh
```

   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.

6. 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.

Installing Linux Binaries using Red Hat RPM

If you are running Red Hat Linux 6.0 or higher, you may use the CWB-NC RPM package to expediate the installation. To use this option do the following,

1. Download cwb-nc-1.2-0.i386.rpm

2. You have to have the root privilege to install the RPM packege. Type the following command,

```
%rpm -i cwb-nc-1.2-0.i386.rpm
```

   You are done! By default the CWB-NC is installed at /usr/share/cwb-nc. If you want to customerize the installation path, you may use the following command instead.

```
%rpm -i --prefix <your CWB-NC path> cwb-nc-1.2-0.i386.rpm
```

   Installation procedure will automatically search for expectk by examining the PATH environment variable. Redhat Linux 6.0 has included expectk. If you chose the standard setup when installing the linux, expectk most likely has been installed at

```
/usr/bin/expectk
```

   Once expectk has been found, the installation procedure will configure the script file for the CWB-NC GUI. You may use flag `-gui` to activate the CWB-NC GUI.

To uninstall the CWB-NC, type

```
%rpm -e cwb-nc-1.2-0
```

Remember that you need root privilege to install/uninstall RPM packages.

Installing the CWB-NC on Win32

Beginning with Version 1.2, a binary distribution of the CWB-NC is also available for Win32 platforms (Windows95/98/NT/2000). The installation process for the CWB-NC on win32 platforms follows the standard installation procedure recommended by Microsoft.

To install the CWB-NC on Win32 platforms do the following,

1. Download cwb-nc-win32-1.2.zip

2. Unzip the downloaded file to a temporary directory. There are many compress/uncompress windows utilities supporting the ZIP format; for example, you may check WINZIP's homepage `www.winzip.com` for more information.

3. Double-click on *setup.exe* to start the installation, then follow the instructions. Note that long path names are not supported by current SML/NJ, so avoid using them for the target directory of CWB-NC. Setup will create a new program group for CWB-NC. Each supported design language will have a shortcut in the group. You may click the shortcuts to launch CWB-NC for the given language. You may also run CWB-NC by typing

   ```
   <your CWB-NC path>\bin\cwb-nc <lang>
   ```

   at the DOS prompt.

To uninstall the CWB-NC, click "Uninstall the CWB-NC" icon in CWB-NC program group.

2.3.2 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 110, which is available free of charge from :

`http://cm.bell-labs.com/cm/cs/what/smlnj/index.html`

This is the latest general release of SML-NJ. The CWB-NC will most likely compile under subsequent working releases although we have not done so. In addition to the compiler you will also need ml-lex, ml-yacc, and the Compilation Manager all of which are included with the distribution located at the above URL. SML-NJ 110 is available on the following platforms:

Architecture	Operating System
DEC Alpha	Digital Unix 4.0
HP HPPA	HPUX 10.1
SGI MIPS R4400, R10000	Irix 5.3,6.2-6.4
Sparc	SunOS 4.1.3, Solaris 2.5.5
IBM RS6000, PowerPC 601	Aix 3.2,4.1
Intel x86	FreeBSD, Linux, win32

The CWB-NC graphical user interface requires Expectk Version 5.20 or later. See the 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 gzipped 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:

   - The CWB-NC installation script:
     - install-cwb-nc.sh
   - Base distribution:
     - cwb-nc.tar.gz
   - Source distribution:
     - cwb-nc-src.tar.gz

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

3. Make sure install-cwb-nc.sh is executable.

   ```
   % chmod +x install-cwb-nc.sh
   ```

4. Run the installation script.

   ```
   % install-cwb-nc.sh
   ```

   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 case, locate the Borne Shell on your system and change the first line of install-cwb-nc.sh. 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 110 must be installed on each platform).

5. You will be asked which design languages you want to support. The nstallation script will configure the source code according to your choice(s). You may choose if the code for the chosen languages are also to be compiled during the installation. Tips: Configuring support for a design language won't cost you extra time but compilation will, so answer yes for all the languages you wish to support. You may run the following command later to compile a design language

   ```
   % <your CWB-NC path>/src/make-cwb <lang>
   ```

   The configuration files are automatically created by the installation script according to the paths you entered. If you relocate the whole CWB-NC directory tree to a different place, just run the installation script again to re-configure the CWB-NC.

6. The script for running the system may be found in `<your CWB-NC path>/bin/cwb-nc`. For details on how to use the system, see Chapter 3 of this manual. Have fun!

2.4 Feedback and Bug Reports

Please send comments and bug reports to `cwb-nc@cs.sunysb.edu.` All feedback is greatly appreciated. Over the first year and a half of its existence the CWB-NC has benefited greatly from user feedback. Thanks to all who have given comments.

Chapter 3

Using the CWB-NC

This section 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 create a *program* in the desired design language 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 investigate 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) [20]	`ccs`
Milner's Synchronous Calculus of Communicating Systems (SCCS) [19]	`sccs`
Version of CCS with prioritized actions [8]	`pccs`
Version of CCS with timed actions [21]	`tccs`
Hoare's Communicating Sequential Processes (CSP) [17]	`csp`
Basic Lotos [3]	`lotos`

The examples here use CCS as the design language. The tool is used in exactly the same fashion for other languages. See Section 7 for details on the syntax and semantics of each language. 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. This section introduces some of the 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 the New Century
(Version 1.2 --- April, 2000)

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 trace equivalence between agents A1 and A2.

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

The `-S` flag indicates that the *semantics* to be used is *trace* semantics. When no `-S` flag is given, observational equivalence is computed by default.

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`.

A \ 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` with respect to observational equivalence.

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 CWB-NC distribution. The file includes two versions of the protocol; the first employs a safe medium that never loses any messages, and the second uses a lossy medium that may arbitrarily drop messages. The sequence of CWB-NC commands and responses described in the following appears in Figure 3.2.

After the CCS-text version of the CWB-NC is started, 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.

```
**********************************************************************
*  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

We now compute whether `ABP-safe` and `Spec` are observationally equivalent. The CWB-NC equivalence checking routines are invoked with the `eq` command, with the semantics flag `-S` *semantics-type* indicating the particular equivalence relation to be computed. The `obseq` argument given here indicates observational equivalence. In the case of this invocation of the `eq` command the result is displayed virtually instantly; however, for larger systems that take longer to analyze, messages such as `Building automaton...` and `Done building automaton` report the progress of the computation to the user. The number of states and transitions constructed to model to behavior of `Spec` and `ABP-lossy` is also reported. 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 4.1 for an explanation of distinguishing formulas. Note that the next `eq` command issued has no `-S` flag; here, the default value, `obseq`, of the `-S` flag is used, and the CWB-NC checks whether `ABP-lossy` is observationally equivalent to `Spec`. In this case the two agents are indeed 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 or CTL 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`. This deadlock could alternatively been detected with the `fd` command (**f**ind **d**eadlock) which finds the deadlock and then invokes the simulator with the path to the deadlock state preloaded. Note that the `fd` command is a specific case of the more general `search` command described in Section 6.2.

3.3 The CWB-NC Graphical User Interface

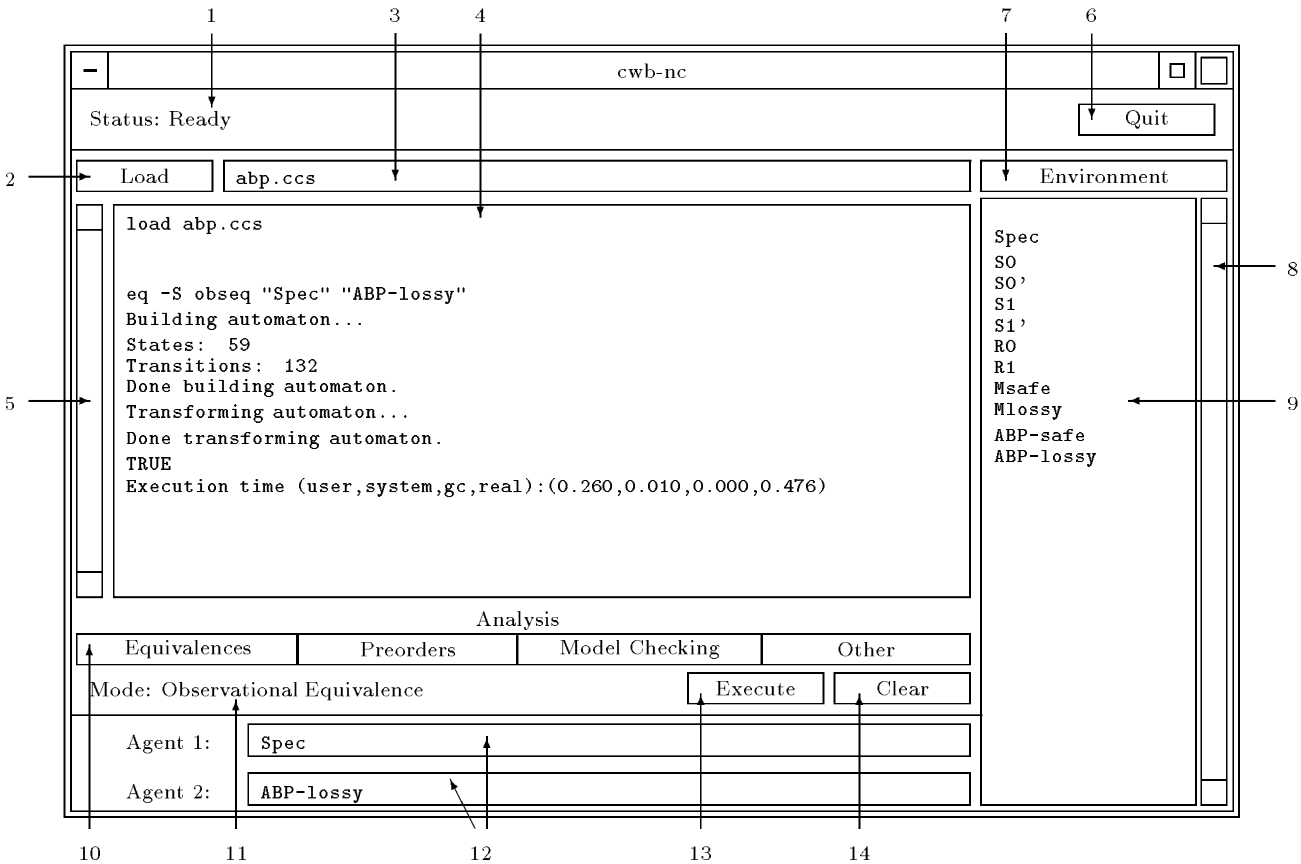
Figure 3.3 contains a diagram of the CWB-NC graphical user interface. To demonstrate how to use the GUI, we describe the sequence of interactions which lead to the GUI state shown in the diagram. In the following we write click(n) to mean clicking with mouse button n.

- 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 to perform the given task.

- 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. To view the item a variable is bound to double-click(1) on the desired variable in the variable list.



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.

- 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, one changes the analysis mode by selecting an item from a menu on the analysis menu bar. For example, by selecting `Bisimulation` from the `Equivalences` menu one may check whether two agents are bisimulation equivalent. When this menu item is selected the current analysis mode changes from observational equivalence to bisimulation equivalence and the CWB-NC is ready to compute bisimulation 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 bisimulation equivalent and a *distinguishing formula* (`[t]ff`) satisfied by `Spec` but not by `ABP-lossy` is displayed. See Section 4.1 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.

Chapter 4

Behavioral-Relation-Based Verification

The preceding chapter introduced the `eq` command as the vehicle for invoking the CWB-NC equivalence checking routines. The `-S` flag indicates the particular equivalence relation to check. Currently supported arguments are `bisim` (bisimulation equivalence), `obseq` (observational equivalence), `trace` (trace equivalence, also called may equivalence), and `must` (must equivalence). The `le` command is used in exactly the same fashion to compute preorders. The currently supported arguments to its -S flag are `may` and `must` for the may and must preorders respectively.

4.1 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 [7]. The information 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.

4.1.1 Bisimulation Equivalence

When two systems are not bisimulation equivalent, the CWB-NC returns a Hennessy-Milner Logic (HML) formula [16] satisfied by one system but not the other. The syntax of HML formulas is defined by the following grammar, where a is an action.

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

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.

4.1.2 Observational Equivalence

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

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

We say that q is a weak a-derivative of p if p can engage in a sequence of transitions, one of them labeled by a and the rest internal, leading to 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. Recall that `t` is the CWB-NC CCS notation for the internal action τ.

```
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)
```

4.1.3 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)
```

4.1.4 Must Equivalence (Preorder)

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

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

A system must pass the test *actSequence* (*acceptanceSet*) if in every state the system reaches after performing the sequence of actions in *actSequence*, some action in *acceptanceSet* can be performed. A system must pass the test *actSequence* `CONV` if in state the system reaches after performing the sequence of actions in *actSequence* is convergent. (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 5

Temporal-Logic-Based Verification

The CWB-NC includes facilities for determining whether or not a system satisfies properties expressed in two temporal logics: an enriched form of the modal mu-calculus [18, 13] that includes the operators of Computation Tree Logic (CTL) [6], and Generalized CTL* (GCTL*). To use the model checkers for these logics, one first creates a file containing definitions of properties of interest. mu-calculus formulas, the file name must end in the `.mu` suffix; for GCTL*, the file name must end in the the `.gctl` suffix. (Note that one cannot include both mu-calculus and GCTL* in the same file.) One then loads this file and the file describing the system description into a CWB-NC session with the `load` command and invokes the model checker with the `chk` command. The `-L` flag to this command specifies the logic the formulas are written in: `chk -L mu`, which is the default, invokes a mu-calculus checker, while `check -L gctl` is uses for GCTL* formulas.

The rest of this chapter describes the logics in more detail. For each we give a Yacc-like grammar, based on the input to the PAC-NC front-end generator [22, 10], that defines the concrete syntax of the logic. We also give an informal semantics as well as examples illustrating how properties may be formulated.

5.1 Mu-Calculus Model Checking

To use the mu-calculus model checker, one creates a `.mu` file containing formulas of interest, loads it, and then runs the model checker using the `chk -L mu` command. The `-L mu` may be omitted.

5.1.1 Syntax of the `.mu` File

A `.mu` file consists of a list of white-space-separated proposition declarations; Figure 5.1 makes the syntax precise. `prop` *id* = *proposition* binds proposition *prop* to a proposition variable *id*. The propositions true and false (**tt** and **ff**) are atomic. Proposition variables may appear within propositions, although circular dependencies are disallowed. 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, which is a possibly empty, comma-separated list of actions that may be preceded by a `-`, the set complementation operator. Note that the empty list denotes the empty set of actions; consequently, the notation `-` (complemented empty list) stands for the set of all actions. 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 5.1.

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 5.1: Syntax of a `.mu` file

Semantically, mu-calculus formulas are interpreted with respect to system states. Every state satisfies `tt`, while no state satisfies `ff`. Formula ϕ_1 `\/` ϕ_2 (ϕ_1 `/\` ϕ_2) holds of a state if one of (both of) ϕ_1 and ϕ_2 do. The <*act_set*> and [*act_set*] operators are modalities. A state satisfies <*act_set*>ϕ if there exists a transition from the state and labeled by an action from *act_set* that leads to a state satisfying ϕ, while it satisfies [*act_set*]ϕ if every transition emanating from the state and labeled by an element of *act_set* satisfies ϕ. The `min` and `max` operators allow formulas to be defined recursively, with `min X=`ϕ and `max X=`ϕ representing the "least permissive" and "most permissive" solutions, respectively, to the "equation" $X = \phi$.

5.1.2 The CTL Encoding

The meaning of each CTL operator is defined in terms of the *computations* of an LTS, which are the maximal execution sequences in which the LTS is capable of engaging. The elements of a computation sequence are states that an LTS passes through during execution. Let $\mathcal{P} = \langle P, p_{start}, \longrightarrow \rangle$ be an LTS. A computation of a state $p \in P$ is a sequence of states $p_0, p_1, ..., p_n$ 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 state the property being formulated should hold for. If quantifier is **A**, then the property should hold for all computations of the state, while if it is **E**, then there should exist a computation of the state for which the property holds.

The second component of a CTL operator is a *path modality* describing a property of of computations, which are (maximal) execution sequences. Let $C = p_0, p_1, ..., p_n$ be a computation.

- *Always.*
  G Φ holds of C if Φ holds for all $p_i, 0 \leq i \leq n$.
- *Eventually.*
  F Φ holds of C if $\exists i, 0 \leq i \leq n$ and p_i satisfies Φ.
- *Until.*
  Φ_1 **U** Φ_2 holds of C if $\exists i : 0 \leq i \leq n : p_i$ satisfies Φ_2 and $\forall j : 0 \leq j < i : p_j$ satisfies Φ_1.
- *Weak Until.*
  Φ_1 **W** Φ_2 holds of C if either Φ_1 **U** Φ_2 does, or ($\forall i : 0 \leq i \leq n : p_i$ satisfies Φ_1).

The two components of a CTL formula combine to allow one to express properties of the computations of an LTS. For example, **AG** Φ holds of an LTS if for all computations its start state G Φ holds.

Table 5.1 describes the translation of CTL formulas into equivalent mu-calculus formulas. The treatment of finite (terminating) behaviors should be noted. Also recall that the set indicated in the modal operators (`-`) is the complement of the empty set, which of course is the set *Act* of all actions.

CTL Formula	*Mu-Calculus Translation*
AG Φ	**max** X = Φ `/\ [-]X`
AF Φ	**min** X = Φ `\/ ( [-]X /\ <->tt)`
A (Φ_1 **U** Φ_2)	**min** X = Φ_2 `\/` (Φ_1 `/\ [-]X /\ <->tt`)
A (Φ_1 **W** Φ_2)	**max** X = Φ_2 `\/` (Φ_1 `/\ [-]X`)
EG Φ	**max** X = Φ `/\ (<->X \/ [-]ff)`
EF Φ	**min** X = Φ `\/ <->X`
E (Φ_1 **U** Φ_2)	**min** X = Φ_2 `\/` (Φ_1 `/\ <->X`)
E (Φ_1 **W** Φ_2)	**max** X = Φ_2 `\/` (Φ_1 `/\ (<->X \/ [-]ff)`)

Table 5.1: Mu-calculus translation of the CTL operators

5.2 Examples

Figure 5.2 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.

5.3 GCTL* Model Checking

GCTL* constitutes an extension of the traditional temporal logic CTL* [12] that is tailored for reasoning about systems whose transitions are labeled by actions. Formulas in the logic are often easier to understand than mu-calculus formulas, especially when one is interested in expressing fairness constraints. In this section we introduce GCTL* and review the CWB-NC's concrete syntax for the logic.

5.3.1 The GCTL* Logic

*Generalized CTL** (GCTL*) extends CTL* by allowing formulas to constrain actions as well as states. The high-level syntax for the logic is given below, where p is an atomic state proposition and and θ is an atomic action proposition.

$$\begin{array}{lcl} \mathcal{S} & ::= & p \mid \neg p \mid \mathcal{S} \wedge \mathcal{S} \mid \mathcal{S} \wedge \mathcal{S} \mid \mathbf{A}\mathcal{P} \mid \mathbf{E}\mathcal{P} \\ \mathcal{P} & ::= & \theta \mid \neg\theta \mid \mathcal{S} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathcal{P} \wedge \mathcal{P} \mid \mathbf{X}\mathcal{P} \mid \mathcal{P}\mathbf{U}\mathcal{P} \mid \mathcal{P}\mathbf{R}\mathcal{P} \end{array}$$

```
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 5.2: Example formulas

The formulas generated by $\mathcal{S}$ are called state formulas, while those generated by $\mathcal{P}$ are called path formulas. The state formulas constitute the formulas of GCTL* . In what follows we use $\psi, \psi', \psi_1, \ldots$ to range over state formulas and $\phi, \phi', \phi_1, \ldots$ to range over path formulas.

Semantically, the logic departs from traditional CTL* in two respects. Firstly, the paths that path formulas are interpreted over have the form $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots$, where the s_i are system states and the a_i are actions, and thus contain actions as well as states. Secondly, as systems may contain deadlocked states some provision must be made for finite maximal paths as models. The GCTL* semantics follows a standard convention in temporal logic by allowing the last state in a finite maximal path to "loop" to itself; the action component of these implicit transitions is assumed to violate all atomic action propositions θ.

The intuitive semantics of the operators are as follows. A state satisfies an atomic proposition p if the interpretation function associated to atomic propositions indicates this is the case, while the state satisfies $\neg p$ if it does not satisfy p. The operators $\wedge$ and $\vee$ represent conjunction and disjunction, respectively; a state satisfies $\psi_1 \wedge \psi_2$ ($\psi_1 \vee \psi_2$) if it satisfies both of (one of) ψ_1 and ψ_2. Finally, a state satisfies $\mathbf{A}\phi$ ($\mathbf{E}\phi$) if every execution (some execution) emanating from the state satisfies ϕ. Regarding path formulas, an execution satisfies a state formula if the initial state in the execution does, and it satisfies θ if the execution contains at least on transition and the label of the first transition on the path satisfies θ. A path satisfies $\neg\theta$ if either the first transition on the path is labeled by an action not satisfying θ or the path has no transitions. $\mathbf{X}$ represents the "next-time operator" and has the usual semantics when the path is not deadlocked; a path $s_0 \xrightarrow{a_1} s_1 \ldots$ satisfies $\mathbf{X}\phi$ if the path $s_1 \ldots$ satisfies ϕ. A deadlocked path of form s satisfies $\mathbf{X}\Phi$ if s satisfies Φ. $\phi_1\mathbf{U}\phi_2$ holds of a path if ϕ_1 remains true until ϕ_2 becomes true. The constructor $\mathbf{R}$ may be thought of as a "release operator"; a path satisfies $\phi_1\mathbf{R}\phi_2$ if ϕ_2 remains true until ϕ_1 "releases" the path from the obligation. This operator is the dual of the until operator.

Other path operators may be defined in terms of the existing ones. The "always" operator, $\mathbf{G}\phi$, may be rendered as $\mathbf{ff}\mathbf{R}\phi$, the "eventually" operator, $\mathbf{F}\phi$, as $\mathbf{tt}\mathbf{U}\phi$, and the weak until operator, $\phi_1\mathbf{W}\phi_2$, as $\phi_2\mathbf{R}(\phi_1 \vee \phi_2)$.

5.3.2 The Syntax of the `.gctl` File

As was the case with `.mu` files, `.gctl` files consist of a sequences of proposition declarations of the form `prop` *id* `=` *prop*. Figure 5.3 contains the specification of the concrete syntax for GCTL* . It should be noted that the operators `A`, `E`, `F`, etc. are keywords, and that successive keywords (as in the formula `A G tt`) must be separated by spaces in order for the parser to handle them.

The nonterminal *prop* corresponds to state formulas. The only atomic state propositions are `tt` (true) and **ff** (false). State formulas may then be built up using negation (`not`), disjunction (`\/`) conjunction (`/\`), and implication (`->`). The modal operators <*act_set*> and [*act_set*] are derived; we return to this point later. Finally, the path quantifiers `A` and `E` may be used to "lift" path formulas to state formulas.

Path formulas are generated from the nonterminal *path*. Every state formula is a path formula; in addition, *atomic action formulas* are also path formulas. In the CWB-NC implementation atomic action formulas have form { *act_list* } or {- *act_list* }; the former is satisfied by an action a if a appears in the given list, while the second is satisfied by an action a if the action does not appear in the given list. A path satisfies one of these atomic action formulas if the path contains at least one transition, and the action labeling the first transition satisfies the action formula. For implementation reasons negation can only be applied to atomic action path formulas. The notation for such a negated formula is obtained by prefixing a tilde, `~`, to an atomic action formula. It should be noted that `~` and `-` are different. For example, a path satisfies `~{a,b}` iff either the path contains no transition (i.e. the first state is deadlocked) or if the first transition is neither `a` or `b`. On the other hand, a path satisfies `{-a,b}` iff the path has a transition, and the first transition is neither `a` or `b`. The other path operators have the obvious interpretations as specified above.

We now return to explaining how <*act_set*> and [*act_set*] may be derived. The translation is as follows.

<*act_set*>ψ = `E` ({ *act_set* } `/\ X` ψ)
[*act_set*]ψ = `A` ({ *act_set* } `-> X` ψ)

Because GCTL* includes the `->` operator, the diamond modality `<- >` must include a space between the `-` and `>`.

5.3.3 Sample GCTL* Formulas

What follows are a collection of sample GCTL* formulas.

```
prop can_deadlock =
    E F ~{- }

prop recv_guarantee =
    A G ({send} ->  F {'receive})

prop fair_recv_guarantee =
    A ((G F {-t}) -> (G {send} -> F {'receive}))
```

Formula `recv_guarantee` stipulates that along every path, whenever a `send` occurs a `'receive` action eventually follows. Formula `fair_recv_guarantee` weakens this property by only requiring it to hold along paths that do not contain an infinite number of internal actions. One may show that the former property does not hold of the lossy ABP example given in Section 3.2.2 of Chapter 3, while the latter does.

Keywords

A, E, F, G, R, U, W, X, ff, not, prop, tt

Tokens

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

Grammar

prop	:	`"tt"` | `"ff"` | *ident* | `"not"` *prop* | *prop* `"\/"` *prop* | *prop* `"/\"` *prop* | *prop* `"->"` *prop* | `"<"` *act_set* `">"` *prop* | `"["` *act_set* `"]"` *prop* | `"A"` *path* | `"E"` *path* | `"("` *prop* `")"`
path	:	*prop* | `"{"` *act_set* `"}"` | `"~"` `"{"` *act_set* `"}"` | `"G"` *path* | `"F"` *path* | `"X"` *path* | *path* `"U"` *path* | *path* `"W"` *path* | *path* `"R"` *path* | *path* `"\/"` *path* | *path* `"/\"` *path* | *path* `"->"` *path* | `"("` *path* `")"`
act	:	ACTSTRING | IDENT
act_set	:	*act_list* | `"-"` *act_list*
prop_decl	:	`"prop"` *ident* `"="` *prop*
ident	:	IDENT

Lists

List name	**May be Empty?**	**Open Delimiter**	**Separator**	**Close Delimiter**	**Items**
act_list	yes	`""`	`","`	`""`	act
prop_decl_list	no	`""`	`""`	`""`	prop_decl

Precedences

`"="` < `"\/"` < `"/\"` < `"not"`, `"G"`, `"F"`, `"A"`, `"E"`, `"X"` <

`"U"`, `"R"`, `"W"` < `"]"`, `"["`, `">"`, `"<"` < `"]]"`, `"[["`, `">>"`, `"<<"`

Figure 5.3: Syntax of a `.gctl` file.

Chapter 6

Other Analysis Capabilities

6.1 Interactive Simulation of Systems

A facility for performing user-directed or random simulation of systems has also been added to the CWB-NC and has proven useful in debugging system designs. In user-directed mode the simulator displays the execution steps the system is capable of taking and queries the user to select one. The resulting system state and possible transitions are then displayed and the user may select the ensuing execution step. Alternatively the user may indicate that a given number of random execution steps be taken. At any point the cumulative execution sequence may be displayed and the user may specify that the system state be reset to any point in the trace.

6.2 Reachability Analysis

The CWB-NC `search` command provides a general reachability analysis capability. Given a system and a mu-calculus formula the command searches the state space of the system for a state satisfying the formula. The command will most often be used with simple formulas containing no fixpoint operators making the check at each state very simple and therefore the exploration of the state space fast. If a state is found satisfying the given formula, the simulator is invoked with the path to the state pre-loaded (i.e. the current simulator state is the state satisfying the formula). Many properties (such as deadlock or mutual exclusion violation) may be checked in this fashion.

Chapter 7

Design Language Syntax and Semantics

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, Timed CCS, SCCS, CSP, and Basic LOTOS. In this chapter 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. In the following we use the words *agent* and *process* interchangeably to refer to any entity that may act independently and interact with its environment. Such entities are the building blocks for the concurrent systems the CWB-NC is meant to analyze.

7.1 Notation for Describing Design Languages

For each language, we give a Yacc-like grammar to define the syntax and provide a set of Plotkin-style SOS rules [25] 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 [22, 10] 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.

$$\textbf{name}\,\frac{premises}{conclusion}(side\ condition)$$

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.

$$\textbf{parallel}_3 \; \frac{ae,\ se\ :\ p_1 \xrightarrow{a} p_1',\ ae,\ se\ :\ p_2 \xrightarrow{b} p_2'}{ae,\ se\ :\ p_1|p_2 \xrightarrow{\mathtt{t}} p_1'|p_2'} (inverses(a,b))$$

The rule states that if process p_1 can engage in an action a and evolve to p_1' and process p_2 can engage in b and evolve to p_2', and a and b are inverses (i.e. constitute an input/output pair on the same communication channel), then $p_1|p_2$ can execute an internal action, `t` (τ), corresponding to the synchronized execution of a and b. Here ae and se are agent and set environments used to resolve free variables in p_1 and p_2.

In each design language supported by the CWB-NC, the base language is extended to allow the use of automata to model the behavior of a process in the obvious fashion. Each state in the automaton represents a state of the modeled process. In a given state an automaton may perform any of the actions corresponding to edges leaving the state and when it does so the label of the edge indicates the action performed and the current state of the automaton becomes the target state of the edge. In the SOS rules for each language the transitions relation for automata is denoted $aut \; > \xrightarrow{a} aut'$ to indicate that the automaton aut may perform the action a and evolve to the automaton aut'.

7.2 CCS

For an introduction to CCS we recommend [4] and [20].

7.2.1 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 specification is defined in a `.ccs` suffixed file consisting of a sequence of declarations that define processes and identifier sets. The keyword `proc` precedes a process declaration.

$$\mathtt{proc}\ procName = ccsTerm$$

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.

$$\mathtt{set}\ setName = idSet$$

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 7.1 defines the full syntax of a `.ccs` file.

tokens

`[1-9][0-9]{0,8} | 0` => INT

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

grammar

spec	:	*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*
id_set	:	`"{"` *id_list* `"}"`
relabeling	:	*rename_list*
rename	:	*id* `"/"` *id*
agent_frame	:	*agent_binding_list*
agent_binding	:	*id* `"="` *agent*
automaton	:	`"Aut"` `"("` `"start"` `"="` *int* `","` *state_list* `")"`
state	:	*int* `":"` *trans_list*
trans	:	*act* *int_list*
id	:	ID
int	:	INT

lists

List name	May be Empty?	Open Delimiter	Separator	Close Delimiter	Items
binding_list	no	""	""	""	binding
agent_binding_list	no	""	"and"	""	agent_binding
rename_list	no	""	","	""	rename
id_list	no	""	","	""	id
state_list	no	""	""	""	state
trans_list	yes	""	""	""	trans
int_list	yes	"{"	","	"}"	int

priorities
`"where"` $<$ `"+"` $<$ `"|"` $<$ `"\"` $<$ `"."` $<$ `"["` $<$ `"nil","@"`

Figure 7.1: Syntax of CCS

7.2.2 CCS Semantics

We first informally describe the behavior of each CCS operator and then give the formal semantics of CCS with the ensuing set of SOS rules. The terminated process `nil` is incapable of performing any further actions. If a set of process declarations includes **proc** $X = p$, then the process variable X behaves like p. The process $a.p$ may perform the action a and subsequently behave as p. The process $p_1 + p_2$ may behave either as p_1 or as p_2. The term $p_1|p_2$ indicates that p_1 is run in parallel with p_2, so the two processes may communicate with each other (one may receive a message the other sends over a channel) or act independently. The process $p\backslash L$ executes as p, but is incapable of performing any action whose channel component is included in the set of channel names L. Finally, $p[r]$ behaves as p but with its observable actions renamed by the relabeling function r. A sequence of local process declarations may be given with the **where-end** notation.

The operational semantics of CCS is shown in Figure 7.2. The rules define the relation:

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

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 $ae,\ se\ :\ p \xrightarrow{a} p'$ 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." The auxiliary predicates and functions used in the rules are described in Table 7.1.

Table 7.1: Functions and predicates used in the SOS for CCS.

inverses : $act \times act \rightarrow bool$	Inverses is true for two actions if they are an input and an output over the same channel. $$\text{inverses}(a,b) = \begin{cases} \text{true} & \text{if } \exists \text{ channel } c \text{ such that} \\ & (a = c \text{ and } b = {}'c) \text{ or} \\ & (a = {}'c \text{ and } b = c) \\ \text{false} & \text{otherwise} \end{cases}$$
labelof : $act \rightarrow id$	Labelof returns the channel component of a given action.
$\notin$: $'a \times ('a\ set) \rightarrow boolean$	$\notin$ takes an item and a set of items and returns true when the item is not in the set.
$r : id \rightarrow id, \;\; \hat{r} : act \rightarrow act$	A relabeling function r is defined by a list of renamings of the form id_2/id_1, meaning id_1 is mapped to id_2. r is lifted to a function over actions ($\hat{r} : act \rightarrow act$) as follows. $$\hat{r}(a) = \begin{cases} r(\text{labelof}(a)) & \text{if a is an input action} \\ {}'(r(\text{labelof}(a))) & \text{if a is an output action} \\ \texttt{t} & \text{if } a = \texttt{t} \end{cases}$$ For example, $(\texttt{b.nil})[\texttt{a/b}] \xrightarrow{\texttt{a}} \texttt{nil}$.
not_relab : $agent \rightarrow bool$	Not_relab is true of any agent whose top most operator is not relabeling, i.e. any agent not of the form $p[r]$.
lookup : $id \times ('a\ env) \rightarrow\ 'a$	Lookup returns the value bound to an id in a given environment.
push_frame : $('a\ frame) \times$ $('a\ env) \rightarrow ('a\ env)$	Push_frame inserts a collection of bindings, called a frame, into a given environment.

vars

a, b : act
$p, p', p_1, p_1', p_2, p_2'$: agent
s : id_set
r, r_1, r_2 : relabeling
ae : (agent env)
se : (id_set env)
A, s_id : id
ag_frame : (agent frame)
aut, aut' : automaton

rules

$$\textbf{prefix}\ \frac{}{ae,\ se\ :\ a.p \xrightarrow{a} p} \qquad \textbf{sum}_1\ \frac{ae,\ se\ :\ p_1 \xrightarrow{a} p_1'}{ae,\ se\ :\ p_1 + p_2 \xrightarrow{a} p_1'} \qquad \textbf{sum}_2\ \frac{ae,\ se\ :\ p_2 \xrightarrow{a} p_2'}{ae,\ se\ :\ p_1 + p_2 \xrightarrow{a} p_2'}$$

$$\textbf{par}_1\ \frac{ae,\ se\ :\ p_1 \xrightarrow{a} p_1'}{ae,\ se\ :\ p_1|p_2 \xrightarrow{a} p_1'|p_2} \qquad \textbf{par}_2\ \frac{ae,\ se\ :\ p_2 \xrightarrow{a} p_2'}{ae,\ se\ :\ p_1|p_2 \xrightarrow{a} p_1|p_2'}$$

$$\textbf{par}_3\ \frac{ae,\ se\ :\ p_1 \xrightarrow{a} p_1',\quad ae,\ se\ :\ p_2 \xrightarrow{b} p_2'}{ae,\ se\ :\ p_1|p_2 \xrightarrow{\mathtt{t}} p_1'|p_2'}(\text{inverses}(a,b))$$

$$\textbf{rel}_1\ \frac{ae,\ se\ :\ p[r_2 \circ r_1] \xrightarrow{a} p'}{ae,\ se\ :\ p[r_1][r_2] \xrightarrow{a} p'} \qquad \textbf{rel}_2\ \frac{ae,\ se\ :\ p \xrightarrow{a} p'}{ae,\ se\ :\ p[r] \xrightarrow{\hat{r}(a)} p'[r]}(\text{not\_relab}(p))$$

$$\textbf{res}_1\ \frac{ae,\ se\ :\ p \xrightarrow{a} p'}{ae,\ se\ :\ p \setminus s \xrightarrow{a} p' \setminus s}(\text{labelof}(a)\ \notin\ s)$$

$$\textbf{res}_2\ \frac{ae,\ se\ :\ p \xrightarrow{a} p'}{ae,\ se\ :\ p \setminus s\_id \xrightarrow{a} p' \setminus s\_id}(\text{labelof}(a)\ \notin\ \text{lookup}(s\_id, se))$$

$$\textbf{aut}\ \frac{aut\ \ >\!\xrightarrow{a} aut'}{ae,\ se\ :\ aut \xrightarrow{a} aut'} \qquad \textbf{ag\_var}\ \frac{ae,\ se\ :\ \text{lookup}(A, ae) \xrightarrow{a} p'}{ae,\ se\ :\ A \xrightarrow{a} p'}$$

$$\textbf{local\_agents}\ \frac{\text{push\_frame}(ag\_frame, ae),\ se\ :\ p \xrightarrow{a} p'}{ae,\ se\ :\ p\ \texttt{where}\ ag\_frame\ \texttt{end} \xrightarrow{a} p'\ \texttt{where}\ ag\_frame\ \texttt{end}}$$

Figure 7.2: Semantics of CCS

7.3 Prioritized CCS

For a description of this version of CCS with priorities, which we call PCCS, see [8, 9]. PCCS extends CCS by allowing different priority values to be assigned 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 $a : n$.

7.3.1 Syntax of Prioritized CCS

The syntax of a .pccs file is basically the same as that for a .ccs file as described in Section 7.2.1, namely it consists 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 main difference is the syntax of actions, which includes a natural number indicating priority. The grammar shown in Figure 7.3 defines the full syntax of a `.pccs` file.

7.3.2 Semantics of Prioritized CCS

The semantics of each operator in PCCS is similar to its CCS counterpart, but with priority taken into account. For example, when two processes are running in parallel a high priority synchronization in one component may preempt a lower priority action in the other. Similarly a high priority synchronization between the two components may preempt actions in both components. Note however that only internal actions (i.e. $\tau : n$, written as `t:n` in our syntax) have preemptive power. The operational semantics of PCCS is shown in Figures 7.4,7.5, and 7.6. The rules define two relations, a *transitions* relation and an auxiliary relation called *inits* which is used in defining the *transitions* relation.

As was the case for CCS, the rules define the relation:

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

The first two components are environments that map identifiers to agents and sets respectively. If $\langle ae, se, p, a, p' \rangle \in transitions$ then we write $ae,\ se\ :\ p \xrightarrow{a} p'$ 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." Defining the PCCS *transitions* relation requires a second relation:

$$inits \subseteq agent\ env \times set\ env \times int \times agent \times act.$$

If $\langle ae, se, n, p, a \rangle \in inits$ then we write:

$$ae,\ se,\ n\ :\ p \xrightarrow{a}$$

Intuitively the *inits* relation determines the initial actions of priority higher than n that an agent is capable of engaging in if preemptions are disregarded.

The SOS rules are very similar to those for CCS except that the side conditions in the rules for summation and parallel composition encode the semantics of preemption. Each of these side conditions has the form $\mathtt{t} \notin \mathrm{inits}(ae, se, prio(a), p)$ which means "p is not capable of performing an internal action having a priority greater than the priority of the action a."

Table 7.2: Functions and predicates used in the SOS for PCCS (augmenting Table 7.1).

prio : $act \to int$	Prio returns the priority level associated with an action, i.e. $\text{prio}(a : n) = n$.
inverses : $act \times act \to bool$	Inverses is true for two actions if they are an input and an output over the same channel and have the same priority value. $$\text{inverses}(a : n, b : m) = \begin{cases} \text{true} & \text{if } n = m \text{ and} \\ & \exists \text{ channel } c \text{ such that} \\ & (a = c \text{ and } b = {'c}) \text{ or} \\ & (a = {'c} \text{ and } b = c) \\ \text{false} & \text{otherwise} \end{cases}$$
polls : $act \times act \to bool$	a polls b is true if a is a polling action and b is an output action over the same channel and they have the same priority level. $$a : n \text{ polls } b : m = \begin{cases} \text{true} & \text{if } n = m \text{ and} \\ & \exists \text{ channel } c \text{ such that} \\ & (a = \hat{}c \text{ and } b = {'c}) \\ \text{false} & \text{otherwise} \end{cases}$$
$<: int \times int \to bool$	Integer comparison. Recall that since increasing priority values means decreasing priority, if $\text{prio}(a) < k$ then action a has a higher priority than those at level k.

Two additional operators (not directly related to priority) called polling and disabling are also included in this version of PCCS. A polling action notated $\hat{}c$ allows an agent to poll its environment to determine if an output over the channel c is immediately available. The disabling operator works as follows. The agent $p_1[\!>p_2$ behaves like p_1 until an event from p_2 occurs thereby disabling p_1 and subsequently behaving as p_2. The SOS rules employ the auxiliary functions and predicates defined for plain CCS in Table 7.1 as well as those shown in Table 7.2.

tokens

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

grammar

spec	:	*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	:	"t" | *id* | "'" *id* | "^" *id*
restriction	:	*id_set* | *id*
relabeling	:	*rename_list*
rename	:	*id* "/" *id*
id_set	:	"{" *id_list* "}"
id_set2	:	*id_list*
agent_frame	:	*agent_binding_list*
agent_binding	:	*id* "=" *agent*
automaton	:	"Aut" "(" "start" "=" *int* "," *state_list* ")"
state	:	*int* ":" *trans_list*
trans	:	*act int_list*
id	:	ID
int	:	INT

lists

List name	May be Empty?	Open Delimiter	Separator	Close Delimiter	Items
binding_list	no	""	""	""	binding
agent_binding_list	no	""	"and"	""	agent_binding
rename_list	no	""	","	""	rename
id_list	no	""	","	""	id
state_list	no	""	""	""	state
trans_list	yes	""	""	""	trans
int_list	yes	"{"	","	"}"	int

priorities

"[>" $<$ "|" $<$ "\" $<$ "[","]","{","}","where" $<$ "+" $<$ "." $<$ "nil","@" $<$ ":"

Figure 7.3: Syntax of PCCS

vars

a, b, c : act
$p, p', p_1, p_1', p_2, p_2'$: agent
s : id_set
r, r_1, r_2 : relabeling
ae : (agent env)
se : (id_set env)
A : id
s_id : id
ag_frame : (agent frame)
aut, aut' : automaton

rules

$$\textbf{prefix}\ \frac{}{ae,\ se\ :\ a.p \xrightarrow{a} p} \qquad \textbf{sum}_1\ \frac{ae,\ se\ :\ p_1 \xrightarrow{a} p_1'}{ae,\ se\ :\ p_1 + p_2 \xrightarrow{a} p_1'}(\mathtt{t} \notin \mathrm{inits}(ae, se, \mathrm{prio}(a), p_2))$$

$$\textbf{sum}_2\ \frac{ae,\ se\ :\ p_2 \xrightarrow{a} p_2'}{ae,\ se\ :\ p_1 + p_2 \xrightarrow{a} p_2'}(\mathtt{t} \notin \mathrm{inits}(ae, se, \mathrm{prio}(a), p_1))$$

$$\textbf{par}_1\ \frac{ae,\ se\ :\ p_1 \xrightarrow{a} p_1'}{ae,\ se\ :\ p_1|p_2 \xrightarrow{a} p_1'|p_2}(\mathtt{t} \notin \mathrm{inits}(ae, se, \mathrm{prio}(a), p_1|p_2))$$

$$\textbf{par}_2\ \frac{ae,\ se\ :\ p_2 \xrightarrow{a} p_2'}{ae,\ se\ :\ p_1|p_2 \xrightarrow{a} p_1|p_2'}(\mathtt{t} \notin \mathrm{inits}(ae, se, \mathrm{prio}(a), p_1|p_2))$$

$$\textbf{par}_3\ \frac{ae,\ se\ :\ p_1 \xrightarrow{a} p_1',\quad ae,\ se\ :\ p_2 \xrightarrow{b} p_2'}{ae,\ se\ :\ p_1|p_2 \xrightarrow{\mathtt{t}\,:\,\mathrm{prio}(b)} p_1'|p_2'}(\mathrm{inverses}(a, b) \text{ and } \mathtt{t} \notin \mathrm{inits}(ae, se, \mathrm{prio}(a), p_1|p_2))$$

$$\textbf{par}_4\ \frac{ae,\ se\ :\ p_1 \xrightarrow{a} p_1',\quad ae,\ se\ :\ p_2 \xrightarrow{b} p_2'}{ae,\ se\ :\ p_1|p_2 \xrightarrow{\mathtt{t}\,:\,\mathrm{prio}(b)} p_1'|p_2}(a \text{ polls } b \text{ and } \mathtt{t} \notin \mathrm{inits}(ae, se, \mathrm{prio}(b), p_1|p_2))$$

$$\textbf{par}_5\ \frac{ae,\ se\ :\ p_1 \xrightarrow{a} p_1',\quad ae,\ se\ :\ p_2 \xrightarrow{b} p_2'}{ae,\ se\ :\ p_1|p_2 \xrightarrow{\mathtt{t}\,:\,\mathtt{prio}(a)} p_1|p_2'}(b \text{ polls } a \text{ and } \mathtt{t} \notin \mathrm{inits}(ae, se, \mathrm{prio}(a), p_1|p_2))$$

$$\textbf{rel}_1\ \frac{ae,\ se\ :\ p[r_2 \circ r_1] \xrightarrow{a} p'}{ae,\ se\ :\ p[r_1][r_2] \xrightarrow{a} p'} \qquad \textbf{rel}_2\ \frac{ae,\ se\ :\ p \xrightarrow{a} p'}{ae,\ se\ :\ p[r] \xrightarrow{\hat{r}(a)} p'[r]}(\mathrm{not\_relab}(p))$$

Figure 7.4: Semantics of PCCS - transitions relation, part one

$$\textbf{disab}_1 \; \frac{ae,\ se\ :\ p_1 \xrightarrow{a}\!\triangleright p_1'}{ae,\ se\ :\ p_1[\!>\!p_2 \xrightarrow{a}\!\triangleright p_1'[\!>\!p_2} (\mathtt{t} \notin \mathrm{inits}(ae, se, \mathrm{prio}(a), p_2))$$

$$\textbf{disab}_2 \; \frac{ae,\ se\ :\ p_2 \xrightarrow{a}\!\triangleright p_2'}{ae,\ se\ :\ p_1[\!>\!p_2 \xrightarrow{a}\!\triangleright p_2'} (\mathtt{t} \notin \mathrm{inits}(ae, se, \mathrm{prio}(a), p_1))$$

$$\textbf{res}_1 \; \frac{ae,\ se\ :\ p \xrightarrow{a}\!\triangleright p'}{ae,\ se\ :\ p \setminus s \xrightarrow{a}\!\triangleright p' \setminus s} (\mathrm{labelof}(a) \notin s)$$

$$\textbf{res}_2 \; \frac{ae,\ se\ :\ p \xrightarrow{a}\!\triangleright p'}{ae,\ se\ :\ p \setminus s\_id \xrightarrow{a}\!\triangleright p' \setminus s\_id} (\mathrm{labelof}(a) \notin \mathrm{lookup}(s\_id, se))$$

$$\textbf{aut} \; \frac{aut\ \ >\!\xrightarrow{a}\!\triangleright aut'}{ae,\ se\ :\ aut \xrightarrow{a}\!\triangleright aut'} \qquad \textbf{ag\_var} \; \frac{ae,\ se\ :\ \mathrm{lookup}(A, ae) \xrightarrow{a}\!\triangleright p'}{ae,\ se\ :\ A \xrightarrow{a}\!\triangleright p'}$$

$$\textbf{local\_agents} \; \frac{\mathrm{push\_frame}(ag\_frame, ae),\ se\ :\ p \xrightarrow{a}\!\triangleright p'}{ae,\ se\ :\ p\ \mathtt{where}\ ag\_frame\ \mathtt{end} \xrightarrow{a}\!\triangleright p'\ \mathtt{where}\ ag\_frame\ \mathtt{end}}$$

Figure 7.5: Semantics of PCCS - transitions relation, part two

vars

a, b : act
p, p_1, p_2 : agent
s : id_set
r : relabeling
ae : (agent env)
se : (id_set env)
A, s_id : id
j, k : int
ag_frame : (agent frame)
aut : automaton

rules

$$\textbf{prefix}\ \frac{}{ae,\ se,\ k:\ a.p \xrightarrow{a}}(prio(a) < k) \qquad \textbf{sum}_1\ \frac{ae,\ se,\ k:\ p_1 \xrightarrow{a}}{ae,\ se,\ k:\ p_1 + p_2 \xrightarrow{a}}$$

$$\textbf{sum}_2\ \frac{ae,\ se,\ k:\ p_2 \xrightarrow{a}}{ae,\ se,\ k:\ p_1 + p_2 \xrightarrow{a}} \qquad \textbf{par}_1\ \frac{ae,\ se,\ k:\ p_1 \xrightarrow{a}}{ae,\ se,\ k:\ p_1|p_2 \xrightarrow{a}} \qquad \textbf{par}_2\ \frac{ae,\ se,\ k:\ p_2 \xrightarrow{a}}{ae,\ se,\ k:\ p_1|p_2 \xrightarrow{a}}$$

$$\textbf{par}_3\ \frac{ae,\ se,\ k:\ p_1 \xrightarrow{a},\quad ae,\ se,\ k:\ p_2 \xrightarrow{b}}{ae,\ se,\ k:\ p_1|p_2 \xrightarrow{\texttt{t}\,:\,\mathrm{prio}(b)}}(\mathrm{inverses}(a,b)) \qquad \textbf{disab}_1\ \frac{ae,\ se,\ k:\ p_1 \xrightarrow{a}}{ae,\ se,\ k:\ p_1[>p_2 \xrightarrow{a}}$$

$$\textbf{par}_4\ \frac{ae,\ se,\ k:\ p_1 \xrightarrow{a},\quad ae,\ se,\ k:\ p_2 \xrightarrow{b}}{ae,\ se,\ k:\ p_1|p_2 \xrightarrow{\texttt{t}\,:\,prio(a)}}(a \text{ polls } b) \qquad \textbf{disab}_2\ \frac{ae,\ se,\ k:\ p_2 \xrightarrow{a}}{ae,\ se,\ k:\ p_1[>p_2 \xrightarrow{a}}$$

$$\textbf{par}_5\ \frac{ae,\ se,\ k:\ p_1 \xrightarrow{a},\quad ae,\ se,\ k:\ p_2 \xrightarrow{b}}{ae,\ se,\ k:\ p_1|p_2 \xrightarrow{\texttt{t}\,:\,prio(b)}}(b \text{ polls } a) \qquad \textbf{rel}\ \frac{ae,\ se,\ k:\ p \xrightarrow{a}}{ae,\ se,\ k:\ p[r] \xrightarrow{\hat{r}(a)}}$$

$$\textbf{res}_1\ \frac{ae,\ se,\ k:\ p \xrightarrow{a}}{ae,\ se,\ k:\ p \setminus s \xrightarrow{a}}(\mathrm{labelof}(a) \notin s) \qquad \textbf{res}_2\ \frac{ae,\ se,\ k:\ p \xrightarrow{a}}{ae,\ se,\ k:\ p \setminus s \xrightarrow{a}}(\mathrm{labelof}(a) \notin \mathrm{lookup}(\mathrm{s\_id}, \mathrm{se}))$$

$$\textbf{aut}\ \frac{aut\ > \xrightarrow{a}}{ae,\ se,\ k:\ aut \xrightarrow{a}}(\mathrm{prio}(a) < k) \qquad \textbf{ag\_var}\ \frac{ae,\ se,\ k:\ \mathrm{lookup}(A, ae) \xrightarrow{a}}{ae,\ se,\ k:\ A \xrightarrow{a}}$$

$$\textbf{local\_agents}\ \frac{\mathrm{push\_frame}(ag\_frame, ae),\ se,\ k:\ p \xrightarrow{a}}{ae,\ se,\ k:\ p\ \texttt{where}\ ag\_frame\ \texttt{end} \xrightarrow{a}}$$

Figure 7.6: Semantics of PCCS - inits relation

7.4 Timed CCS

The version of Timed CCS (TCCS) implemented here borrows ideas from a number of timed process calculi. The notion of time is discrete, meaning that time proceeds in a series of ticks. In addition, the semantics of calculus employs the *maximal progress* hypothesis: time can only progress when no internal computation is possible.

7.4.1 Syntax of Timed CCS

The syntax of TCCS is almost identical to that of CCS and is given by the grammar in Figure 7.7. The key differences arise in the syntax of actions and in the inclusion of delay and disabling constructs. In addition to the input, output, and internal actions of CCS, TCCS allows (but does not require) internal, input and output to carry *labels*. A labeled internal action `@(`*id*`)` behaves like an internal action except that *id* is "visible". Labeled input and output actions have form id_1`(`id_2`)` and `'`id_1`(`id_2`)`, respectively; the intention is that when these actions synchronize the result is a labeled internal action. The semantics section makes this precise. The label-bearing capability turns out to be useful in analyzing the behavior of systems, as it allows specific synchronizations to be observed.

The *delay* construct has form n`.`*agent*, where n is a natural number. Intuitively, n`.`*agent* idles for n clock ticks before behaving like *agent*. The disabling construct `[>` allows its right agent argument to "interrupt" its left and is virtually identical to the construct found in LOTOS. Finally, the delabeling construct `:`S converts labeled internal actions whose labels are not in S into the unlabeled standard internal action. This operator can be used in conjunction with the CWB-NC minimization routines to reduce the sizes of system state spaces.

7.4.2 Semantics of Timed CCS

The SOS for TCCS is shown in Figures 7.8, 7.9 and 7.10, which define an auxiliary relation, *atrans*, in addition to the *transitions* relation The two relations have the following types and representations.

Relation	**Denoted**
$transitions \subseteq (agent\ env) \times (id\_set\ env) \times agent \times act \times agent$	$ae,\ se\ :\ p \xrightarrow{a} p'$
$atrans \subseteq (agent\ env) \times (id\_set\ env) \times agent \times act \times agent$	$ae,\ se\ :\ p \gg\!\xrightarrow{a} p'$

As was the case for CCS and PCCS, *transitions* determines the single-step execution steps that an agent is capable of performing. Here these steps may be (labeled) input, output, or internal actions, or delays. The *atrans* relation computes an over-approximation of a process's transitions; in particular, it does not take account of the maximal progress assumption and therefore allows delay steps even in the process of internal computation. The *transitions* relation then removes these erroneous transitions. The SOS rules for TCCS employ the auxiliary functions and predicates defined for plain CCS in Table 7.1 as well as those shown in Table 7.3. The semantics also distinguishes the subset of *user actions*, which consists of all actions except the clock tick.

Table 7.3: Functions and predicates used in the SOS for TCCS (augmenting Table 7.1).

$>, =: int \times int \rightarrow bool$	Integer comparison and equality.
dec : $int \rightarrow int$	Integer decrement operator.
external : $user\_act \rightarrow bool$	external(a) holds if a is not internal.
inverses : $user\_act \times user\_act \rightarrow bool$	inverses(a, b) holds if a and b represent an input/output pair on the same channel.
sync : $user\_act \times user\_act \rightarrow user\_act$	sync is a partial function that is only defined on a and b if a and b are inverses. When this is the case sync(a, b) returns a labeled or unlabeled internal action, depending on the form of a and b. If neither is labeled, then an unlabeled internal action is produced. If exactly one is labeled, then a labeled internal action is produced bearing this label. If both are labeled, then a labeled internal action is produced bearing the label of the input action.
labelof : $user\_act \rightarrow id$	Returns the channel component of the given action. Any label the action might also have is ignored; so labelof$a(b) = a$.
delabel : $user\_act \times act\_set \rightarrow user\_act$	Removes labels of internal actions if the labels do not belong to s. So delabel($\texttt{@a}, \{\texttt{b}\}) = \texttt{t}$.
. stable : $(agent\ env) \times (id\_set\ env) \times agent \rightarrow bool$	Stable is true of a process if it is incapable of performing an internal action. $$\text{stable}(ae, se, p) = \begin{cases} \text{false} & \text{if } ae,\ se\ :\ p \xrightarrow{\texttt{t}} \\ \text{true} & \text{otherwise} \end{cases}$$

Keywords
Aut, and, end, nil, proc, set, start, t, where

Tokens

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

Grammar

spec	:	*binding_list*
agent	:	"nil" | *id* | *user_a ct* "." *agent* | *int* "." *agent* | *agent* "+" *agent* | *agent* "|" *agent* | *agent* "\\" *restriction* | *agent* ":" *restriction* | *agent* "[" *relabeling* "]" | *agent* "[>" *agent* ; | *agent* "where" *agent_frame* "end" | ; *automaton* | "(" *agent* ")"
binding	:	"proc" *id* "=" *agent* | "set" *id* "=" *id_set*
agent_binding	:	*id* "=" *agent*
automaton	:	"Aut" "(" "start" "=" *int* "," *state_list* ")"
user_act	:	"t" | *id* | "'" *id* | *id* "(" *id* ")" | ěrb@”’”@ *id* "(" *id* ")" | "@" *i d*
act	:	*user_act* | "#"
act	:	*user_act* | "#"
state	:	*int* ":" *trans_list*
trans	:	*act int_list*
restriction	:	*id_set* | *id*
relabeling	:	*rename_list*
rename	:	*id* "/" *id*
id_set	:	"{" *id_list* "}"
id_set2	:	*id_list*
agent_frame	:	*agent_binding_list*
id	:	ID
int	:	INT

Lists

List name	May be Empty?	Open Delimiter	Separator	Close Delimiter	Items
binding_list	no	""	""	""	binding
agent_binding_list	no	""	"and"	""	agent_binding
rename_list	no	""	","	""	rename
id_list	no	""	","	""	id
state_list	no	""	""	""	state
trans_list	yes	""	""	""	trans
int_list	yes	"{"	","	"}"	int

Precedences

"[>" $<$ "|" $<$ "\" $<$ "[", "]", "{", "}", "where" $<$ "+" $<$ ".", ":"

Figure 7.7: Syntax of Timed CCS

vars

l : act	a, b : user_act
$p, p', p1, p1', p2, p2'$: agent	s, rs : id_set
r : relabeling	ae : (agent env)
se : (id_set env)	A : id
s_id : id	ag_frame : (agent frame)
aut, aut' : automaton	k : int

rules

$$\textbf{prefix}\ \frac{-}{ae,\ se\ :\ a.p \gg\!\xrightarrow{a}\!\triangleright\ p} \qquad \textbf{tprefix1}\ \frac{-}{ae,\ se\ :\ k.p \gg\!\xrightarrow{\#}\!\triangleright\ \mathrm{dec}(k).p}(k > 1)$$

$$\textbf{tprefix2}\ \frac{-}{ae,\ se\ :\ k.p \gg\!\xrightarrow{\#}\!\triangleright\ p}(k = 1) \qquad \textbf{tprefix3}\ \frac{-}{ae,\ se\ :\ a.p \gg\!\xrightarrow{\#}\!\triangleright\ a.p}(external(a))$$

$$\textbf{sum1}\ \frac{ae,\ se\ :\ p1 \gg\!\xrightarrow{a}\!\triangleright\ p1'}{ae,\ se\ :\ p1 + p2 \gg\!\xrightarrow{a}\!\triangleright\ p1'} \qquad \textbf{sum2}\ \frac{ae,\ se\ :\ p2 \gg\!\xrightarrow{a}\!\triangleright\ p2'}{ae,\ se\ :\ p1 + p2 \gg\!\xrightarrow{a}\!\triangleright\ p2'}$$

$$\textbf{tsum}\ \frac{ae,\ se\ :\ p1 \gg\!\xrightarrow{\#}\!\triangleright\ p1', ae,\ se\ :\ p2 \gg\!\xrightarrow{\#}\!\triangleright\ p2'}{ae,\ se\ :\ p1 + p2 \gg\!\xrightarrow{\#}\!\triangleright\ p1' + p2'}$$

$$\textbf{disabling1}\ \frac{ae,\ se\ :\ p1 \gg\!\xrightarrow{a}\!\triangleright\ p1'}{ae,\ se\ :\ p1[> p2 \gg\!\xrightarrow{a}\!\triangleright\ p1'[> p2} \qquad \textbf{disabling2}\ \frac{ae,\ se\ :\ p2 \gg\!\xrightarrow{a}\!\triangleright\ p2'}{ae,\ se\ :\ p1[> p2 \gg\!\xrightarrow{a}\!\triangleright\ p2'}$$

$$\textbf{tdisabling}\ \frac{ae,\ se\ :\ p1 \gg\!\xrightarrow{\#}\!\triangleright\ p1', ae,\ se\ :\ p2 \gg\!\xrightarrow{\#}\!\triangleright\ p2'}{ae,\ se\ :\ p1\texttt{[>}p2 \gg\!\xrightarrow{\#}\!\triangleright\ p1'\texttt{[>}p2'}$$

$$\textbf{par1}\ \frac{ae,\ se\ :\ p1 \gg\!\xrightarrow{a}\!\triangleright\ p1'}{ae,\ se\ :\ p1|p2 \gg\!\xrightarrow{a}\!\triangleright\ p1'|p2} \qquad \textbf{par2}\ \frac{ae,\ se\ :\ p2 \gg\!\xrightarrow{a}\!\triangleright\ p2'}{ae,\ se\ :\ p1|p2 \gg\!\xrightarrow{a}\!\triangleright\ p1|p2'}$$

$$\textbf{par3}\ \frac{ae,\ se\ :\ p1 \gg\!\xrightarrow{a}\!\triangleright\ p1', ae,\ se\ :\ p2 \gg\!\xrightarrow{b}\!\triangleright\ p2'}{ae,\ se\ :\ p1|p2 \gg\!\xrightarrow{\mathrm{sync}(a,b)}\!\triangleright\ p1'|p2'}(\mathrm{inverses}(a, b))$$

$$\textbf{tpar}\ \frac{ae,\ se\ :\ p1 \gg\!\xrightarrow{\#}\!\triangleright\ p1', ae,\ se\ :\ p2 \gg\!\xrightarrow{\#}\!\triangleright\ p2'}{ae,\ se\ :\ p1|p2 \gg\!\xrightarrow{\#}\!\triangleright\ p1'|p2'}$$

Figure 7.8: TCCS semantics: *atrans* relation, part I

$$\textbf{relabel}\ \frac{ae,\ se\ :\ p\ \gg\!\xrightarrow{a}\!\triangleright\, p'}{ae,\ se\ :\ p[r]\ \gg\!\xrightarrow{\text{apply}(a,r)}\!\triangleright\, p'[r]}$$

$$\textbf{trelabel}\ \frac{ae,\ se\ :\ p\ \gg\!\xrightarrow{\#}\!\triangleright\, p'}{ae,\ se\ :\ p[r]\ \gg\!\xrightarrow{\#}\!\triangleright\, p'[r]}$$

$$\textbf{restrict1}\ \frac{ae,\ se\ :\ p\ \gg\!\xrightarrow{a}\!\triangleright\, p'}{ae,\ se\ :\ p\backslash s\ \gg\!\xrightarrow{a}\!\triangleright\, p'\backslash s}(\text{labelof}(a) \notin s))$$

$$\textbf{trestrict1}\ \frac{ae,\ se\ :\ p\ \gg\!\xrightarrow{\#}\!\triangleright\, p'}{ae,\ se\ :\ p\backslash s\ \gg\!\xrightarrow{\#}\!\triangleright\, p'\backslash s}$$

$$\textbf{restrict2}\ \frac{ae,\ se\ :\ p\ \gg\!\xrightarrow{a}\!\triangleright\, p'}{ae,\ se\ :\ p\backslash s\_id\ \gg\!\xrightarrow{a}\!\triangleright\, p'\backslash s\_id}(\text{labelof}(a) \notin \text{lookup}(s\_id, se)))$$

$$\textbf{trestrict2}\ \frac{ae,\ se\ :\ p\ \gg\!\xrightarrow{\#}\!\triangleright\, p'}{ae,\ se\ :\ p\backslash s\_id\ \gg\!\xrightarrow{\#}\!\triangleright\, p'\backslash s\_id}$$

$$\textbf{delabel1}\ \frac{ae,\ se\ :\ p\ \gg\!\xrightarrow{a}\!\triangleright\, p'}{ae,\ se\ :\ p : s\ \gg\!\xrightarrow{\text{delabel}(a,s)}\!\triangleright\, p' : s}$$

$$\textbf{delabel2}\ \frac{ae,\ se\ :\ p\ \gg\!\xrightarrow{a}\!\triangleright\, p'}{ae,\ se\ :\ p : s\_id\ \gg\!\xrightarrow{\text{delabel}(a, \text{lookup}(s\_id, se))}\!\triangleright\, p' : s\_id}$$

$$\textbf{tdelabel1}\ \frac{ae,\ se\ :\ p\ \gg\!\xrightarrow{\#}\!\triangleright\, p'}{ae,\ se\ :\ p : s\ \gg\!\xrightarrow{\#}\!\triangleright\, p' : s}$$

$$\textbf{tdelabel2}\ \frac{ae,\ se\ :\ p\ \gg\!\xrightarrow{\#}\!\triangleright\, p'}{ae,\ se\ :\ p : s\_id\ \gg\!\xrightarrow{\#}\!\triangleright\, p' : s\_id}$$

$$\textbf{process\_constant}\ \frac{ae,\ se\ :\ \text{lookup}(A, ae)\ \gg\!\xrightarrow{a}\!\triangleright\, p'}{ae,\ se\ :\ A\ \gg\!\xrightarrow{a}\!\triangleright\, p'}$$

$$\textbf{tprocess\_constant}\ \frac{ae,\ se\ :\ \text{lookup}(A, ae)\ \gg\!\xrightarrow{\#}\!\triangleright\, p'}{ae,\ se\ :\ A\ \gg\!\xrightarrow{\#}\!\triangleright\, p'}$$

$$\textbf{fixpoint}\ \frac{\text{push\_frame}(ag\_frame, ae),\ se\ :\ p\ \gg\!\xrightarrow{a}\!\triangleright\, p'}{ae,\ se\ :\ p\ \texttt{where}\ ag\_frame\ \texttt{end}\ \gg\!\xrightarrow{a}\!\triangleright\, p'\ \texttt{where}\ ag\_frame\ \texttt{end}}$$

$$\textbf{tfixpoint}\ \frac{\text{push\_frame}(ag\_frame, ae),\ se\ :\ p\ \gg\!\xrightarrow{\#}\!\triangleright\, p'}{ae,\ se\ :\ p\ \texttt{where}\ ag\_frame\ \texttt{end}\ \gg\!\xrightarrow{\#}\!\triangleright\, p'\ \texttt{where}\ ag\_frame\ \texttt{end}}$$

$$\textbf{aut\_rule}\ \frac{aut\ >\!\xrightarrow{l}\!\triangleright\, aut'}{ae,\ se\ :\ aut\ \gg\!\xrightarrow{l}\!\triangleright\, aut'}$$

Figure 7.9: TCCS semantics: *atrans* relation, part II

vars

ae : (agent env) a : user_act
$p1, p2$: agent se : (id_set env)

rules

$$\textbf{action\_rule}\ \frac{ae,\ se\ :\ p1\ \gg\!\xrightarrow{a}\!\triangleright\ p2}{ae,\ se\ :\ p1\ \xrightarrow{a}\!\triangleright\ p2}$$

$$\textbf{tick\_rule}\ \frac{ae,\ se\ :\ p1\ \gg\!\xrightarrow{\#}\!\triangleright\ p2}{ae,\ se\ :\ p1\ \xrightarrow{\#}\!\triangleright\ p2}(\mathrm{stable}(ae, se, p))$$

Figure 7.10: TCCS semantics: *transitions* relation

Table 7.4: Auxiliary functions used in semantics of SCCS.

mult : $act \times act \rightarrow act$	$\text{mult}(a_1, a_2)$ returns the action corresponding to the simultaneous execution of a_1 and a_2.
apply : $morphism \times act \rightarrow act$	$\text{apply}(m, a)$ returns the action obtained by replacing each port p mentioned in a by the action $m(p)$ and then "multiplying out" the result.
generates : $id\_set \times act \rightarrow bool$	s generates a holds if a is an element of the subgroup generated by s (i.e. if a can be constructed from the ports in s).

7.5 SCCS

Synchronous CCS (SCCS) is a version of CCS featuring a synchronous notion of parallel composition [19]. In SCCS actions form a commutative group, with the group operation representing "simultaneous execution". The version of the language presented here changes some of the operators slightly; in particular, the "co-restriction" operator found in the original treatment is simplified somewhat. The interface was written by S. Arun-Kumar.

7.5.1 Syntax of SCCS

An SCCS specification consists of a sequence of declarations contained in a file whose name ends in the `.sccs` suffix. Comments begin with `*` and continue until an end-of-line character is encountered. Apart from this, Figure 7.11 summarizes the concrete syntax of SCCS. Two types of declarations bind an identifier to either a process or a set of identifiers. The keyword `proc` precedes a process declaration.

`proc` *id* = *agent*

This declaration binds the SCCS expression *agent* to the name *id*. Set declarations work similarly.

The syntax of agents deserves some comment. Terms `nil` and `@` represent termination and divergence, respectively, although `@` diverges only in the sense of representing undefined behavior; it has no transitions. The prefixing operator, `:`, allows actions to be prepended to agents. Actions generally have the form `'~a.b.~a.c'`, where the `'` and `'` serve as delimiters and the `.` can be thought of as the group operation on the set of actions. The group in question is freely generated from the set of identifiers ("port names"), with `~` representing the inverse operation. Finally, `+`, `#`, `!` and `[ ]` correspond to choice, synchronous parallel composition, restriction, and relabeling (morphism). Relabelings enable ports to be relabeled into actions; in $p[a/b.c]$ occurrences of a are replaced by $b.c$. It should also be noted that the pairs in a relabeling are written in *old*/*new* style, in contrast to the CCS *new*/*old* format used in other front ends in the CWB-NC.

7.5.2 Semantics of SCCS

The operational semantics of SCCS is given as a collection of SOS rules in Figure 7.12. In addition to the operations "lookup" and "push_frame", which are borrowed from the CCS specification and defined in Table 7.1, the semantics uses the auxiliary functions in Table 7.4.

tokens

```
   [1-9][0-9]{0,8} | 0    =>   INT
[a-zA-Z][a-zA-Z0-9~_-]*   =>   ID
```

grammar

spec : *binding_list*

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

agent : `"nil"` | `"@"` | *id* | *act* `":"` *agent* | *agent* `"+"` *agent* | *agent* `"#"` *agent* | *agent* `"!"` *restriction* | *agent* `"["` *morphism* `"]"` | *agent* `"where"` *agent_frame* `"end"` | *automaton* | `"("` *agent* `")"`

act : *part_list*

part : *id* | `"~"` *id*

restriction : *id_set* | *id*

id_set : `"{"` *id_list* `"}"`

morphism : *replace_list*

replace : *id* `"/"` *act*

agent_frame : *agent_binding_list*

agent_binding : *id* `"="` *agent*

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

state : *int* `":"` *trans_list*

trans : *act* *int_list*

id : ID

int : INT

lists

List name	May be Empty?	Open Delimiter	Separator	Close Delimiter	Items
part_list	yes	"‘"	"."	"’"	part
binding_list	no	""	""	""	binding
agent_binding_list	no	""	"and"	""	agent_binding
replace_list	yes	""	","	""	replace
id_list	no	""	","	""	id
state_list	no	""	""	""	state
trans_list	yes	""	""	""	trans
int_list	yes	"{"	","	"}"	int

priorities
`"where"` < `"+"` < `"#"` < `"!"` < `":"` < `"["` < `"."` < `"nil"`, `"@"`

Figure 7.11: Syntax of SCCS

vars

$a, a1, a2$: act	A : id
$p, p', p1, p1', p2, p2'$: agent	s_id : id
s : id_set	ag_frame : (agent frame)
m : morphism	aut, aut' : automaton
ae : (agent env)	se : (id_set env)

rules

$$\textbf{prefix}\ \frac{-}{ae,\ se\ :\ a : p \xrightarrow{a} p}$$

$$\textbf{sum}_1\ \frac{ae,\ se\ :\ p1 \xrightarrow{a} p1'}{ae,\ se\ :\ p1 + p2 \xrightarrow{a} p1'} \qquad \textbf{sum}_2\ \frac{ae,\ se\ :\ p2 \xrightarrow{a} p2'}{ae,\ se\ :\ p1 + p2 \xrightarrow{a} p2'}$$

$$\textbf{product}\ \frac{ae,\ se\ :\ p1 \xrightarrow{a1} p1',\quad ae,\ se\ :\ p2 \xrightarrow{a2} p2'}{ae,\ se\ :\ p1\#p2 \xrightarrow{\text{mult}(a1, a2)} p1'\#p2'}$$

$$\textbf{morphism}\ \frac{ae,\ se\ :\ p \xrightarrow{a} p'}{ae,\ se\ :\ p[m] \xrightarrow{\text{apply}(a, m)} p'[m]}$$

$$\textbf{restrict}_1\ \frac{ae,\ se\ :\ p \xrightarrow{a} p'}{ae,\ se\ :\ p!s \xrightarrow{a} p'!s}(s \text{ generates } a)$$

$$\textbf{restrict}_2\ \frac{ae,\ se\ :\ p \xrightarrow{a} p'}{ae,\ se\ :\ p!s\_id \xrightarrow{a} p'!s_id}(\text{lookup}(s\_id, se) \text{ generates } a)$$

$$\textbf{aut\_rule}\ \frac{aut\ \succ\!\xrightarrow{a} aut'}{ae,\ se\ :\ aut \xrightarrow{a} aut'} \qquad \textbf{process\_constant}\ \frac{ae,\ se\ :\ \text{lookup}(A, ae) \xrightarrow{a} p'}{ae,\ se\ :\ A \xrightarrow{a} p'}$$

$$\textbf{fixpoint}\ \frac{\text{push\_frame}(ag\_frame, ae),\ se\ :\ p \xrightarrow{a} p'}{ae,\ se\ :\ p \text{ where } ag\_frame \text{ end} \xrightarrow{a} p' \text{ where } ag\_frame \text{ end}}$$

Figure 7.12: Semantics of SCCS — transition relation

7.6 CSP

A CWB-NC interface for Hoare's Communicating Sequential Processes (CSP) [17] has also been implemented. The version of CSP implemented here is similar to the version of CSP supported by the FDR tool [14]; however, our language does not allow for the communication of data values as is possible in the FDR version. The CSP interface was implemented by Joel Gray [15].

7.6.1 Syntax of CSP

A CSP specification is defined as a sequence of declarations in a `.csp` suffixed file as indicated by the grammar in Figure 7.13. Comments begin with `--` and continue until an end of line character. Three types of declarations bind an identifier to either a process, action set, or relabeling. The keyword `proc` precedes a process declaration.

$$\texttt{proc}\ procName = cspTerm$$

Here the behavior of a system component is defined in CSP by *cspTerm* and this expression is bound to the variable *procName*. Declarations for action sets and relabelings are similar.

7.6.2 Semantics of CSP

The SOS for CSP, shown in Figures 7.14, 7.15, and 7.16, defines the relation:

$$transitions \subseteq process\ env \times set\ env \times relabeling\ env \times agent \times act \times agent$$

The first three components are environments which map identifiers to processes, id sets, and relabelings respectively. If $\langle ae, se, re, p, a, p'\rangle \in transitions$ then we write $ae,\ se,\ re\ :\ p \xrightarrow{a}\triangleright\ p'$ and read this as "p may perform an action a and become p' when free agent, set, and relabeling variables are resolved by the bindings defined in ae, se, and re." Note that actions may be either a synchronization over a channel, an internal action `t` (tau), or a termination action `delta` (sometimes referred to as a check action in the literature). In the SOS rules, *user_act* is a subclass of act that includes `t` and synchronization actions but not `delta`. Several set operators are used in the rules including $\in, \notin$, inter (intersection), and diff (difference). We adopt the notation that $se.s$ indicates that any free variables in the set s are resolved by the bindings in se. A renaming function r is defined by a list of renamings. A renaming of the form a `<-` b indicates that a should be replaced by b.

tokens

`[0-9]{1,9}`	`=>`	INT
`[a-zA-Z]|([a-zA-Z][a-zA-Z0-9'_-]*[a-zA-Z0-9'])`	`=>`	ID

grammar

spec : *binding_list*

binding : `"proc"` *id* `"="` *process* | `"set"` *id* `"="` *action_set* | `"relab"` *id* `"="` `"[["` *relabeling* `"]]"`

process : `"SKIP"` | `"STOP"` | *id* | *user_act* `"->"` *process* | *process* `";"` *process* | *process* `"[|"` *action_set* `"|]"` *process* | *process* `"["` *action_set* `"||"` *action_set* `"]"` *process* | *process* `"|||"` *process* | *process* `">"` *action_set* `">"` *process* | *process* `">"` *action_set* `","` *action_set* `">"` *process* | *process* `"[]"` *process* | *process* `"|~|"` *process* | *process* `"\"` *action_set* | *process* `"[["` *renaming* `"]]"` | *automaton* | `"("` *process* `")"`

act : *user_act* | `"delta"`

user_act : `"t"` | *id*

action_set : *id_set* | *id* | `"inter"` `"("` *action_set* `","` *action_set* `")"` | `"union"` `"("` *action_set* `","` *action_set* `")"` | `"diff"` `"("` *action_set* `","` *action_set* `")"`

id_set : `"{"` *id_list* `"}"`

renaming : *relabeling* | *id*

relabeling : *rename_list*

rename : *id* `"<-"` *id*

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

state : *int* `":"` *trans_list*

trans : *act int_list*

int : INT

id : ID

lists

List name	May be Empty?	Open Delimiter	Separator	Close Delimiter	Items
binding_list	no	""	""	""	binding
rename_list	no	""	","	""	rename
id_list	yes	""	","	""	id
state_list	no	""	""	""	state
trans_list	yes	""	""	""	trans
int_list	yes	"{"	","	"}"	int

priorities

`"|||"` $<$ `"[|","|]","[","]",">"` $<$ `"\","[["` $<$ `"[]","|~|"` $<$ `";"` $<$ `"->"` $<$ `"SKIP","STOP"`

Figure 7.13: Syntax of CSP

vars

p, p', q, q' : process
a, b : user_act
c, d : act
r : renaming
s, s_1, s_2 : action_set
A : id
pe : (process env)
se : (id_set env)
re : (relabeling env)

rules

$$\mathbf{seq_1}\ \frac{pe,\ se,\ re\ :\ p \xrightarrow{a} p'}{pe,\ se,\ re\ :\ p;q \xrightarrow{a} p';q} \qquad \mathbf{seq_2}\ \frac{pe,\ se,\ re\ :\ p \xrightarrow{\mathtt{delta}} p'}{pe,\ se,\ re\ :\ p;q \xrightarrow{\mathtt{t}} q}$$

$$\mathbf{par^1_1}\ \frac{pe,\ se,\ re\ :\ p \xrightarrow{a} p'}{pe,\ se,\ re\ :\ p[|s|]q \xrightarrow{a} p'[|s|]q}(\text{label(a)} \notin se.s) \qquad \mathbf{skip}\ \frac{}{pe,\ se,\ re\ :\ \mathtt{SKIP} \xrightarrow{\mathtt{delta}} \mathtt{STOP}}$$

$$\mathbf{par^1_2}\ \frac{pe,\ se,\ re\ :\ q \xrightarrow{a} q'}{pe,\ se,\ re\ :\ p[|s|]q \xrightarrow{a} p[|s|]q'}(\text{label(a)} \notin se.s) \qquad \mathbf{pre}\ \frac{}{pe,\ se,\ re\ :\ (a \mathbin{-\!\!>} p) \xrightarrow{a} p}$$

$$\mathbf{par^1_3}\ \frac{pe,\ se,\ re\ :\ p \xrightarrow{c} p',\ \ pe,\ se,\ re\ :\ q \xrightarrow{d} q'}{pe,\ se,\ re\ :\ p[|s|]q \xrightarrow{c} p'[|s|]q'}((c = d)\text{ and }((\text{label(c)} \in se.s)\text{ or }(c = \mathtt{delta})))$$

$$\mathbf{par^2_1}\ \frac{pe,\ se,\ re\ :\ p \xrightarrow{a} p'}{pe,\ se,\ re\ :\ p[s_1||s_2]q \xrightarrow{a} p'[s_1||s_2]q}(\text{label(a)} \in se.\text{diff}(s_1, s_2))$$

$$\mathbf{par^2_2}\ \frac{pe,\ se,\ re\ :\ q \xrightarrow{a} q'}{pe,\ se,\ re\ :\ p[s_1||s_2]q \xrightarrow{a} p[s_1||s_2]q'}(\text{label(a)} \in se.\text{diff}(s_2, s_1))$$

$$\mathbf{par^2_3}\ \frac{pe,\ se,\ re\ :\ p \xrightarrow{c} p',\ \ pe,\ se,\ re\ :\ q \xrightarrow{d} q'}{pe,\ se,\ re\ :\ p[s_1||s_2]q \xrightarrow{c} p'[s_1||s_2]q'}((c = d)\text{ and }((\text{label(c)} \in se.\text{inter}(s_1, s_2))\text{ or }(c = \mathtt{delta})))$$

$$\mathbf{par^3_1}\ \frac{pe,\ se,\ re\ :\ p \xrightarrow{a} p'}{pe,\ se,\ re\ :\ p|||q \xrightarrow{a} p'|||q} \qquad \mathbf{par^3_2}\ \frac{pe,\ se,\ re\ :\ q \xrightarrow{a} q'}{pe,\ se,\ re\ :\ p|||q \xrightarrow{a} p|||q'}$$

$$\mathbf{par^3_3}\ \frac{pe,\ se,\ re\ :\ p \xrightarrow{\mathtt{delta}} p',\ \ pe,\ se,\ re\ :\ q \xrightarrow{\mathtt{delta}} q'}{pe,\ se,\ re\ :\ p|||q \xrightarrow{\mathtt{delta}} p'|||q'}$$

Figure 7.14: Semantics of CSP - transitions relation, part 1

$$\textbf{chain}_1^1 \; \frac{pe,\ se,\ re\ :\ p \xrightarrow{a} p'}{pe,\ se,\ re\ :\ p > s > q \xrightarrow{a} p' > s > q} (\text{label(a)} \notin se.s)$$

$$\textbf{chain}_2^1 \; \frac{pe,\ se,\ re\ :\ q \xrightarrow{a} q'}{pe,\ se,\ re\ :\ p > s > q \xrightarrow{a} p > s > q'} (\text{label(a)} \notin se.s)$$

$$\textbf{chain}_3^1 \; \frac{pe,\ se,\ re\ :\ p \xrightarrow{a} p', \quad pe,\ se,\ re\ :\ q \xrightarrow{b} q'}{pe,\ se,\ re\ :\ p > s > q \xrightarrow{\texttt{t}} p' > s > q'} ((a = b) \text{ and } (\text{label(a)} \in se.s))$$

$$\textbf{chain}_4^1 \; \frac{pe,\ se,\ re\ :\ p \xrightarrow{\texttt{delta}} p', \quad pe,\ se,\ re\ :\ q \xrightarrow{\texttt{delta}} q'}{pe,\ se,\ re\ :\ p > s > q \xrightarrow{\texttt{delta}} p' > s > q'}$$

$$\textbf{chain}_1^2 \; \frac{pe,\ se,\ re\ :\ p \xrightarrow{a} p'}{pe,\ se,\ re\ :\ p > s1, s2 > q \xrightarrow{a} p' > s1, s2 > q} (\text{label(a)} \in se.\text{diff}(s1, s2))$$

$$\textbf{chain}_2^2 \; \frac{pe,\ se,\ re\ :\ q \xrightarrow{a} q'}{pe,\ se,\ re\ :\ p > s1, s2 > q \xrightarrow{a} p > s1, s2 > q'} (\text{label(a)} \in se.\text{diff}(s2, s1))$$

$$\textbf{chain}_3^2 \; \frac{pe,\ se,\ re\ :\ p \xrightarrow{a} p', \quad pe,\ se,\ re\ :\ q \xrightarrow{b} q'}{pe,\ se,\ re\ :\ p > s1, s2 > q \xrightarrow{\texttt{t}} p' > s1, s2 > q'} ((a = b) \text{ and } (\text{label(a)} \in se.\text{inter}(s1, s2)))$$

$$\textbf{chain}_4^2 \; \frac{pe,\ se,\ re\ :\ p \xrightarrow{\texttt{delta}} p', \quad pe,\ se,\ re\ :\ q \xrightarrow{\texttt{delta}} q'}{pe,\ se,\ re\ :\ p > s1, s2 > q \xrightarrow{\texttt{delta}} p' > s1, s2 > q'}$$

Figure 7.15: Semantics of CSP - transitions relation, part 2

$$\textbf{ext\_choice}_1\ \frac{pe,\ se,\ re\ :\ p \xrightarrow{a} p'}{pe,\ se,\ re\ :\ p[]q \xrightarrow{\mathtt{t}} p'[]q}(a = \mathtt{t}) \qquad \textbf{ext\_choice}_2\ \frac{pe,\ se,\ re\ :\ q \xrightarrow{a} q'}{pe,\ se,\ re\ :\ p[]q \xrightarrow{\mathtt{t}} p[]q'}(a = \mathtt{t})$$

$$\textbf{ext\_choice}_3\ \frac{pe,\ se,\ re\ :\ p \xrightarrow{c} p'}{pe,\ se,\ re\ :\ p[]q \xrightarrow{c} p'}(c \neq \mathtt{t}) \qquad \textbf{ext\_choice}_4\ \frac{pe,\ se,\ re\ :\ q \xrightarrow{c} q'}{pe,\ se,\ re\ :\ p[]q \xrightarrow{c} q'}(c \neq \mathtt{t})$$

$$\textbf{int\_choice}_1\ \frac{}{pe,\ se,\ re\ :\ p \sqcap q \xrightarrow{\mathtt{t}} p} \qquad \textbf{int\_choice}_2\ \frac{}{pe,\ se,\ re\ :\ p \sqcap q \xrightarrow{\mathtt{t}} q}$$

$$\textbf{hide}_1\ \frac{pe,\ se,\ re\ :\ p \xrightarrow{c} p'}{pe,\ se,\ re\ :\ p \setminus s \xrightarrow{c} p' \setminus s}(\text{label(c)} \notin se.s)$$

$$\textbf{hide}_2\ \frac{pe,\ se,\ re\ :\ p \xrightarrow{a} p'}{pe,\ se,\ re\ :\ p \setminus s \xrightarrow{\mathtt{t}} p' \setminus s}(\text{label(a)} \in se.s)$$

$$\textbf{rel}_1\ \frac{pe,\ se,\ re\ :\ p \xrightarrow{c} p'}{pe,\ se,\ re\ :\ p[[r]] \xrightarrow{\text{apply\_renaming}(c, re, r)} p'[[r]]}$$

$$\textbf{proc\_var}\ \frac{pe,\ se,\ re\ :\ \text{lookup}(A, pe) \xrightarrow{c} p'}{pe,\ se,\ re\ :\ A \xrightarrow{c} p'}$$

Figure 7.16: Semantics of CSP - transitions relation, part 3

7.7 Basic LOTOS

For an introduction to Basic LOTOS, see [3]. We adopt the standard notation for all operators, but, for efficiency reasons, do not allow local process definitions.

7.7.1 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 Figure 7.17. The shorthand `process`(A) is used to indicate that the process bound to identifier A is invoked with actual parameters the same as its formal parameters.

7.7.2 Semantics of Basic LOTOS

The operational semantics of Basic LOTOS is shown in Figures 7.18 and 7.19.

tokens

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

grammar

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

lists

List name	May be Empty?	Open Delimiter	Separator	Close Delimiter	Items
id_list	no	""	","	""	id
process_def_list	no	""	""	""	process
rename_list	no	""	","	""	rename
state_list	no	""	""	""	state
trans_list	yes	""	""	""	trans
int_list	yes	"{"	","	"}"	int

priorities

">>","[>" $<$ "|[","]|","||","|||" $<$ "[","]" $<$ "where","endproc","in","exit" $<$ "[]" $<$ ";" $<$ "stop"

Figure 7.17: Syntax of Basic LOTOS

vars

a, b : user_act
c, d : act
pe : (proc env)
$p, p', p_1, p_1', p_2, p_2'$: bexp
S : id_set
r, r_1, r_2 : relabeling
g : (id list)
A : id
aut, aut' : automaton

rules

$$\mathbf{pre}\ \frac{}{pe\ :\ a;p \xrightarrow{a} p} \qquad \mathbf{choice_1}\ \frac{pe\ :\ p_1 \xrightarrow{c} p_1'}{pe\ :\ p_1[]p_2 \xrightarrow{c} p_1'} \qquad \mathbf{choice_2}\ \frac{pe\ :\ p_2 \xrightarrow{c} p_2'}{pe\ :\ p_1[]p_2 \xrightarrow{c} p_2'}$$

$$\mathbf{par_1^1}\ \frac{pe\ :\ p_1 \xrightarrow{a} p_1'}{pe\ :\ p_1|[S]|p_2 \xrightarrow{a} p_1'|[S]|p_2}(\text{label(a)} \notin S) \qquad \mathbf{par_2^1}\ \frac{pe\ :\ p_2 \xrightarrow{a} p_2'}{pe\ :\ p_1|[S]|p_2 \xrightarrow{a} p_1|[S]|p_2'}(\text{label(a)} \notin S)$$

$$\mathbf{par_3^1}\ \frac{pe\ :\ p_1 \xrightarrow{c} p_1',\quad pe\ :\ p_2 \xrightarrow{d} p_2'}{pe\ :\ p_1|[S]|p_2 \xrightarrow{c} p_1'|[S]|p_2'}((c = d) \text{ and } ((\text{label(c)} \in S) \text{ or } (c = \texttt{delta})))$$

$$\mathbf{par_1^2}\ \frac{pe\ :\ p_1 \xrightarrow{\texttt{i}} p_1'}{pe\ :\ p_1||p_2 \xrightarrow{\texttt{i}} p_1'||p_2} \qquad \mathbf{par_2^2}\ \frac{pe\ :\ p_2 \xrightarrow{\texttt{i}} p_2'}{pe\ :\ p_1||p_2 \xrightarrow{\texttt{i}} p_1||p_2'}$$

$$\mathbf{par_3^2}\ \frac{pe\ :\ p_1 \xrightarrow{c} p_1',\quad pe\ :\ p_2 \xrightarrow{d} p_2'}{pe\ :\ p_1||p_2 \xrightarrow{c} p_1'||p_2'}((c = d) \text{ and } (c \neq \texttt{i}))$$

$$\mathbf{par_1^3}\ \frac{pe\ :\ p_1 \xrightarrow{a} p_1'}{pe\ :\ p_1|||p_2 \xrightarrow{a} p_1'|||p_2} \qquad \mathbf{par_2^3}\ \frac{pe\ :\ p_2 \xrightarrow{a} p_2'}{pe\ :\ p_1|||p_2 \xrightarrow{a} p_1|||p_2'}$$

$$\mathbf{par_3^3}\ \frac{pe\ :\ p_1 \xrightarrow{\texttt{delta}} p_1',\quad pe\ :\ p_2 \xrightarrow{\texttt{delta}} p_2'}{pe\ :\ p_1|||p_2 \xrightarrow{\texttt{delta}} p_1'|||p_2'}$$

$$\mathbf{exit}\ \frac{}{pe\ :\ \texttt{exit} \xrightarrow{\texttt{delta}} \texttt{stop}}$$

Figure 7.18: Semantics of Basic LOTOS - transitions relation, part 1

$$\mathbf{hide_1}\;\frac{pe \;:\; p \xrightarrow{c} p'}{pe \;:\; \texttt{hide}\ S\ \texttt{in}\ p \xrightarrow{c} \texttt{hide}\ S\ \texttt{in}\ p'}(\mathrm{label(c)} \notin S)$$

$$\mathbf{hide_2}\;\frac{pe \;:\; p \xrightarrow{a} p'}{pe \;:\; \texttt{hide}\ S\ \texttt{in}\ p \xrightarrow{\texttt{i}} \texttt{hide}\ S\ \texttt{in}\ p'}(\mathrm{label(a)} \in S)$$

$$\mathbf{enab_1}\;\frac{pe \;:\; p_1 \xrightarrow{a} p_1'}{pe \;:\; p_1 >> p_2 \xrightarrow{a} p_1' >> p_2} \qquad \mathbf{enab_2}\;\frac{pe \;:\; p_1 \xrightarrow{\texttt{delta}} p_1'}{pe \;:\; p_1 >> p_2 \xrightarrow{\texttt{i}} p_2}$$

$$\mathbf{disab_1}\;\frac{pe \;:\; p_1 \xrightarrow{a} p_1'}{pe \;:\; p_1[> p_2 \xrightarrow{a} p_1'[> p_2} \qquad \mathbf{disab_2}\;\frac{pe \;:\; p_1 \xrightarrow{\texttt{delta}} p_1'}{pe \;:\; p_1[> p_2 \xrightarrow{\texttt{delta}} p_1'}$$

$$\mathbf{disab_3}\;\frac{pe \;:\; p_2 \xrightarrow{c} p_2'}{pe \;:\; p_1[> p_2 \xrightarrow{c} p_2'}$$

$$\mathbf{rel_1}\;\frac{pe \;:\; p[\mathrm{compose}(r_1 : r_2)] \xrightarrow{c} p'}{pe \;:\; p[r_1][r_2] \xrightarrow{c} p'} \qquad \mathbf{rel_2}\;\frac{pe \;:\; p \xrightarrow{c} p'}{pe \;:\; p[r] \xrightarrow{\mathrm{apply}(c,r)} p'[r]}(\mathrm{not\_relab}(p))$$

$$\mathbf{proc\_inst_1}\;\frac{pe \;:\; \mathrm{get\_bexp}(\mathrm{lookup}(A, pe))[\mathrm{mk\_relabeling}(g, \mathrm{get\_formal\_gates}(\mathrm{lookup}(A, pe)))] \xrightarrow{c} p'}{pe \;:\; A[g] \xrightarrow{c} p'}$$

$$\mathbf{proc\_inst_2}\;\frac{pe \;:\; \mathrm{get\_bexp}(\mathrm{lookup}(A, pe)) \xrightarrow{c} p'}{pe \;:\; \texttt{process}(A) \xrightarrow{c} p'}$$

$$\mathbf{aut}\;\frac{aut \; > \!\xrightarrow{a} aut'}{pe \;:\; aut \xrightarrow{a} aut'}$$

Figure 7.19: Semantics of Basic LOTOS - transitions relation, part 2

Chapter 8

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.

8.1 Top-Level Commands

NAME

caching

SYNOPSIS

caching arg

DESCRIPTION

The "caching" command toggles whether or not transitions are cached during analysis. "arg" is either "on" or "off" When caching is on, the transitions of subterms of the parallel composition operator are cached as they are computed, which usually leads to impressive speedups since these are recomputed numerous times for a typical design. However, if the subterms of a parallel composition operator are automata (as is the case when doing component wise minimization) caching is unnecessary and wastes space, therefore should not be used.

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 [-L logic] [-O] agent formula

DESCRIPTION

chk invokes the model checker to determine whether or not "agent" satisfies "formula". The currently supported temporal logics for the specification of formulas are the mu-calculus (-L mu, which is the default) and GCTL* (-L gctl). The -O option indicates that model-checking is to be on-the-fly, if possible. Global model-checking, if possible, is the default. At this time, global model-checking is not possible for GCTL* formulas and is only available for alternation-free mu-calculus formulas.

EXAMPLE

```
cwb-nc> load abp.ccs
Execution time (user,system,gc,real):(0.010,0.020,0.000,0.074)
cwb-nc> load /abp.mu
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.016)
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):(0.030,0.000,0.000,0.027)
cwb-nc> chk -O ABP-safe can_deadlock
Invoking L2 model checker.
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.000,0.010,0.000,0.016)
cwb-nc> chk -L gctl ABP-safe "E F ~{-}"
Generating ABTA from GCTL* formula...done
Initial ABTA has 6 states.
Simplifying ABTA:
Minimizing sets of accepting states...done
Performing constant propagation...done
Joining commutative operations...done
Shrinking automaton...done
Computing bisimulation...
Done computing bisimulation.
Simplification completed.
Simplified ABTA has 4 states.
Starting ABTA model checker.
Model checking completed.
Expanded state-space 15 times.
Stored 0 dependencies.
```

```
TRUE, the agent satisfies the formula.
Execution time (user,system,gc,real):(0.010,0.000,0.000,0.013)
```

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

save

SYNOPSIS

save filename [identifier ...]

DESCRIPTION

The save command saves the bindings in the current environment, or selected bindings, to a file. If one or more identifier arguments are given, then the bindings for these identifiers are saved to filename. If no identifier arguments are given, then all bindings in the environment are saved to a file. The save command is useful for saving minimized automata for future use.

SEE ALSO

load

NAME

search

SYNOPSIS

search agent formula

DESCRIPTION

search invokes the CWB-NC reachability analysis routine. Given an agent and a formula the routine searches the set of reachable states in the agent for a state satisfying the given formula. If such a state is found, the simulator is invoked with a path to the state pre-loaded. The formula may be any mu-calculus (or CTL) formula supported by the CWB-NC model checkers; however, in most cases it will be a Hennessy-Milner Logic (HML) formula that may be checked very quickly at each state. Note that a HML formula is simply a mu-calculus formula that contains no fix point operators or weak modalities.

EXAMPLE

```
cwb-nc> load abp.ccs
Execution time (user,system,gc,real):(0.140,0.020,0.020,0.212)
cwb-nc> load abp.mu
Execution time (user,system,gc,real):(0.030,0.000,0.000,0.035)
cwb-nc> search ABP-safe is_deadlocked

States explored: 8
State found satisfying is_deadlocked.
Path to state contains 8 states, invoking simulator.
1: ABP-safe -- send -->
2: (R0 | Msafe | S0')\Internals -- t -->
3: (R0 | 'r0.Msafe | (rack0.S1 + rack1.S0' + t.S0'))\Internals -- t -->
4: ('receive.'sack0.R1 | Msafe | (rack0.S1 + rack1.S0' + t.S0'))\
 Internals -- t -->
5: ('receive.'sack0.R1 | Msafe | S0')\Internals -- t -->
6: ('receive.'sack0.R1 | 'r0.Msafe | (rack0.S1 + rack1.S0' + t.S0'))\
 Internals -- t -->
7: ('receive.'sack0.R1 | 'r0.Msafe | S0')\Internals -- 'receive -->
8: ('sack0.R1 | 'r0.Msafe | S0')\Internals
cwb-nc-sim>
```

SEE ALSO

fd, chk, load

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.
```

8.2 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 [flag] 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

```
cwb-nc> 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 "num" is a positive integer less than or equal to the 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

semantics

SYNOPSIS

semantics semantic-type

DESCRIPTION

The "semantics" command sets the mode for computing transitions in the simulator. The two available "semantic-type" arguments are "bisim" and "obseq". The two different modes treat internal actions differently. In "bisim" (bisimulation) mode internal actions are treated exactly as visible actions. In "obseq" (observational equivalence) mode the transitions displayed are those that may be taken from the current system state by performing zero or more internal actions followed by some action (either visible or internal) followed by zero or more internal actions.

EXAMPLE

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

Bibliography

[1] G. Bhat. *Tableau-Based Approaches to Model-Checking.* PhD thesis, North Carolina State University, Raleigh, 1998.

[2] G. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In *Eleventh Annual Symposium on Logic in Computer Science (LICS '96)*, pages 304–312, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

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

[4] G. Bruns. *Distributed Systems Analysis with CCS.* Prentice-Hall, London, 1997.

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

[6] 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.

[7] 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.

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

[9] R. Cleaveland, G. Luettgen, V. Natarajan, and S. Sims. Modeling and verifying distributed systems using priorities: A case study. *Software Concepts and Tools*, 17(2):50–62, 1996.

[10] 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.

[11] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In R. Alur and T. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[12] E.A. Emerson and J.Y. Halpern. 'Sometime' and 'not never' revisited: On branching versus linear time temporal logic. *Journal of the Association for Computing Machinery*, 33(1):151–178, January 1986.

[13] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Symposium on Logic in Computer Science (LICS '86)*, pages 267–278, Cambridge, Massachusetts, June 1986. IEEE Computer Society Press.

[14] Formal Systems (Europe) Ltd. *Failures Divergence Refinement: User Manual and Tutorial Version 1.42*, February 1995.

[15] J. Gray. A CSP interface for the concurrency workbench. Undergraduate Honors Thesis, Department of Computer Science, North Carolina State University, May 1996.

[16] 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.

[17] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, London, 1985.

[18] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, December 1983.

[19] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

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

[21] F. Moller and C. Tofts. Relating processes with respect to speed. In J.C.M. Baeten and J.F. Groote, editors, *CONCUR '91*, volume 527 of *Lecture Notes in Computer Science*, pages 424–438, Amsterdam, August 1991. Springer-Verlag.

[22] The Process Algebra Compiler of North Carolina. `http://www.reactive-systems.com/pac/`.

[23] The Concurrency Workbench of the New Century. `http://www.cs.sunysb.edu/~cwb`.

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

[25] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

Index