Research Report
ISSN 0167-9708
Coden: TEUEDE

Eindhoven
University of Technology
Netherlands

Faculty of Electrical Engineering

CCSTOOL2: An Expansion, Minimization and Verification Tool for Finite State CCS Descriptions

by
A. van Rangelrooij
J.P.M. Voeten

EUT Report 94-E-284
ISBN 90-6144-284-2
December 1994

Eindhoven University of Technology Research Reports

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
Eindhoven, The Netherlands

ISSN 0167-9708 Coden: TEUEDE

CCSTOOL2: An Expansion, Minimization, and Verification Tool for Finite State CCS Descriptions

by

A. van Rangelrooij
J.P.M. Voeten

EUT Report 94-E-284
ISBN 90-6144-284-2

Eindhoven
December 1994

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Rangelrooij, A. van

CCSTOOL2: an expansion, minimization, and verification
tool for finite state CCS descriptions / by A. van
Rangelrooij, J.P.M. Voeten. - Eindhoven : Eindhoven
University of Technology, Faculty of Electrical
Engineering. - Fig., tab. - (EUT report, ISSN 0167-9708 ;
94-E-284)
With ref.
ISBN 90-6144-284-2
NUGI 832
Subject headings: formal verification / formal
specification / CCS.

CCSTOOL2: An Expansion, Minimization, and Verification Tool for Finite State CCS Descriptions

A. van Rangelrooij and J.P.M. Voeten

Abstract

CCStool2 is an automated tool able to perform various computations on fsCCS (finite state CCS (Calculus of Communicating Systems)) descriptions, including expansion, reduction, abstraction, restriction, and strong and weak minimization and verification. Basically, CCStool2 behaves as a filter taking an input file containing an fsCCS description together with a command indicating the computation to be performed on that description, and generating an output file containing the result of the computation in the form of another fsCCS description and/or optionally a report file containing information concerning the performed computation. CCStool2 is efficient both in time and space, and can handle fsCCS descriptions with a relative large number of states and transitions. In contrast with other verification tools, graph generation (expansion) is performed very time-efficiently, probably due to the chosen fsCCS description language. The tool has been implemented in the programming language C and is portable to a wide range of computers and operating systems including MS-DOS and UNIX.

The key feature of CCStool2 is modularity. The fsCCS description language is compositional, the input files and output files are in plain ASCII, and the provided functions can easily be combined to form more complex ones. Therefore, CCStool2 together with file-handling and text-handling commands and tools provide a powerful mechanism to use macros or scripts to define generic higher-order functions and specific complex tasks.

Keywords: formal verification, formal specification, CCS.

Van Rangelrooij A. and J.P.M. Voeten
CCSTOOL2: An Expansion, Minimization, and Verification Tool
for Finite State CCS Descriptions
Eindhoven : Faculty of Electrical Engineering, Eindhoven University of Technology, 1994.
EUT Report 94-E-284

Address of the authors:
Section of Digital Information Systems
Faculty of Electrical Engineering
Eindhoven University of Technology
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Currently, there is a growing interest within the Digital Information Systems Group in the use of formal specification, description, and verification techniques for the development of complex digital information systems. One of the techniques we are looking at is CCS (Calculus of Communicating Systems) [Mil89, Koo91]. CCS is one of the first theories dealing in a formal way with the communication behaviour of parallel systems. Over the past years it has proven its usefulness in the specification and verification of complex parallel systems. A number of currently used formal specification, description, and verification techniques are (partially) based on CCS.

In practice, CCS (and other algebraic process) descriptions tend to become rather large. Without the proper tools such descriptions become unmanageble. This especially applies to computations on CCS descriptions, such as expansion, minimization, and verification, which are rather laborious, time consuming, and prone to error if done by hand. In this report we describe an automated tool called CCStool2 which has been developed within the Digital Information Systems Group, and which is able to perform various computations on fsCCS (finite state CCS) descriptions. This chapter gives a global description of CCStool2, relates it to already-existing tools, and gives an overview of the remainder of this report.

Section 1.1 gives a global description CCStool2. Section 1.2 describes a number of related tools. Section 1.3 contains an overview of the organization of the remainder of this report.

1.1 CCStool2

We have developed CCStool2 with the following goals in mind: Firstly, it should offer a rather complete set of basic functions including expansion, minimization, and verification (see below), which should be easily extensible to meet possible future requirements. Secondly, CCStool2 should support the possibility to combine these basic functions into user-defined macros or scripts. Thirdly, the tool should be efficient in both time and space. Since in practice space efficiency is more important than time efficiency [Kor91], we especially put effort in the former. Last but not least, the tool should be easy to use, and portable to a wide range of computers and operating systems including at least MS-DOS and UNIX.

CCStool2 offers the following main functions:

- expansion
- reduction modulo observational equivalence
- minimization modulo observational equivalence
- minimization modulo strong equivalence
- observational equivalence verification
- strong equivalence verification

Further, it offers the following supporting functions:

- syntactical and semantical check
- state relabeling
- action relabeling
- extraction
- abstraction
- restriction

1.2 Related Work

Several other automatic verification tools for the analysis of concurrent systems have been developed and are still under development. Examples are The Concurrency Workbench [CPS89, CPS93], Auto & Autograph [BRSV89, RS90], Winston [MSGS88], and Clara [GS88]. They vary in the supported process algebras and process calculi, the offered functionality, and the efficiency in time and space. An overview and evaluation of a number of tools can be found in [Kor91].

1.3 Report Organization

In this report we describe the functionality of CCStool2, the core algorithms implementing the main part of this functionality, various implementation aspects, and the macro functionality. Further, it gives the results of several performance tests on CCStool2. Finally, it presents the conclusions and some recommendations for future extensions. For information on how to install and use CCStool2, see [VV94].

The remainder of this report is organized as follows:

- Chapter 2 gives a detailed description of the functionality of CCStool2.
- Chapter 3 discusses the core algorithms implementing the main part of this functionality.
- Chapter 4 describes various implementation aspects of CCStool2.
- Chapter 5 presents the macro functionality of CCStool2.
- Chapter 6 contains the results of several performance tests on CCStool2.
- Chapter 7 presents the conclusions and recommendations for future extensions.
- Appendix A contains a context-free grammar of the fsCCS description language of CCStool2.

Chapter 2

Functionality

In this chapter we give a detailed description of the functionality of CCStool2. We assume that the reader is familiar with CCS. For a formal introduction into CCS we suggest [Mil89]. In [Koo91] a link is established between theoretical work on CCS and its application in several design fields.

As mentioned in Chapter 1, CCStool2 can perform the following computations on fsCCS (finite state CCS) descriptions:

- syntactical and semantical check
- state relabeling
- action relabeling
- extraction
- abstraction
- restriction
- expansion
- minimization modulo observational equivalence
- minimization modulo strong equivalence
- reduction modulo observational equivalence
- observational equivalence verification
- strong equivalence verification

Section 2.1 presents the basic functionality of CCStool2. Sections 2.2–2.13 respectively give a detailed description of each of the computations listed above. Section 2.14 describes several combinations of computations.

2.1 Basic Functionality

Basically, CCStool2 behaves as a filter taking an input file containing an fsCCS description together with a command indicating the computation to be performed on this description, and generating an output file containing the result of the computation in the form of another fsCCS description and/or optionally a report file containing information concerning the performed computation.

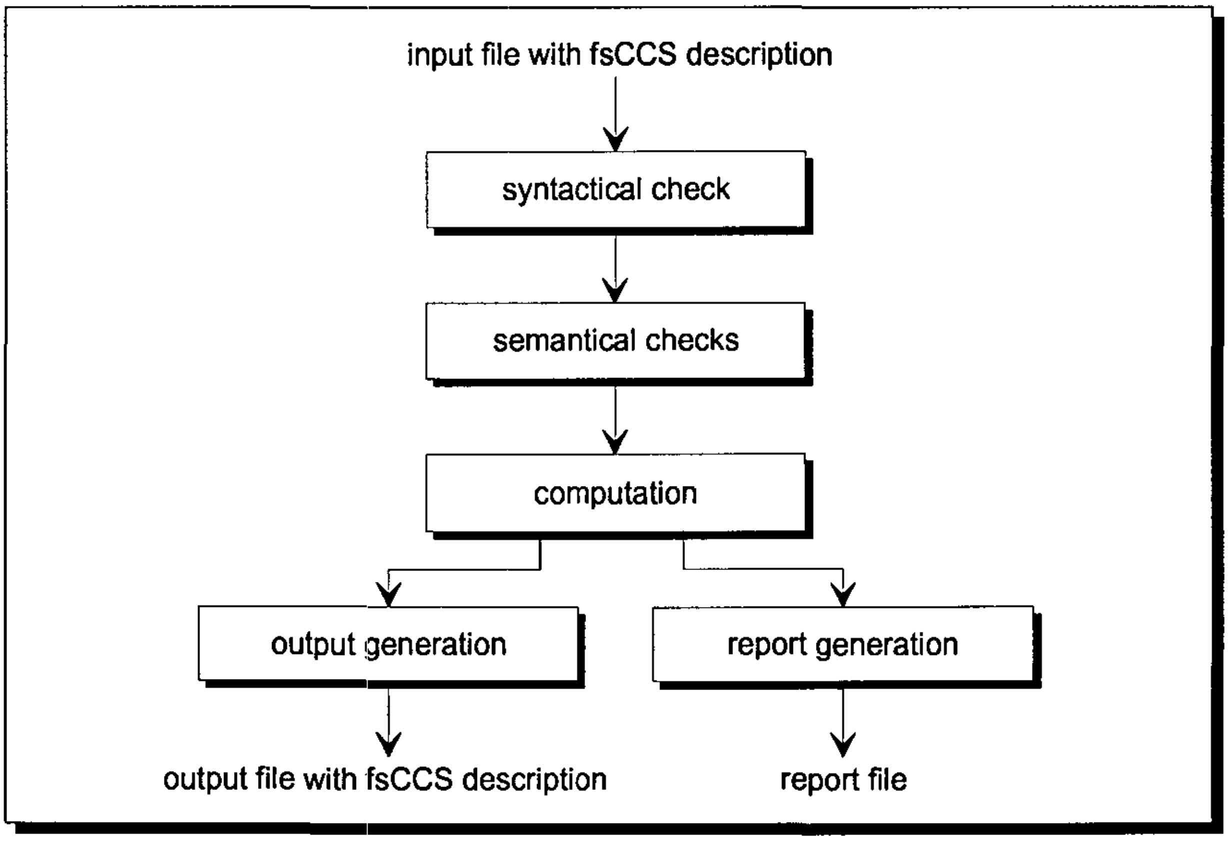

Figure 2.1: Flow diagram of the behaviour of CCStool2

A flow diagram of the behaviour of CCStool2 is shown in figure 2.1. After reading an input file with an fsCCS description, first a syntactical check is performed on this description. Next, various semantical checks are performed. If no syntactical or semantical errors are encountered, the indicated computation is performed. Finally, the result of this computation is decompiled into an fsCCS description and written to an output file, and/or optionally a report is generated and written to a report file.

An fsCCS description consists of a number of fsCCS equations and a number of operators.

The fsCCS equations specify the "pure" behaviour of one or more finite state CCS agents. The operators specify certain operations to be performed on certain states or actions of these agents. Further, an fsCCS description may also contain comment. Appendix A contains a context-free grammar defining the fsCCS description language of CCStool2. This language is based on the CCS description language described in [Por87] and on the so-called normal form of [Koo91].

As an example we consider the parallel composition of two 1-place buffers called $BufA_0$ and $BufB_0$ as shown in figure 2.2.

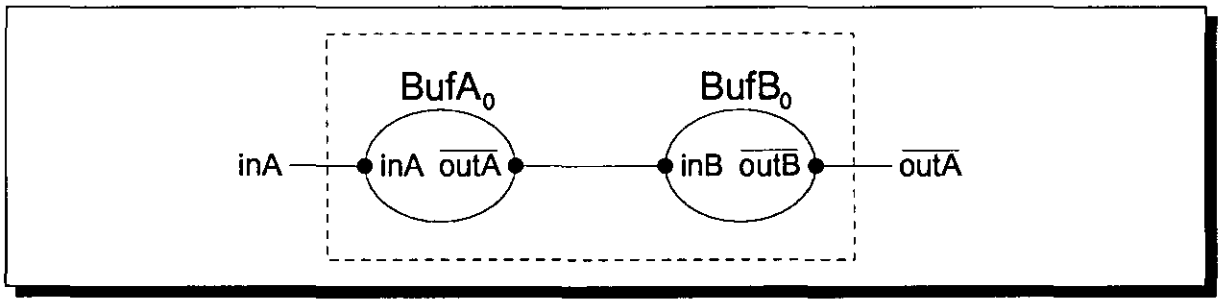

Figure 2.2: Parallel composition of two 1-place buffers

These buffers may be specified by the following fsCCS agent expression:

$$\big(\big(BufA_0 \mid (BufB_0\,[outA/inB])\big) \setminus \{outA\}\big)\,[outA/outB]$$

where

$$\begin{aligned} BufA_0 &\overset{def}{=} inA \cdot BufA_1 \\ BufA_1 &\overset{def}{=} \overline{outA} \cdot BufA_0 \end{aligned}$$

$$\begin{aligned} BufB_0 &\overset{def}{=} inB \cdot BufB_1 \\ BufB_1 &\overset{def}{=} \overline{outB} \cdot BufB_0 \end{aligned}$$

An equivalent specification written in the fsCCS description language of CCStool2 is

```
{ fsCCS description of the parallel composition }
{ of two 1-place buffers                        }

BufA0 =   inA?  : BufA1
BufA1 =   outA! : BufA0

BufB0 =   inB?  : BufB1
BufB1 =   outB! : BufB0
```

```
start (BufA0, BufB0)
link (outA!, inB?)
invisible (outA!, inB?)
relabel (outB!, outA!)
```

In this description we recognize 4 fsCCS equations and 4 operators, as well as some comment. The operators have the following general meaning:

- The *start* operator defines the starting states of all agents involved. This operator implements the parallel operator $\cdot \mid \cdot$ as defined in [Mil89].
- A *link* operator specifies a pair of actions which are connected. It implements both the relabel operator $\cdot[f]$ and the action complementation operator $\bar{\cdot}$ as defined in [Mil89].
- The *invisible* operator defines which actions are not externally observable. Depending on the computation to be performed, this operator implements the restriction operator $\cdot \setminus \cdot$ as defined in [Mil89] or the hiding operator $\cdot / \cdot$ as defined in [Hoa85].
- The *relabel* operator implements the action relabel operator $\cdot[f]$. `relabel (a, b)` indicates that all actions a must be renamed to b.

All operators may occur more than once in any fsCCS description and have the following properties:

- $\cdots \mathit{operator}_1 \cdots \mathit{operator}_2 \cdots = \cdots \mathit{operator}_2 \cdots \mathit{operator}_1 \cdots$
- $\mathit{start}\,(\ldots, A, \ldots, B, \ldots) = \mathit{start}\,(\ldots, B, \ldots, A, \ldots)$
- $\cdots \mathit{start}\,(A_1, \ldots, A_m) \cdots \mathit{start}\,(B_1, \ldots, B_n) \cdots =$
  $\cdots \mathit{start}\,(A_1, \ldots, A_m, B_1, \ldots, B_n) \cdots\cdots$
- $\mathit{link}\,(a, b) = \mathit{link}\,(b, a)$
- $\mathit{link}\,(a, b) \cdots \mathit{invisible}\,(a, b) =$
  $\mathit{link}\,(a, b) \cdots \mathit{invisible}\,(a) =$
  $\mathit{link}\,(a, b) \cdots \mathit{invisible}\,(b)$
- $\mathit{invisible}\,(\ldots, a, \ldots, b, \ldots) = \mathit{invisible}\,(\ldots, b, \ldots, a, \ldots)$
- $\mathit{invisible}\,(\ldots, a, \ldots, a, \ldots) = \mathit{invisible}\,(\ldots, a, \ldots, \ldots)$
- $\cdots \mathit{invisible}\,(a_1, \ldots, a_m) \cdots \mathit{invisible}\,(b_1, \ldots, b_n) \cdots =$
  $\cdots \mathit{invisible}\,(a_1, \ldots, a_m, b_1, \ldots, b_n) \cdots\cdots$

As shown in figure 2.1, CCStool2 first checks the syntactical correctness of an fsCCS description. This is done using a built-in combination of a lexical scanner, a recursive descent parser, and a compiler. Next, CCStool2 always checks whether this fsCCS description meets several context conditions in the following way:

1. Check whether all information needed to perform the indicated function is present.
2. Verify whether each state S defined as a NIL state (`S = NIL`) is a proper NIL state, which means that S cannot perform any actions.
3. Determine whether each defined start state is actually used in at least one fsCCS equation.
4. If any start states are defined, the corresponding agents are extracted from the fsCCS description. The remaining semantical checks as well as the indicated computation are performed on these extracted agents. If no start states are defined, the original fsCCS description is used.
5. Check whether each linked action is used in at least one fsCCS equation, is not a silent τ action, and is linked to at most one other action.
6. Determine whether each invisible action is used in at least one fsCCS equation, and is not a silent action.
7. Verify whether each action to be relabeled is used in at least one fsCCS equation and is to be relabeled only once. Further, check that the τ action is not involved in any *relabel* operator.

Syntactical and semantical errors are reported by corresponding error messages. If no such errors are encountered, the tool performs the indicated computation. The result of this computation is decompiled into an fsCCS description and written to an output file. Further, the output file contains all, possibly adapted, operators found in the original fsCCS description, except in the following cases:

- action relabeling consumes the *relabel* operators
- abstraction and restriction both consume the *invisible* operators
- expansion consumes the *link* operators and the *invisible* operators, and reduces the amount of start states to one

Comment found in the original fsCCS description is not retained.

2.2 Syntactical and Semantical Check

This function performs the syntactical and semantical checks mentioned in Section 2.1. If no syntactical or semantical errors are encountered, the *original* fsCCS description is written to an output file in the structured form used in the example of Section 2.1. Further, the description is checked for so-called *undefined NIL states*. This are states which only appear on the right-hand side of an fsCCS equation. If such states are encountered, a corresponding warning is issued. In the output file these states appear as proper defined NIL states (see item 2 of semantical checks above).

2.3 State Relabeling

The state relabel function offers the possibility to replace the state names of all agents in an fsCCS description specified by the *start* operators by new state names $\langle Str\rangle_0$, $\langle Str\rangle_1$, $\langle Str\rangle_2$, ..., where $\langle Str\rangle$ denotes a user-definable character string (default the character string Q is used). The start states listed in the *start* operators are also relabeled. If no start states are defined, all states in the fsCCS description are renamed. Next to the output file, a report file can be generated containing a list of all relabeled states and their new names.

2.4 Action Relabeling

Through action relabeling the action names in an fsCCS description can be renamed. Action renamings are defined by the *relabel* operators, which are consumed. All action names contained in other operators are also renamed.

2.5 Extraction

Extraction can be used to extract one or more agents from one or more larger agents. The *start* operator specifies the agents to be extracted. Given such a start state, extraction searches for all reachable states.

2.6 Abstraction

Given an agent P and a set of actions L, one can compute the abstracted agent P/L, where $\cdot/\cdot$ is the hiding operator as defined in [Hoa85]. The actions to be abstracted are defined by the *invisible* operators, which are consumed.

2.7 Restriction

Given an agent P and a set of actions L, restriction offers the functionality to compute the restricted agent $P \setminus L$, where $\cdot \setminus \cdot$ is the restrict operator as defined in [Mil89]. The difference between hiding and restricting can intuitively be seen as follows:

- Hiding a set of actions L *permits* the actions in L *to occur unobserved*, which basically amounts to replacing them by the silent τ action in the fsCCS description.
- Restricting a set of actions L *prevents* the actions in L *from occurring*. This basically amounts to removing them from the fsCCS description.

The actions to be restricted are specified using the *invisible* operators.

2.8 Expansion

This function implements the expansion theorem as defined in [Mil89]. In the context of expansion the operators *start*, *link*, and *invisible* have the following meaning:

- The *start* operators are used to indicate the initial states of all agents involved in the expansion process.
- The *link* operators specify a pair of connected actions. If agents P and Q can perform actions p and q respectively, and if these actions are linked, P and Q *may* synchronize on these actions if p and q are not restricted, otherwise they have to synchronize, i.e., they may not perform actions p and q separately. Such a synchronization results in a silent (τ) action.
- The *invisible* operators define which actions are not externally observable. If an fsCCS description contains no *invisible* operators, it is implicitly assumed that all actions are observable.

As an example, the expansion of the fsCCS specification of the two 1-place buffers in Section 2.1 will result in the following fsCCS description:

```
BufA0|BufB0 =   inA?  : BufA1|BufB0
BufA1|BufB0 =   tau   : BufA0|BufB1
BufA0|BufB1 =   inA?  : BufA1|BufB1
              + outB! : BufA0|BufB0
BufA1|BufB1 =   outB! : BufA1|BufB0

start (BufA0|BufB0)

relabel (outB!, outA!)
```

Here, `tau` denotes the silent τ action. Note that the *relabel* operator is retained. After performing an explicit action relabeling, we get the following result:

```
BufA0|BufB0 =   inA?  : BufA1|BufB0
BufA1|BufB0 =   tau   : BufA0|BufB1
BufA0|BufB1 =   inA?  : BufA1|BufB1
              + outA! : BufA0|BufB0
BufA1|BufB1 =   outA! : BufA1|BufB0

start (BufA0|BufB0)
```

2.9 Minimization Modulo Observational Equivalence

CCStool2 can be used to compute a minimal agent, with respect to the amount of states and transitions, which is observational equivalent ($\approx$) to a given agent. The *start* operator specifies the agents which have to be minimized. All agents which are not specified in this way are left out of the resulting minimal fsCCS description. If no start state is defined, it is assumed that all agents have to be minimized. Next to the output file, a report file can be generated containing a so-called state-partition table. This table indicates which states are observational equivalent. It also indicates which states are used as representatives (of the state equivalence classes) in the resulting fsCCS description.

Consider, for example, the expanded and relabeled buffers of Section 2.8. The minimization of this example results in the following fsCCS description:

```
BufA0|BufB0 =   inA?  : BufA1|BufB0
BufA1|BufB0 =   inA?  : BufA1|BufB1
              + outA! : BufA0|BufB0
BufA1|BufB1 =   outA! : BufA1|BufB0

start (BufA0|BufB0)
```

and in the following state-partition table:

```
State Partition Table

BufA1|BufB0 <- <BufA1|BufB0, BufA0|BufB1>
NIL         <- <NIL>
BufA1|BufB1 <- <BufA1|BufB1>
BufA0|BufB0 <- <BufA0|BufB0>
```

This table shows, for example, that state `BufA1|BufB0` is equivalent to state `BufA0|BufB1`, and that it is used to represent the equivalence class `<BufA1|BufB0,BufA0|BufB1>` in the resulting fsCCS description.

2.10 Minimization Modulo Strong Equivalence

Using this type of minimization one can compute a minimal agent, with respect to the amount of states and transitions, which is strong equivalent ($\sim$) to a given agent. Except for the fact that this functionality deals with strong equivalence instead of observational equivalence, it is completely equivalent to the functionality described in the previous section.

2.11 Reduction Modulo Observational Equivalence

Minimization modulo observational equivalence can be used to compute a *minimal* agent which is observational equivalent to a given agent. The worst-case complexity of this type of minimization is of order $\mathcal{O}\left(|Q|^3\right)$, where $|Q|$ denotes the amount of states of the agent to be minimized. Next to minimization, CCStool2 offers the possibility for *reduction.* Reducing an agent means computing an agent which is observational equivalent but has fewer states and transitions. Reduction basically consists of deletion of all strongly-connected τ-components and of single-τ transitions (see Section 3.4), and is performed in *linear time.*

For example, reconsider the expanded and relabeled buffers of Section 2.8. The reduction of this example results in the following fsCCS description:

```
BufA0|BufB0 =   inA?  : BufA0|BufB1
BufA0|BufB1 =   inA?  : BufA1|BufB1
              + outA! : BufA0|BufB0
BufA1|BufB1 =   outA! : BufA0|BufB1

start (BufA0|BufB0)
```

and in the following state-partition tables:

```
State Partition Table of Strongly-Connected Tau-Components Reduction

BufA1|BufB1 <- <BufA1|BufB1>
BufA1|BufB0 <- <BufA1|BufB0>
BufA0|BufB1 <- <BufA0|BufB1>
BufA0|BufB0 <- <BufA0|BufB0>
NIL         <- <NIL>

State Partition Table of Single-Tau Reduction

NIL         <- <NIL>
BufA1|BufB1 <- <BufA1|BufB1>
BufA0|BufB1 <- <BufA0|BufB1, BufA1|BufB0>
BufA0|BufB0 <- <BufA0|BufB0>
```

Note that in this case *two* state-partition tables are generated. The first is a result of the reduction modulo strongly-connected τ-components and the second is a result of the single-τ reduction applied to the agent which is previously reduced modulo strongly-connected τ-components. Both tables can easily be combined into a single partition table by substituting all state names in the right-hand side of the second table by the state names of the corresponding equivalence classes in the right-hand side of the first partition table.

In the case of the expanded and relabeled buffers, reduction and minimization (modulo observational equivalence) yield similar results (except for the fact that different state representatives are used). Note that, in general, this will not be the case.

2.12 Observational Equivalence Verification

CCStool2 also supports the possibility to verify whether two agents are observational equivalent. The original fsCCS description must contain *start* operators defining exactly 2 start states indicating the agents to be verified. The equivalence verification results in a positive or a negative answer. Further, a report file containing the state-partition table can be generated (see Section 2.9).

Assume, for example, that we would like to verify whether the expanded and relabeled version of the buffers in Section 2.8 is equivalent to the minimal version in Section 2.9. This problem can be specified in terms of an fsCCS description as follows:

```
BufA0|BufB0_1 =   inA?  : BufA1|BufB0_1
BufA1|BufB0_1 =   tau   : BufA0|BufB1_1
BufA0|BufB1_1 =   inA?  : BufA1|BufB1_1
                + outA! : BufA0|BufB0_1
BufA1|BufB1_1 =   outA! : BufA1|BufB0_1

BufA0|BufB0_2 =   inA?  : BufA0|BufB1_2
BufA0|BufB1_2 =   inA?  : BufA1|BufB1_2
                + outA! : BufA0|BufB0_2
BufA1|BufB1_2 =   outA! : BufA0|BufB1_2

start (BufA0|BufB0_1, BufA0|BufB0_2)
```

Equivalence verification results in the following message from CCStool2:

```
Agents BufA0|BufB0_1 and BufA0|BufB0_2 are observational equivalent
```

The resulting state-partition table is as follows:

```
State Partition Table

BufA1|BufB0_1 <- <BufA1|BufB0_1, BufA0|BufB1_1, BufA1|BufB0_2>
NIL           <- <NIL>
BufA1|BufB1_1 <- <BufA1|BufB1_1, BufA1|BufB1_2>
BufA0|BufB0_1 <- <BufA0|BufB0_1, BufA0|BufB0_2>
```

2.13 Strong Equivalence Verification

This functionality is equivalent to the functionality described in the previous section, except for the fact that it deals with strong equivalence instead of observational equivalence.

2.14 Combined Functions

CCStool2 provides the possibility to combine the reduction function described in Section 2.11 with

- **expansion**
  In this case reduction is used as a *postprocessing* step. First, the agents are expanded as described in Section 2.8. Next, the resulting agent is reduced. The fsCCS description of the reduced agent is written to an output file and optionally the resulting state-partition tables are written to a report file.
- **minimization modulo observational equivalence**
  Now reduction is used as a *preprocessing* step. The agents to be minimized are first reduced and optionally the resulting state-partition tables are written to a report file. Next, the reduced agents are minimized as described in Section 2.9. The fsCCS description of the minimized agents is written to an output file and the resulting state-partition table can be appended to the (already existing) report file.
- **observational equivalence verification**
  In this case reduction is also used as a *preprocessing* step. The agents to be verified are first reduced and optionally the resulting state-partition tables are written to a report file. Next, the reduced agents are verified as described in Section 2.12. The resulting state-partition table can appended to the (already existing) report file.

Further, it is possible to combine expansion with reduction and/or with either observational or strong minimization as a postprocessing step. The fsCCS description of the minimized agent is written to an output file and optionally the resulting state-partition tables are written to a report file.

Chapter 3

Algorithms

In Chapter 2 we described the functionality of CCStool2. In this chapter we discuss the core algorithms which implement the following functions:

- minimization modulo observational equivalence
- minimization modulo strong equivalence
- reduction modulo observational equivalence
- observational equivalence verification
- strong equivalence verification

Section 3.1 describes the algorithm used to verify whether two agents are strong equivalent. Section 3.2 presents the algorithm used to verify observational equivalence. Section 3.3 describes two minimization algorithms: one for minimization modulo observational equivalence and one for minimization modulo strong equivalence. Section 3.4 discusses two algorithms for reduction modulo observational equivalence.

3.1 An Algorithm for Strong Equivalence Verification

Strong equivalence is defined on *transition graphs. Vertices* in these graphs correspond to the *states* a system may reach during execution, with one vertex being distinguished as the *start state.* The *edges*, which are directed, are labeled with the actions and represent the *state transitions* a system may undergo. The formal definition is as follows:

Definition 3.1
A transition graph is a quadruple $\langle Q, q, Act, \rightarrow \rangle$, where

- Q is a set of states

- $q \in Q$ is the start state
- Act is a set of actions
- $\rightarrow \subseteq Q \times Act \times Q$ is a derivation relation

□

We shall often write $q_1 \xrightarrow{a} q_2$ to indicate that there is a edge labeled a from state q_1 to state q_2. When a graph does not have a start state indicated, we shall refer to the corresponding triple as a *transition system.* A transition graph may contain a number of *strong-equivalent states.* Two states in a transition system are strong equivalent if there exists a *strong bisimulation* relating them. The formal definition of a strong bisimulation is as follows:

Definition 3.2
Let $\langle Q, Act, \rightarrow\rangle$ be a transition system. Then a relation $R \subseteq Q \times Q$ is a strong bisimulation if R is symmetric and whenever $q_1\ R\ q_2$, the following holds:

- If $q_1 \xrightarrow{a} q_1'$ then there is a q_2' such that $q_2 \xrightarrow{a} q_2'$ and $q_1'\ R\ q_2'$.

□

When states q_1 and q_2 are strong equivalent we often write $q_1 \sim q_2$.

Let $G_1 = \langle Q_1, q_1, Act, \rightarrow_1\rangle$ and $G_2 = \langle Q_2, q_2, Act, \rightarrow_2\rangle$ be two transition graphs satisfying $Q_1 \cap Q_2 = \emptyset$. G_1 and G_2 are strong equivalent exactly if the two start states q_1 and q_2 are strong equivalent in the transition system $\langle Q_1 \cup Q_2, Act, \rightarrow_1 \cup \rightarrow_2\rangle$.

The most popular technique to verify strong equivalence is based on *partition-refinement algorithms* [Kor91]. To decide whether two graphs T_1 and T_2 are strong equivalent, one first takes the union T of these graph. Next, a partition-refinement algorithm is applied to this graph T. If the two start states of T_1 and T_2 appear in the same equivalence class of the final partition of T, one can conclude that the graphs T_1 and T_2 are strong equivalent.

The partition-refinement algorithm used in CCStool2 is based on the implementation of the Kanellakis-Smolka algorithm [KS83] as described in [Bou92]. This implementation exploits the fact that an equivalence relation on a set of states may be viewed as a *partition,* or as a set of pairwise-disjoint subsets called *blocks*, of the set of states, whose union is the set of states. In this representation a block corresponds to an *equivalence class.* Two states are equivalent exactly if they belong to the same block. Starting with the partition containing one block representing the trivial equivalence relation consisting of one equivalence class, the algorithm repeatedly *refines* this partition by *splitting* blocks, until the associated equivalence class becomes a *bisimulation.* To determine whether the partition needs further refining, the algorithm looks at each block in turn. If a state in a block B has an a-derivative in a block B' and another state in B does not, the algorithm splits B into two blocks, one containing the states having an a-derivative in B' and the other containing the states that do not. A block B' against which other blocks can be split is called *unstable* in the corresponding partition. If no blocks can be split against a certain block B'', B'' is called *stable*. If the algorithm cannot split any blocks anymore, the resulting equivalence relation corresponds exactly to bisimulation equivalence on the given transition system. The abstract algorithm is shown in figure 3.1.

```
procedure $Bisim\,(Q, Act, \rightarrow)$
[
      $StableBlocks := \langle\rangle$
    ; $UnstableBlocks := \langle Q \rangle$
    ; do $UnstableBlocks \neq \langle\rangle$
      $\rightarrow$
            $B := First\,(UnstableBlocks)$
          ; $Delete\,(B, UnstableBlocks)$
          ; $AddFirst\,(B, StableBlocks)$
          ; $CheckUnstableBlock\,(B, StableBlocks, UnstableBlocks)$
      od
]
```

Figure 3.1: Algorithm of procedure *Bisim*

Throughout the algorithm we maintain two lists of blocks. One is called *StableBlocks* which contains the blocks with respect to which each block of the current partition is stable. The other one, called *UnstableBlocks*, collects the blocks with respect to which stability still has to be checked for each block of the partition. Initially, *StableBlocks* is empty and *UnstableBlocks* contains the unique block of the initial partition. The loop of the algorithm repeatedly takes a block B from *UnstableBlock* and transfers it temporarily to *StableBlocks*. Then the procedure *CheckUnstableBlock* takes each block of the current partition and checks whether block B is stable with respect to those blocks. If B is not stable with respect to a certain block C, C is split into C_1 and C_2 which are both added to *UnstableBlocks*. Procedure *CheckUnstableBlock* is shown in figure 3.2.

The function *sort* used in the procedure *CheckUnstableBlocks* determines all actions a with respect to a block B for which there exists a transition labeled a with a target state inside B, or formally

$$sort\,(B) = \left\{ a \in Act \mid \exists\, s \in Q\, \exists\, s' \in B : s \xrightarrow{a} s' \right\}$$

```
procedure $CheckUnstableBlock\,(B, StableBlocks, UnstableBlocks)$
[[
  for all $a \in sort(B)$ and while $first(StableBlocks) = B$
  $\rightarrow$
        $BlockSet := \{Bl \mid (Bl\ \underline{in}\ StableBlocks \vee Bl\ \underline{in}\ UnstableBlocks) \wedge$
                  $(\exists\, s \in Bl\, \exists\, s' \in B : s \xrightarrow{a} s')\}$
     ;  $Refine\,(B, BlockSet, StableBlocks, UnstableBlocks, a)$
  rof
]]
```

Figure 3.2: Algorithm of procedure *CheckUnstableBlock*

```
procedure $Refine\,(B, BlockSet, StableBlocks, UnstableBlocks, a)$
[[
  do $BlockSet \neq \emptyset \wedge first\,(StableBlock) = B$
  $\rightarrow$
        $C :\in BlockSet$
     ;  if $splittable\,(C, a, B)$
        $\rightarrow$
              $C_1 := \{s \in C \mid \exists\, s' \in B : s \xrightarrow{a} s\}$
           ;  $C_2 := C \setminus C_1$
           ;  $delete\,(C, StableBlocks)$
           ;  $delete\,(C, UnstableBlocks)$
           ;  $add\,(C_1, UnstableBlocks)$
           ;  $add\,(C_2, UnstableBlocks)$
        fi
     ;  $BlockSet := BlockSet \setminus \{C\}$
  od
]]
```

Figure 3.3: Algorithm of procedure *Refine*

Procedure *CheckUnstableBlock* is a loop over the *sort* of B while it does not appear as an unstable block (with respect to itself). Inside the loop a set of blocks *BlockSet* is determined. This set contains all blocks in *StableBlocks* or *UnstableBlocks* from which there exists an outgoing transition labeled a with a target state inside B. These blocks are potential candidates to be split against block B. After *BlockSet* is calculated, all blocks in this set are refined in procedure *Refine* which is shown in figure 3.3.

Procedure *Refine* loops over all blocks in *BlockSet* while B is still stable. In the body of the loop an element C of *BlockSet* is taken. Function *splittable* checks whether C can really be split against B. This is the case if block C contains a state with an a-derivative in block B and another state with an a-derivative not in B, or formally

$$\mathit{splittable}\,(C, a, B) = \left(\exists\, s \in C\, \exists\, s' \in B : s \xrightarrow{a} s'\right) \wedge \left(\exists\, s \in C\, \exists\, s' \notin B : s \xrightarrow{a} s'\right)$$

If C is splittable, it is split into sets C_1 and C_2. C_1 contains all states of C having an a-derivative in B and C_2 contains all other states of C. After this, C is deleted either from *UnstableBlocks* or from *StableBlocks*. Both sets C_1 and C_2 are added to *UnstableBlocks*.

The worst-case complexity of algorithm *Bisim* is $\mathcal{O}\left(|\!\rightarrow\!| * |\; Q \;|\right)$. A more sophisticated partition-refinement algorithm has been developed by Paige and Tarjan [PT87]. This latter algorithm has a worst-case complexity of order $\mathcal{O}\left(|\!\rightarrow\!| * \mathit{log} \;|\; Q \;|\right)$, but only becomes practical in the case of large graphs. For graphs of a normal size it has too much overhead [Kor91]. Further, there is not yet enough evidence to suggest that it is appreciably faster in practice [CPS93].

3.2 An Algorithm for Observational Equivalence Verification

Observational equivalence is one of the most-used behavioral equivalences to verify concurrent systems. Observational equivalence on a transition graph is defined as follows:

Definition 3.3
Let $\langle Q, \mathit{Act}, \rightarrow\rangle$ be a transition system.

- If $t \in \mathit{Act}^*$, then $\hat{t} \in (\mathit{Act} \setminus \{\tau\})^*$ is the sequence obtained by deleting all occurrences of τ from t.
- If $t = \alpha_1 \cdots \alpha_n \in \mathit{Act}^*$, then $s \overset{t}{\Longrightarrow} s'$ if $s(\xrightarrow{\tau})^* \xrightarrow{\alpha_1} (\xrightarrow{\tau})^* \cdots (\xrightarrow{\tau})^* \xrightarrow{\alpha_n} (\xrightarrow{\tau})^* s'$.
- A binary relation $R \subseteq Q \times Q$ is a weak bisimulation, if R is symmetric and whenever $q_1 \; R \; q_2$: if $q_1 \xrightarrow{a} q_1'$ then for some q_2', $q_2 \overset{\hat{a}}{\Longrightarrow} q_2'$ and $q_1' \; R \; q_2'$.

□

As in the case of strong equivalence, two states in a transition system are observational equivalent if there is a *weak bisimulation* relating them. If states q_1 and q_2 are observational equivalent, we often write $q_1 \approx q_2$. Verifying whether two transition graphs are observational equivalent is done in the same way as in the case of strong equivalence. It is not hard to proof that the definition of a weak-bisimulation relation is equivalent to

Definition 3.4
Let $\langle Q, Act, \rightarrow \rangle$ be a transition system. A binary relation $R \subseteq Q \times Q$ is a weak bisimulation if R is symmetric, and whenever $q_1 \; R \; q_2$

- if $q_1 \overset{\hat{a}}{\Longrightarrow} q_1'$ then, for some q_2' $q_2 \overset{\hat{a}}{\Longrightarrow} q_2'$ and $q_1' \; R \; q_2'$

□

According to this definition we can decide whether two states q_1 and q_2 in a transition system $\langle Q, Act, \rightarrow \rangle$ are observational equivalent in the following way:

- Construct relation $\Longrightarrow \subseteq Q \times \widehat{Act} \times Q$. This relation is called the *reflexive-transitive closure* of $\rightarrow$. An algorithm for constructing $\Longrightarrow$ is described below.
- Decide whether q_1 and q_2 are *strong* equivalent in transition system $\langle Q, \widehat{Act}, \Longrightarrow \rangle$ where $\widehat{Act} = \{\hat{a} \mid a \in Act\} = \{\varepsilon\} \cup Act \setminus \{\tau\}$. This can be decided using the algorithms described in Section 3.1.

Our algorithm for calculating relation $\Longrightarrow$ is based upon the following property:

Property 3.1
$\Longrightarrow \; = \; \overset{\varepsilon}{\Longrightarrow} \cup \overset{\varepsilon}{\Longrightarrow} \rightarrow \overset{\varepsilon}{\Longrightarrow}$

□

The algorithm consists of two procedures called *τ-Closure* and *Closure*. Procedure *τ-Closure* (shown in figure 3.4) calculates, given a transition system $\langle Q, Act, \rightarrow \rangle$, the corresponding relation $\overset{\varepsilon}{\Longrightarrow}$. Procedure *Closure* (shown in figure 3.5) takes as input a transition system $\langle Q, Act, \rightarrow \rangle$ together with relation $\overset{\varepsilon}{\Longrightarrow}$ and computes $\Longrightarrow$.

The procedure *τ-Closure* loops over all states of the transition system. For every state s it is determined which states can be reached, starting in s and by performing only (zero or more) τ steps. To establish this two sets *StatesVisited* and *StatesToVisit* are maintained. *StatesVisited* contains all states which have already been inspected. States which still have to be visited are stored in *StatesToVisit*. This algorithm is a variant of the famous *Single Source Shortest Path* algorithm of Edger W. Dijkstra [AHU87]. The worst-case complexity of *τ-Closure* is $\mathcal{O}\left(\mid Q \mid^3\right)$.

```
procedure $\tau$ - Closure $(Q, Act, \rightarrow)$
[
      $\overset{\varepsilon}{\Longrightarrow} := \varnothing$
   ;  for $s \in Q$
      $\rightarrow$
            StatesToVisit $:= \{s\}$
         ;  StatesVisited $:= \varnothing$
         ;  do StatesToVisit $\neq \varnothing$
            $\rightarrow$
                  $s'$ $:\in$ StatesToVisit
               ;  $\overset{\varepsilon}{\Longrightarrow} := \overset{\varepsilon}{\Longrightarrow} \cup \{(s, \varepsilon, s')\}$
               ;  StatesToVisit := StatesToVisit $\cup \{s'' \in Q \mid s' \xrightarrow{\tau} s''\} \setminus$
                                   (StatesToVisit $\cup$ StatesVisited)
               ;  StatesToVisit := StatesToVisit $\setminus \{s'\}$
               ;  StatesVisited := StatesVisited $\cup \{s'\}$
            od
      rof
]
```

Figure 3.4: Algorithm of procedure τ – *Closure*

The first main loop of *Closure* calculates relation $\Longrightarrow' = \overset{\varepsilon}{\Longrightarrow} \cup \overset{\varepsilon}{\Longrightarrow} \rightarrow$. For every state s it determines every state s' which is reachable (with respect to $\rightarrow$) from s by only performing τ steps. After that all transitions of the form $s' \xrightarrow{a} s''$ are determined and finally transition (s, a, s'') is added to $\Longrightarrow'$. The second part of the algorithm uses $\Longrightarrow'$ to calculate relation $\Longrightarrow$. It operates in a similar way as the first part.

Procedure *Closure* has a worst-case complexity of order $\mathcal{O}\left(\mid Q \mid^3\right)$ and thus the complexity of calculating the reflexive-transitive closure $\Longrightarrow$ is $\mathcal{O}\left(\mid Q \mid^3\right)$. In the literature there are numerous sub-cubic reflexive-transitive closure algorithms which work in $\mathcal{O}\left(\mid Q \mid^{2.49}\right)$ or less [Kor91]. However, these algorithms tend to be practical only for large values of $\mid Q \mid$ (i.e., $\mid Q \mid > 1000$) [BS87].

We conclude this section by noting that, by using the algorithms described in this section and in the previous section, observational equivalence can be decided in worst-case time $\mathcal{O}\left(\mid Q \mid^3\right)$.

```
procedure $Closure\left(Q, Act, \rightarrow, \overset{\varepsilon}{\Longrightarrow}\right)$
[
        $\Longrightarrow' := \overset{\varepsilon}{\Longrightarrow}$
    ;   for $s \in Q$
        $\rightarrow$
          for $s' \in \{t \in Q \mid s \overset{\varepsilon}{\Longrightarrow} t\}$
          $\rightarrow$
            for$(a, s'') \in \{(b, t) \mid s' \xrightarrow{b} t \wedge b \neq \tau\}$
            $\rightarrow$
              $\Longrightarrow' := \Longrightarrow' \cup \{(s, a, s'')\}$
            rof
          rof
        rof
    ;   $\Longrightarrow := \overset{\varepsilon}{\Longrightarrow}$
    ;   for $s \in Q$
        $\rightarrow$
          for$(a, s') \in \{(b, t) \mid s \overset{b}{\Longrightarrow}' t \wedge b \neq \tau\}$
          $\rightarrow$
            for $s'' \in \{t \in Q \mid s' \overset{\varepsilon}{\Longrightarrow} t\}$
            $\rightarrow$
              $\Longrightarrow := \Longrightarrow \cup \{(s, a, s'')\}$
            rof
          rof
        rof
]
```

Figure 3.5: Algorithm of procedure *Closure*

3.3 Two Minimization Algorithms

Algorithms which compute state partitions or bisimulation relations are often used as a *preprocessing step* in the computation of minimal transition graphs. A minimal graph is equivalent to the original graph and is minimal in states and edges.

Let $\langle Q, q, Act, \rightarrow\rangle$ be a transition graph. An algorithm for constructing a minimal graph $\langle Q', q', Act', \rightarrow'\rangle$ which is strong equivalent to the original is the following:

- The set of actions Act' becomes equal to the set of actions Act.

- Calculate the state partition P of transition system $\langle Q, Act, \rightarrow\rangle$. The blocks of P become the states of Q'.
- The start state q' becomes the unique block B for which $q \in B$ holds.
- If B_1 and B_2 are blocks in P, $s_{B_1} \in B_1$, $s_{B_2} \in B_2$ and if $s_{B_1} \xrightarrow{a} s_{B_2}$, then add triple (B_1, a, B_2) to relation $\rightarrow'$.

Computing a graph which is minimal modulo observational equivalence is done in a similar way. The algorithm is as follows:

- The set of actions Act' becomes equal to the set of actions Act.
- Calculate the state partition P of transition system $\langle Q, \widehat{Act}, \Longrightarrow\rangle$ (as defined in Section 3.2). The blocks of P become the states of Q'.
- The start state q' becomes the block B for which $q \in B$ holds.
- If B_1 and B_2 are blocks in P, $s_{B_1} \in B_1$, $s_{B_2} \in B_2$ and if $s_{B_1} \xrightarrow{a} s_{B_2}$ and *if* $B_1 \neq B_2$ *and* $a \neq \tau$, then add triple (B_1, a, B_2) to relation $\rightarrow'$.

Note the differences between the last parts of both algorithms. Because observational equivalence "ignores" τ-loops, i.e., $B \approx B + \tau : B$, these are never added to relation $\rightarrow'$. In the case of strong equivalence τ-loops may not be left out the relation.

3.4 Two Algorithms for Reduction Modulo Observational Equivalence

As described in the previous section, minimization modulo observational equivalence and verification modulo observational equivalence are based upon the following two algorithms:

- an algorithm for calculating the reflexive-transitive τ-closure with a worst-case complexity of $\mathcal{O}\left(\mid Q \mid^3\right)$
- a partition-refinement algorithm with a worst-case complexity of $\mathcal{O}\left(\mid\rightarrow\mid * \mid Q \mid\right)$

The partition-refinement algorithm is applied to the reflexive transitive τ-closure of the transition system for which a state partition has to be calculated. Such a closure can become extremely large, especially if the transition system contains a lot of τ-transitions. For example, consider transition system $T = \langle Q, Act, \rightarrow\rangle$ defined as follows:

- $Q = \{Q_1, \cdots, Q_{1000}\}$
- $Act = \{\tau\}$

- $\rightarrow = \{(Q_i, \tau, Q_{i+1}) \mid 1 \leq i \leq 1000\} \cup \{(1000, \tau, 1)\}$

This transition system T has 1000 states and 1000 transitions. The reflexive-transitive τ-closure of T, however, has as many as 1000 states and 1000000 transitions! This is due to the fact that T has a large τ-cycle $Q_1, Q_2, \cdots, Q_{1000}$. In general, transitions systems which have (large) τ-cycles may have *huge* reflexive-transitive τ-closures. This can make minimization modulo observational equivalence and verification modulo observational equivalence very expensive in both time and space.

This section describes two algorithms which reduce the amount of τ-transitions as well as the amount of states of a transition system in *linear time*. The first algorithm deletes so-called *strongly-connected τ-components*. The second algorithm deletes so-called *single-τ chains*. Both algorithms reduce modulo observational equivalence.

3.4.1 An Algorithm for Reduction Modulo Strongly-Connected τ-Components

We will start with the definition of an equivalence relation $\asymp$.

Definition 3.5
Let $T = \langle Q, Act, \rightarrow\rangle$ be a transition system. We define relation $\asymp \subseteq Q \times Q$ as follows:

$$q_1 \asymp q_2 \text{ if and only if } q_1(\xrightarrow{\tau})^* q_2 \text{ and } q_2(\xrightarrow{\tau})^* q_1$$

□

Two states q_1 and q_2 are related by relation $\asymp$ if and only if there exists a path consisting of τ-transitions from q_1 to q_2 as well as from q_2 to q_1. It is easy to proof that $\asymp$ is an equivalence relation. This implies that $\asymp$ induces a partitioning of the set of states of transition system T. If $q \in Q$ we will denote the equivalence class induced by q as

$$[q]_{\asymp} = \{r \in Q \mid q \asymp r\}$$

We will call such an equivalence class a *strongly-connected τ-component*, since such a class is a strongly-connected component of the transition system T obtained by deleting from T all non-τ transitions, i.e., all transitions which are not labeled with a τ.

The algorithm is based upon the following property:

Property 3.2
Let $\langle Q, Act, \rightarrow\rangle$ be a transition system and let $q \in Q$. Then for all $q \in Q$ the following holds:

$$\forall q_1, q_2 \in [q]_{\asymp} : q_1 \approx q_2$$

□

This property implies that, given a transition graph $T = \langle Q, q, Act, \rightarrow \rangle$, we can construct a reduced observational-equivalent transition-graph $T' = \langle Q', q', Act, \rightarrow' \rangle$ using the following algorithm:

- The set of actions Act' becomes equal to the set of actions Act.
- Calculate the state partition P induced by relation $\asymp$ of transition system T. The blocks of P become the states of Q'.
- The start state q' becomes the unique block B for which $q \in B$ holds.
- If B_1 and B_2 are blocks in P, $s_{B_1} \in B_1$, $s_{B_2} \in B_2$ and if $s_{B_1} \xrightarrow{a} s_{B_2}$, then add triple (B_1, a, B_2) to relation $\rightarrow'$.

The calculation of the state partition P of a transition system $T = \langle Q, q, Act, \rightarrow \rangle$ amounts to the calculation of all strongly-connected τ-components of T which are obtained by deleting all non-τ transitions from T. This can be performed by the well-known *linear-time* algorithm *Strongconnect.* This algorithm is based upon depth-first search and was first exploited by Tarjan [Tar72].

3.4.2 An Algorithm for Reduction Modulo Single-τ Chains

The main idea behind this algorithm is that, if two states are connected by precisely one τ-transition (are "single-τ connected"), those states are observational equivalent. In general, all states in a chain of pairwise single-τ connected states are observational equivalent. To characterize such chains we will define an ordering relation $\sqsubseteq^*$.

Definition 3.6
Let $\langle Q, Act, \rightarrow \rangle$ be a τ-loop free transition system, i.e., $\forall q \in Q : \neg\, q(\xrightarrow{\tau})^+ q$.

□

We define relation $\sqsubseteq \subseteq Q \times Q$ as

$$q_1 \sqsubseteq q_2 \text{ iff } q_1 \xrightarrow{\tau} q_2 \text{ and } \forall \alpha \in Act:\ \forall q \in Q : \text{if } q_1 \xrightarrow{\tau} q \text{ then } \alpha = \tau \text{ and } q = q_2$$

So, $q_1 \sqsubseteq q_2$ if and only if q_1 has precisely one outgoing transition, which is labeled with a τ action and which ends in q_2.

We will write $\sqsubseteq^*$ to denote the *reflexive-transitive closure* of $\sqsubseteq$. It is not hard to proof that $\sqsubseteq^*$ is an ordering and thus that $(Q, \sqsubseteq^*)$ is a *POSET* (Partially-Ordered SET).

The algorithm is based upon the following property:

Property 3.3
Let $\langle Q, Act, \rightarrow \rangle$ be a τ-loop-free transition system, let M be the set of maximal elements of the corresponding *POSET* $(Q, \sqsubseteq^*)$, and define $[m] = \{q \sqsubseteq^* m \mid q \in Q\}$ for all $m \in M$, then

- $\{[m] \mid m \in M\}$ is a partitioning of Q, and
- for all $m \in M$ for all $q_1, q_2 \in [m]$, q_1 and q_2 are observational equivalent

□

Ordering relation $\sqsubseteq^*$ partitions the set of states Q into a set of observational-equivalence classes. This set of equivalence classes is induced by the set of maximal elements of *POSET* $(Q, \sqsubseteq^*)$. A class $[m]$ is called a single-τ chain because $[m]$ is a chain and all states in $[m]$ are pairwise single-τ connected.

```
procedure Single − τ partitioning (Q, Act, →)
[
      ⇛ := ∅
    ; maximal_element_list := ε
    ; for q₁ ∈ Q
      →
        if there exists a q₂ such that q₁ ⊑ q₂
          then ⇛ := ⇛ ∪ {(q₂, τ, q₁)}
          else Add (q₁, maximal_element_list)
        fi
      rof
    ; P := ∅
    ; do maximal_element_list ≠ ε
      →
            m := First (maximal_element_list)
          ; Delete (m, maximal_element_list)
          ; S := Reachable (m, ⇛)
          ; Add (S, P)
      od
]
```

Figure 3.6: Algorithm of procedure *Single* $-$ τ *partitioning*

The state partitioning P can be calculated in *linear time* using the algorithm shown in figure 3.6. The first part of the algorithm constructs an "inverse"-τ relation $\Rrightarrow$ such that $q_1 \Rrightarrow q_2$ if and only if $q_2 \xrightarrow{\tau} q_1$ for all $q_1, q_2 \in Q$. Also a list *maximal_element_list* of all maximal elements of *POSET* $(Q, \sqsubseteq)$ is built which is used in the second part of the algorithm. In this second part, for every maximal element m in the list, the set of states which are reachable from m in relation $\Rrightarrow$ is calculated using a variant of Dijkstra's *Single Source Shortest Path* algorithm [AHU87] and assigned to S. It is easy to see that this set S is equal to set $[m]$.

Note that the algorithm only operates properly if the transition system contains no τ-loops, for example, after the strongly-connected τ-components have been removed.

Chapter 4

Implementation

As mentioned in Chapter 1, CCStool2 should be efficient in both time and space. The use of fast algorithms such as those described in Chapter 3 is a necessary but not sufficient condition to be fulfilled to be time efficient. The complementary condition is to use combinations of a data representation and a data structure storing this representation which allow for a fast data manipulation as needed by these algorithms. At the same time these algorithms as well as both these data representations and these data structures should be space efficient. Since in practice space efficiency is more important than time efficiency [Kor91], special attention has been given to the former. Last but not least, the implementation language and environment should provide support for both these efficiencies.

Section 4.1 describes the representation and storage of data within CCStool2. Section 4.2 discusses the choice of implementation language and environment.

4.1 Data Representation and Storage

This section presents three basic data structures supporting both a fast data manipulation and an efficient memory usage on which the data structures used in CCStool2 are based. This section also describes the representation and storage of the two main data objects in CCStool2: CCS agents, and the state partitions as used in the partition-refinement algorithm discussed in Section 3.1. Further, this section presents the so-called *skip list.* Skip lists are used in the representation and storage of CCS agents to improve the search for the names of states and actions. This data structure has been introduced only recently as an alternative to balanced trees and self-adjusting trees. Skip-list algorithms are simpler, and therefore easier to implement, than, and provide significant constant-factor speed-improvements over balanced-tree and self-adjusting-tree algorithms.

Subsection 4.1.1 discusses the three basic data structures. Subsections 4.1.2 describes the representation and storage of CCS agents. Subsection 4.1.3 presents the skip list. Subsection 4.1.4 describes the representation and storage of the state partition.

4.1.1 Basic Data Structures

The data structures used in CCStool2 are based on three basic data structures each of which support both a fast data manipulation and an efficient memory usage. The first basic data structure is used when the amount of data to be stored is known and consists of a *dynamically allocated one-dimensional array* with a large enough size.

The other two basic data structures are used when the amount of data to be stored in not known. There are several solutions to solve the problem of storing an unknown amount of data. The use of static arrays with a "large enough" predefined size does not provide a solution to this problem for apparent reasons.

An obvious solution is to use a *list-type data structure*, but this creates the following new problems:

- Manipulation of list-type data structures tends to be time consuming.
- List-type data structures can be very inefficient with respect to the memory usage for the following reasons:
  - Each data item usually is stored in a separate memory block. Besides the item itself, a memory block also contains data needed by the memory-management scheme with a typical size of four or eight bytes.
  - Each item contains a pointer to the next item.

  This inefficiency especially applies if data items with a size of only a few bytes, such as booleans or words, need to be stored.

Another solution is to use a *dynamically (re)allocated one-dimensional array*: One allocates an array which can store a certain amount of data. When the array is filled up, it is reallocated to store a certain larger amount of data, etc. Although this solution allows for a fast access to individual data items and has a low memory-management overhead, it does not provide a *general* solution for the problems of a slow data manipulation and an inefficient memory usage for the following reasons:

- Depending on the used memory-management scheme, the memory may be fragmented in such a way that a reallocation results in a move of the array to be reallocated to a new memory location. With an array of a few hundred bytes this is not a problem, but moving an array of a size of several tens or hundreds of kilobytes can take a lot of time.
- The memory may even be fragmented in such a way that, although the total amount of free memory is sufficient to store the array to be reallocated, none of the free memory-blocks is large enough to store this array.

However, to store dynamically-sized character strings, representing, for example, state names and action names, this data structure is very suitable. Therefore, it is applied as such as the second basic data structure.

A more general solution is to use a *dynamically (re)allocated two-dimensional array.* This array consists of one so-called *column array* and several so-called *row arrays* as shown in figure 4.1. The column array contains pointers to the row arrays. The row arrays contain the representation of the actual data. Initially, one allocates a column array to contain pointers to a certain amount of row arrays. Further, one allocates one row array. When this row array is filled up, one allocates a second row array, etc. When as many row arrays have been allocated as the column array can contain pointers to, the column array is reallocated to contain a certain larger amount of pointers to row arrays, etc. This solution does not only have the same advantages as using a dynamically (re)allocated one-dimensional array, but also solves the problems of a slow data manipulation and an inefficient memory usage.

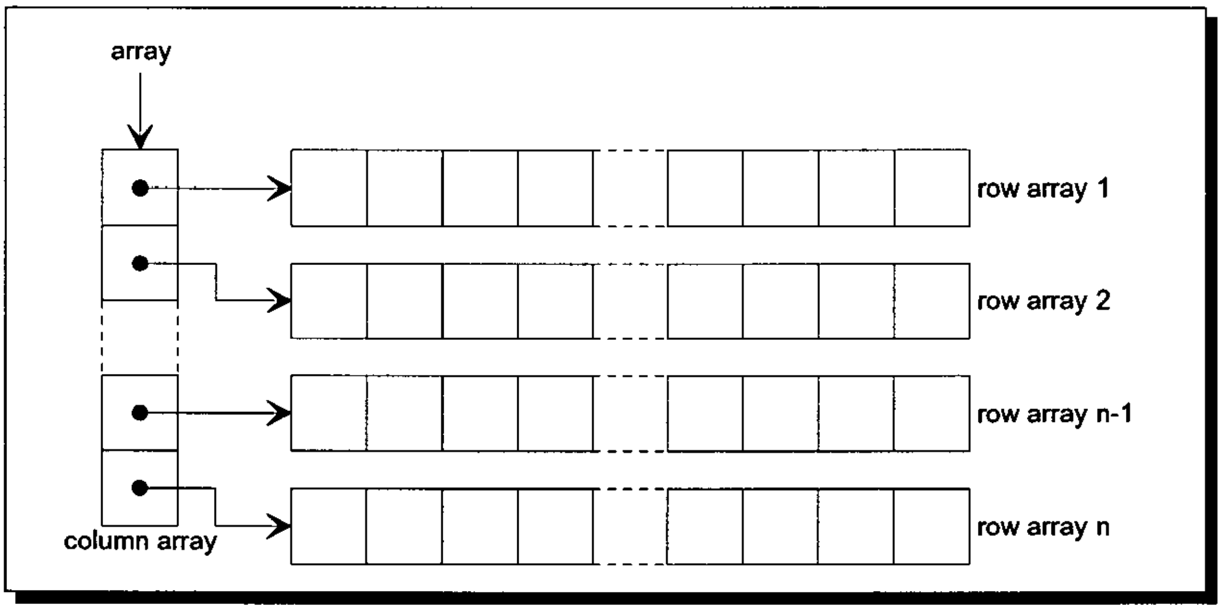

Figure 4.1: Basic data structure

The key issue of this third basic data structure is the size of the row array. This size influences the amount of data to be reallocated (the column array), the memory-management overhead (the number of blocks), and the memory waste due to a partially-filled last row array. Given a certain amount of data to be stored, the following applies: A large row array results in few column array reallocations and a low memory-management overhead, but also in a large waste of memory if the last row array is only partially filled. A small row array reduces this waste, but results in more column array reallocations and a larger memory-management overhead. Currently, the various data structures used in CCStool2 which are based on this data structure have been configured to accommodate for medium-sized CCS agents with row-array sizes ranging from 5 to 1000 data items depending on the amount of data items expected to be stored.

To access the i-th item, one calculates the column-array index and the row-array index of this element as follows (using zero-based indices):

$$\begin{aligned} \mathit{column\_array\_index} &= i \;/\; \mathit{row\_array\_size} \\ \mathit{row\_array\_index} &= i - x * \mathit{row\_array\_size} \end{aligned}$$

Next, one uses these indices to access the item itself, e.g. as follows:

$$\mathit{array}[\mathit{column\_array\_index}][\mathit{row\_array\_index}] = \mathit{value}$$

4.1.2 CCS Agent Representation and Storage

The following three types of operations are performed on CCS agents in CCStool2:

- manipulation of the structure of CCS agents, e.g. as in the abstraction and restriction functions (see Sections 2.6 and 2.7 respectively), and in the two closure algorithms used for observational equivalence verification discussed in Section 3.2
- manipulation of the names of states and actions, e.g. as in the state relabel and action relabel functions (see Section 2.3 and 2.4 respectively)
- manipulation of both the structure and the names, e.g. as in the extraction and expansion functions (see Sections 2.5 and 2.8 respectively)

For all three types of operations to be efficient in both time and space, the representation of CCS agents is split into two parts, each of which is stored in a separate data structure. The first data structure stores the representation of the structure of the agents with the states and actions represented as numbers. The second data structure stores the names of these states and actions represented as strings, and provides a mapping of the numbers used in the representation of the structure to the actual names. This latter data structure also contains the skip lists (see Subsection 4.1.3) which are used to improve the search for these names as mentioned previously. Further, this second data structure also stores the defined NIL states, start states, linked actions, invisible actions, and relabel actions (see Section 2.1), with the first represented as numbers and the latter four represented as strings.

Using the latter two basic data structure described in Subsection 4.1.1, the two data structures used to store the representation of CCS agents become as shown in figures 4.2 and 4.3 respectively.

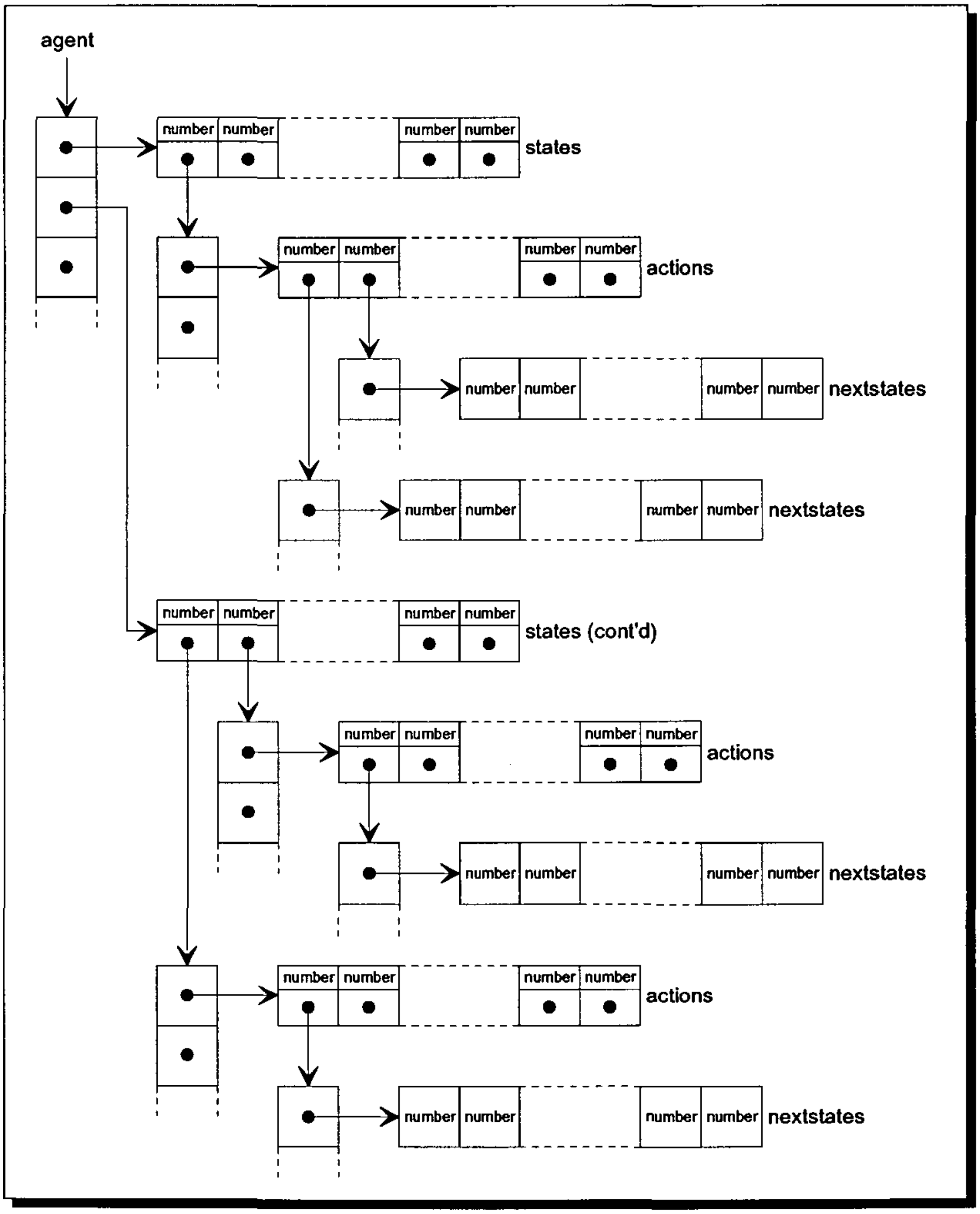

Figure 4.2: Data structure used to store the representation of the structure of CCS agents

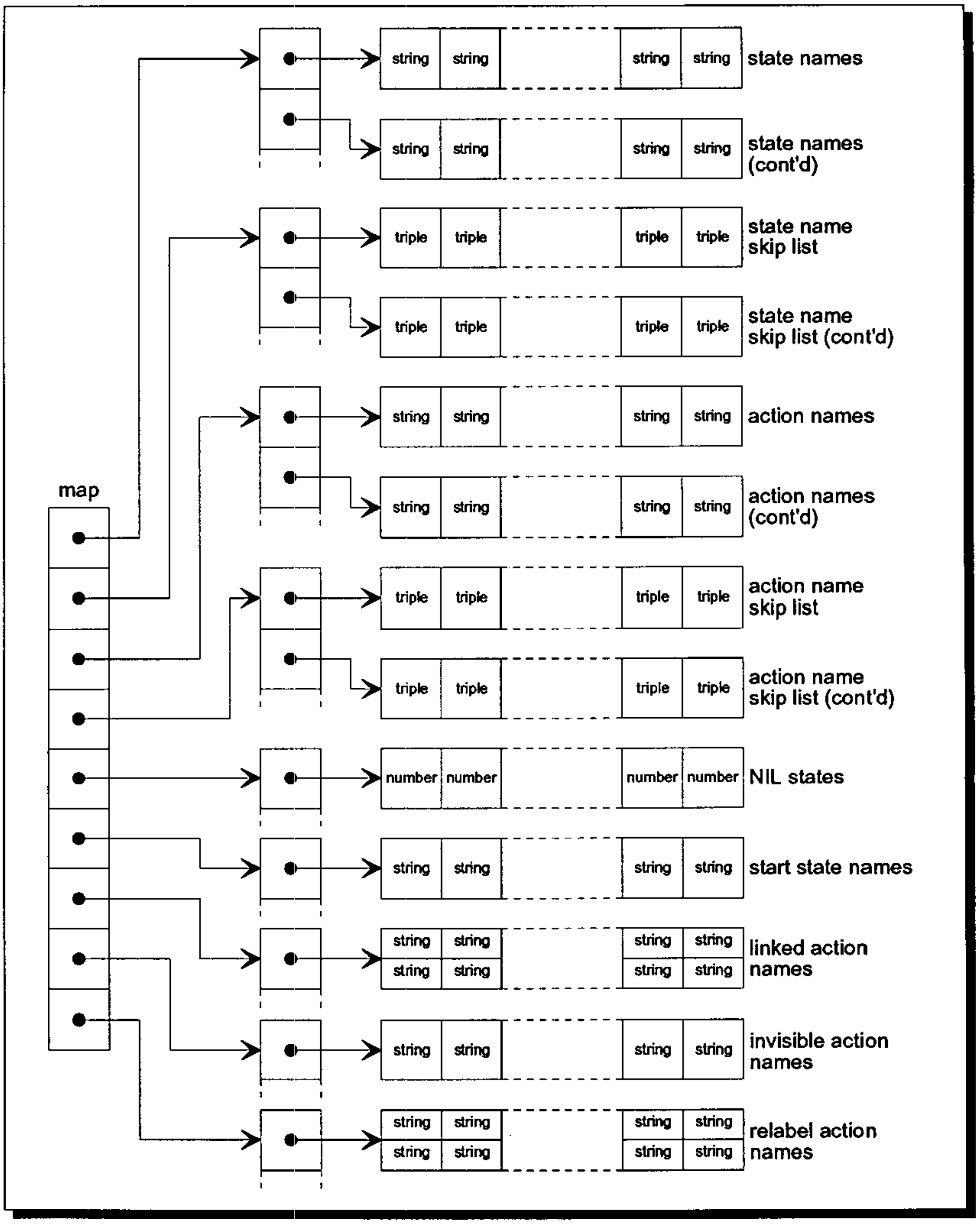

Figure 4.3: Data structure used to store the representation of the names of CCS agents

4.1.3 Skip List

In the data structure storing the representation of the names of the states and actions of CCS agents described in Subsection 4.1.2, these names are represented as strings which are stored in an array. Searching a string in an array using linear search takes a worst-case time of $\mathcal{O}(n)$ with n the number of strings. For the names of the start states, restricted actions, and linked actions, linear search is not a problem given the facts that the amount of names usually is small and that the names are accessed only a few times. However, linear search can become a serious problem for the names of states and actions, since CCStool2 has to be able to handle CCS agents with a large amount of states and actions. Especially compilation and expansion, during which the amount of states and actions with corresponding names increases, suffer from this.

The obvious approach to get worst-case search-costs of a order lower than $\mathcal{O}(n)$ is to use a binary-search scheme using a balanced tree or a self-adjusting tree. Binary search has a worst-case time of $\mathcal{O}(log\ n)$. However, balanced-tree and self-adjusting-tree algorithms are complex, hard to implement, and time consuming.

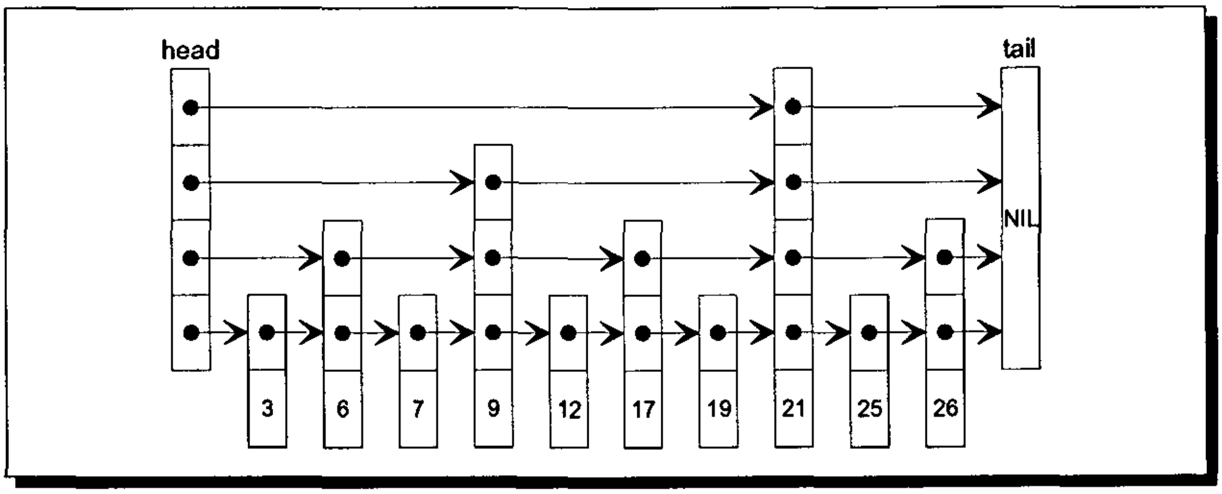

Figure 4.4: Example of a skip list

An only-recently introduced alternative to balanced trees and self-adjusting trees is the so-called *skip list* [Pug90a, Pug90b, LD91, MPS92, PMP92] of which an example is shown in figure 4.4. Skip-list algorithms are simpler, and therefore easier to implement, than, and provide significant constant-factor speed-improvements over, balanced-tree and self-adjusting-tree algorithms. The basic idea behind the skip list is to give certain nodes in a linked list not only a forward pointer to the successor node, but also to nodes "further away" to make it possible to *skip* one or more nodes when searching for a particular node. Searching for an element is done as follows: Starting at the top level of forward pointers of a dummy header-node, we traverse forward pointers that do not overshoot the element being searched for. When no more progress can be made at the current level of forward pointers, the search moves down one level. When no more progress can be made at level 1, we must be immediately in front of the node containing the element being searched for, if it is in the list.

The original skip list [Pug90a, Pug90b] is probabilistic in nature, i.e., given a set of values to be stored in a skip list and the insertion order of these values, the shape of the resulting skip list is not determined, but depends on the outcome of coin flips. Although the *average* search-cost of the *probabilistic* skip list is $O(log\ n)$, the *worst-case* search-cost is $O(n)$.

In [MPS92] several *deterministic* versions of the skip list are presented that have a *worst-case* search-cost of $O(log\ n)$. We have chosen for the so-called *top-down 1-2-3 skip list* and especially for the *linked-list representation* of this type of skip list for reasons of code simplicity. Further, all operations on this type of skip list are performed in a *top-down manner* which eliminates the need to maintain a stack for the search path. The main characteristic of this type of skip list is that only 1 or 2 nodes with $k-1$ forward pointers are allowed between two nodes with k forward pointers. An example is shown in figure 4.5.

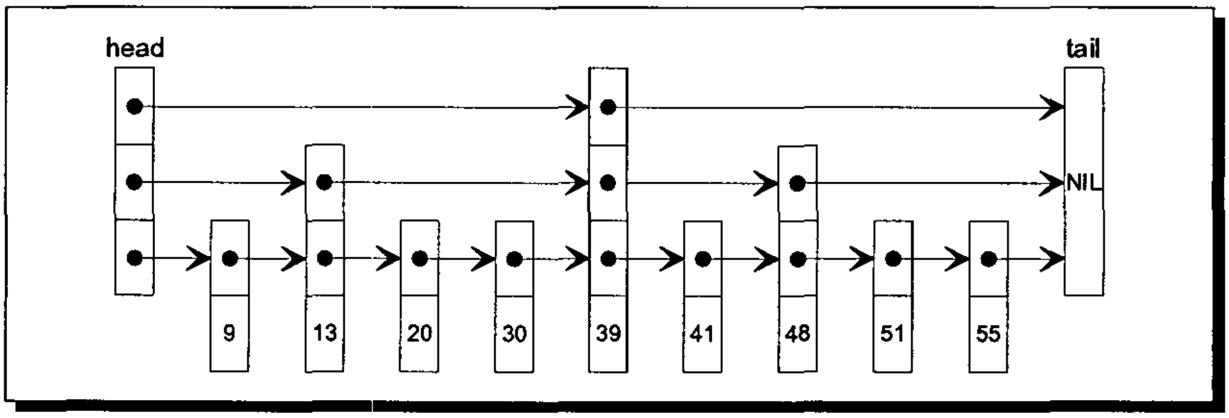

Figure 4.5: Example of a top-down 1-2-3 skip list

Each node in this skip list has two pointers: one forward pointer to the successor node at the same level and one pointer to the node at the next lower level. The skip list has a dummy node *head* and two sentinel nodes *bottom* and *tail* to avoid special cases in the algorithms. Further, to avoid a level of indirection, each node in which a comparison is made also contains the element needed for this comparison. Figure 4.6 shows another example of a top-down 1-2-3 skip list and its linked-list representation. Figure 4.7 shows the linked-list representation of an empty top-down 1-2-3 skip list.

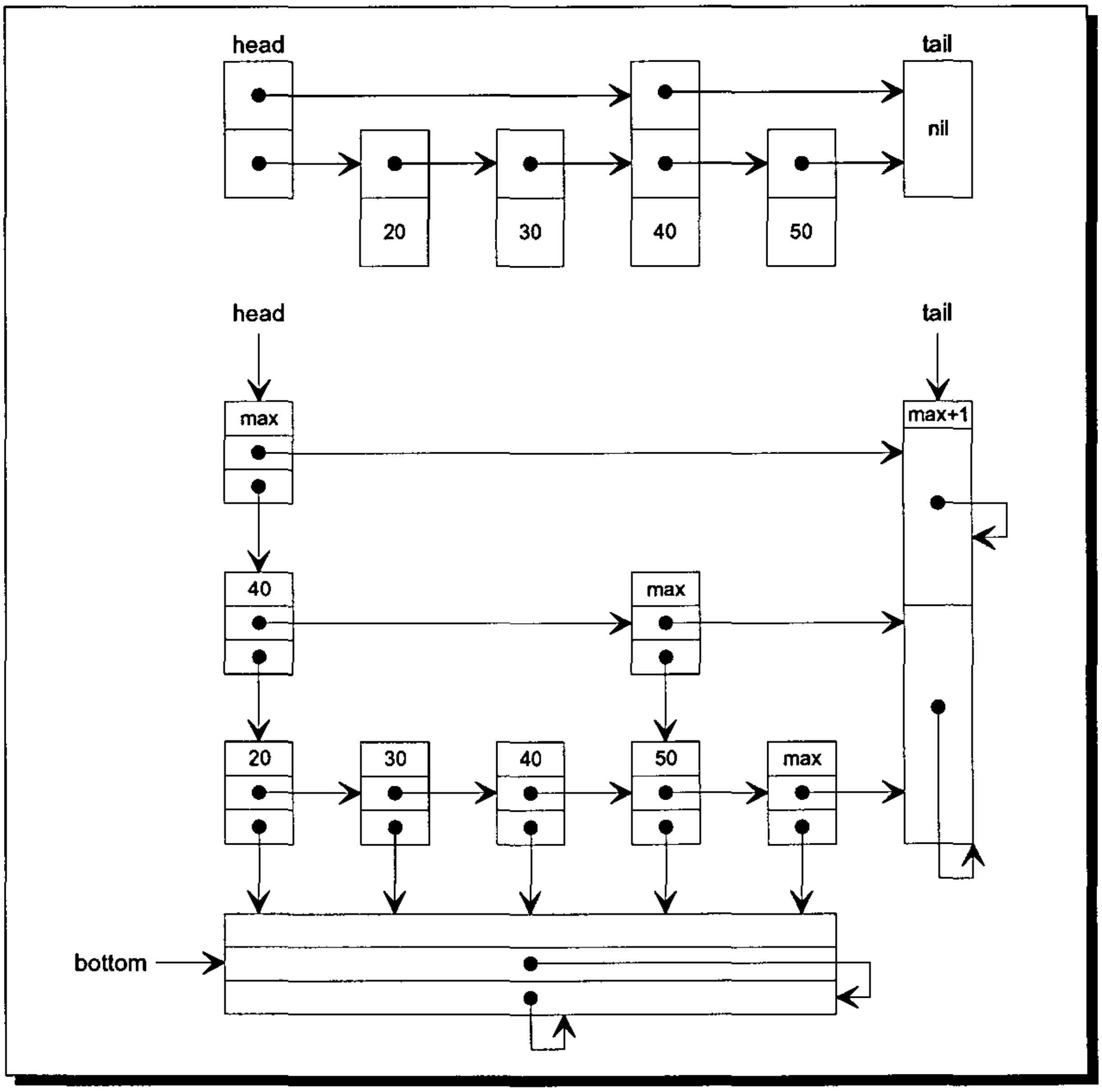

Figure 4.6: Example of a top-down 1-2-3 skip list and its linked-list representation

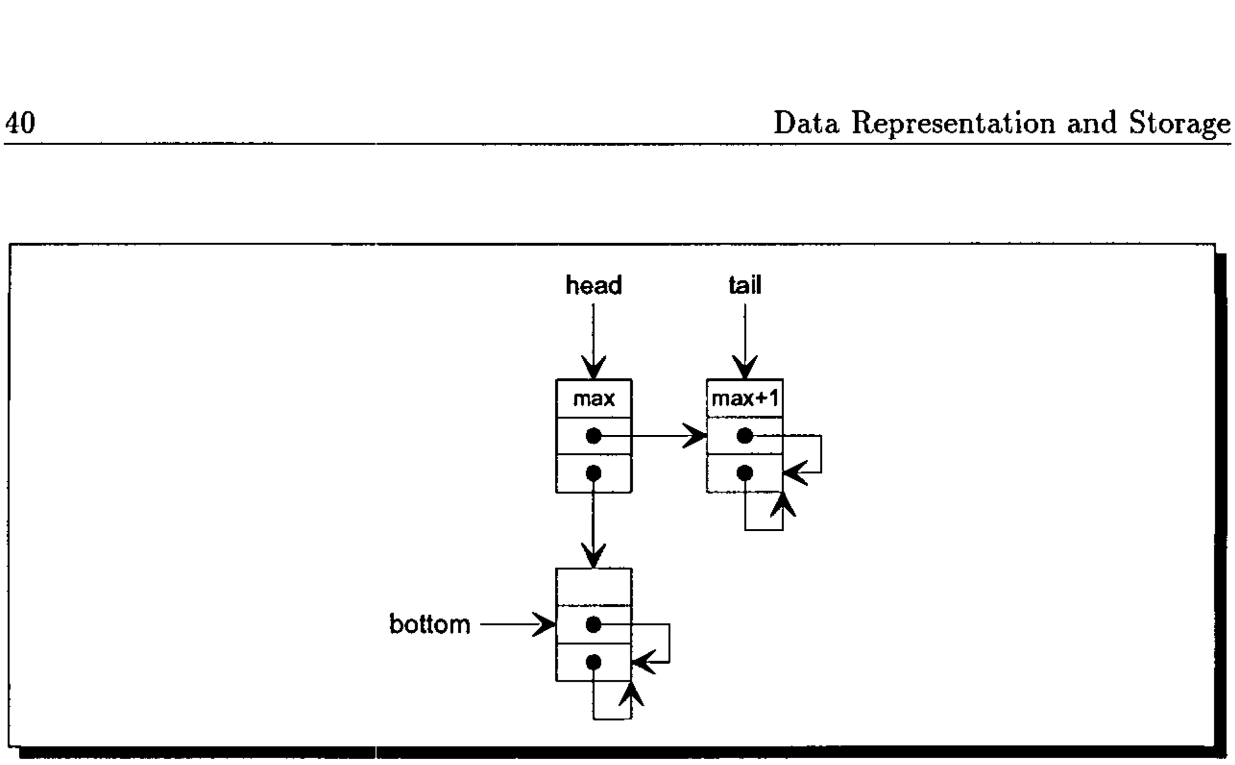

Figure 4.7: Linked-list representation of an empty top-down 1-2-3 skip list

Skip lists are implemented using dynamically (re)allocated two-dimensional arrays described in Subsection 4.1.1. Skip lists are used alongside string arrays as shown in figure 4.3 to keep these arrays accessible in a linear way. The elements of the skip list are the indices into these arrays.

4.1.4 State-Partition Representation and Storage

As mentioned in Section 3.1, the partition-refinement algorithm used in CCStool2 is based on the Kanellakis-Smolka algorithm [KS83]. The data structure used to store the representation of the state partition as used in our partition-refinement algorithm is based on the one described in [Bou92].

As also mentioned in Section 3.1, the partition-refinement algorithm maintains two block lists called *UnstableBlocks* and *StableBlocks*. Each block contains three lists: a list of the states in the block (state list), a list of the transitions of which the target states are states in the block (transition list) and a list of the actions through which the states in the block can be reached (action list). For efficiency reasons the block lists contain pointers to the blocks instead of the blocks themselves.

The states in the state list and the actions in the action list are represented as numbers which correspond with their respective numbers in the representation of the structure of a CCS agent described in Subsection 4.1.2. For each state a 2-tuple of the form $\langle block, marker\rangle$ is maintained for various computational purposes with *block* a pointer to the block that state belongs to and *marker* a boolean. Since the amount of states is known when building the initial partition (through maintenance of a counter in the representation of the structure of a CCS agent), a dynamically allocated one-dimensional array as described in Subsection 4.1.1 is used to store the representation of these 2-tuples. The state numbers in the state list are used as indices in this array.

The transitions are represented as 3-tuples of the form $\langle sourcestate, action, targetstate\rangle$ of which the elements are also represented as numbers which correspond to their respective numbers in the representation of the structure of a CCS agent. Since the amount of transitions also is known when building the initial partition (also through maintenance of a counter in the representation of the structure of a CCS agent), again a dynamically allocated one-dimensional array as described in Subsection 4.1.1 is used to store the representation of these 3-tuples. The transition list contains the indices in this array of the transitions in the list represented as numbers.

Using the basic data structures described in Subsection 4.1.1, the two data structures used to store the representation of a state-partition block and a state partition become as shown in figures 4.8 and 4.9 respectively.

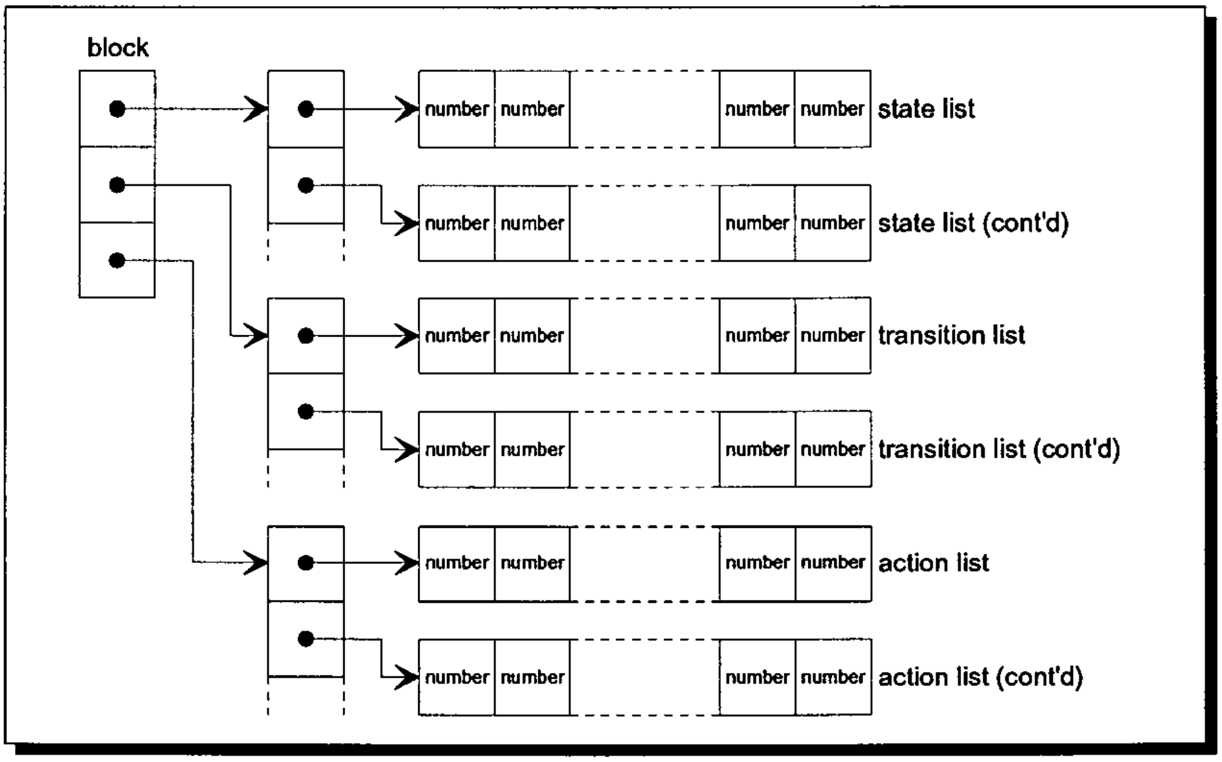

Figure 4.8: Data structure used to store the representation of a state-partition block

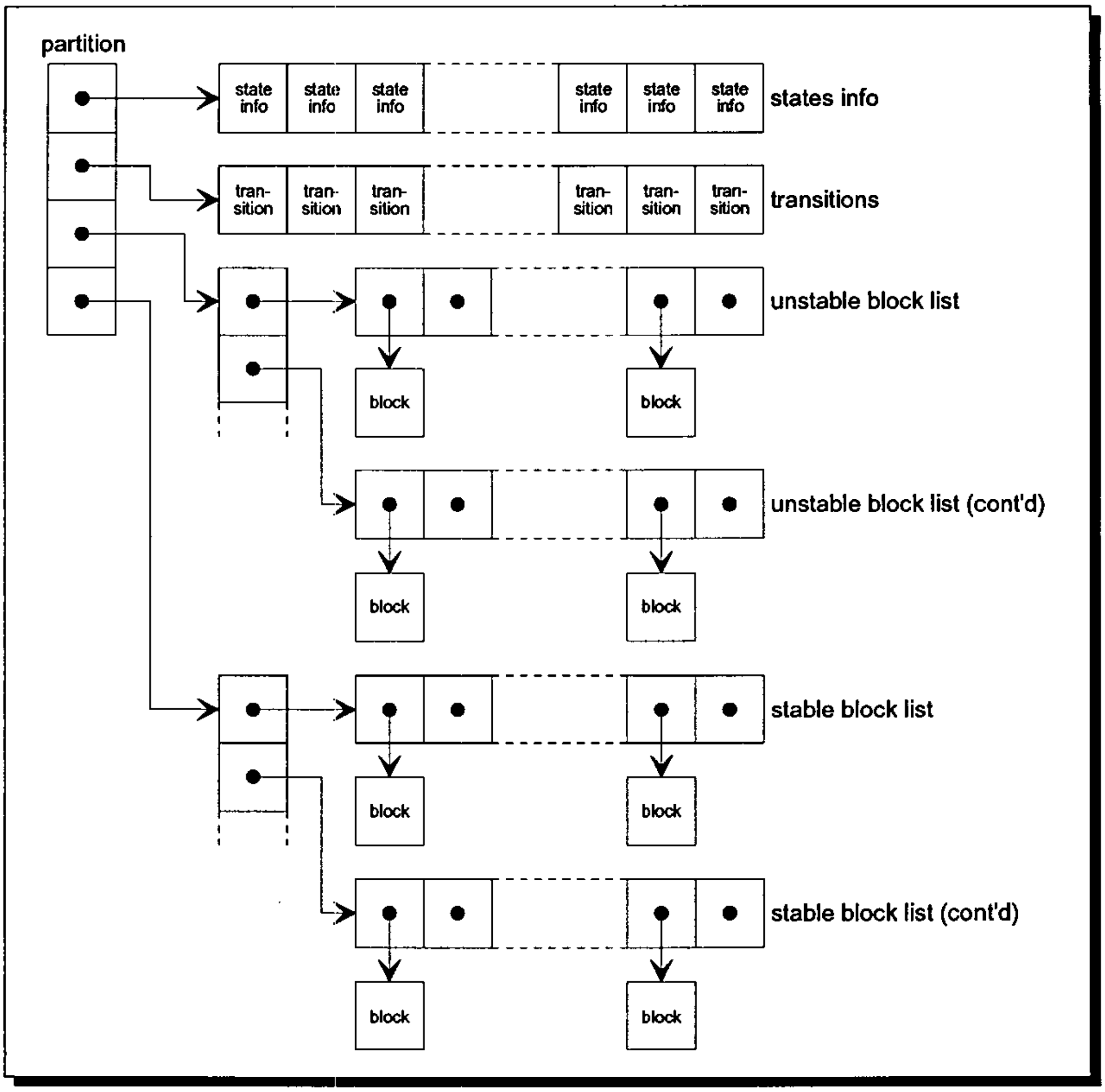

Figure 4.9: Data structure used to store the representation of a state partition

4.2 Implementation Language and Environment

One of the problems in implementing CCStool2 is the access to the available memory resources. The way these resources are made available to a program depends on the run-time environment of the program. To keep a program (as much as possible) independent of the way these resources are made available to it, and thereby "easily" portable, the run-time environment has to present these memory resources as one (large) memory pool, provide a uniform interface to it and take care of the mapping of this memory pool onto the available memory resources. A UNIX environment provides such a run-time environment, but an MS-DOS environment does not. In an MS-DOS environment programs are run in the so-called *real mode* of the processor in which the available memory resources are limited to 640kb. To access more memory, usually available as XMS and/or EMS, special code is needed which makes the program less portable to other environments. This latter approach is not needed, if a so-called *DOS extender* is used. Such an extender creates a run-time environment of the type described above by forcing the processor to run in the so-called *protected mode.*

We have chosen to implement CCStool2 in the programming language C for the following reasons:

- C is supported on a wide variety of computers and operating systems. This should make it easy to port a program written in C from one computing environment to another, especially if written in ANSI C.
- Implementations in C are in general efficient in both time and space.
- C makes it possible to implement pointer-based and array-based data structures very efficiently.

We have used the GNU C compiler which is available for both MS-DOS and UNIX environments. There are two MS-DOS ports of this compiler available, both of which include a DOS extender. For more information on this, see [VV94].

Chapter 5

Macro Functionality

CCStool2 has the following properties:

- The fsCCS description language is compositional. This means that the concatenation of two *syntactically* correct fsCCS descriptions is again a *syntactically* correct fsCCS description.
- The input files and output files are plain ASCII files.
- The provided functions can easily be combined to form more complex functions.

The combination of these three properties together with file-handling and text-handling commands and utilities provide a powerful mechanism to use macros or scripts to define generic higher-order functions and specific complex tasks.

In this chapter we give a number of examples demonstrating this mechanism. In these examples we use UNIX file-handling commands as well as CCStool2 commands. For the specific syntax of these latter commands, see [VV94].

Example 1

The first example is a macro to verify whether two agents are observational equivalent. Both agents are described in separate files. The macro is called `verify` and is invoked as

```
verify file1 file2
```

The body of the macro itself is shown in figure 5.1.

Example 2

A second example is a macro to rename a single action `action1` in an fsCCS description to `action2`. This macro is called `rename` and is used as

```
rename file action1 action2
```

The body of this macro is shown in figure 5.2.

```
#!/bin/sh

# concatenate both input files
cat $1 $2 > dummy.ccs

#call CCStool2 to verify observational equivalence
ccstool2 v -o dummy.ccs

# delete temporary file
rm dummy.ccs
```

Figure 5.1: Macro verify

```
#!/bin/sh

# generate a temporary file with a relabel operator
echo "relabel ($2, $3)" >> relabel.ccs

# concatenate this file and the fsCCS description
cat $1 relabel.ccs > dummy.ccs

# call CCStool2 to perform the action relabeling
ccstool2 l -a dummy.ccs $1

# delete temporary files
rm relabel.ccs
rm dummy.ccs
```

Figure 5.2: Macro rename

Example 3

A frequently occurring problem is the following: Given a specification *Spec*. Verify whether a potential implementation $(I_1 \mid I_2 \mid \ldots \mid I_n) \setminus L \approx Spec$.

Assume that agents $I_1, \ldots, I_n$ and *Spec* are described in separate files. Assume further that $\setminus L$ is described in a separate file using the appropriate *link* and *invisible* operators. We write a macro `direct_verify` invoked as

```
direct_verify I L n Spec
```

where `I` denotes the set of implementation components stored in the files `I1`, ..., `In`, `L` denotes the set of restrictions, `n` indicates the amount of components, and `Spec` is the specification.

The body of this macro is shown in figure 5.3.

```
#!/bin/sh

# concatenate restrict information and minimized components
cp $2 X
i=0
while [ $i != `expr $3` ]
do
   i=`expr $i + 1`
   ccstool2 m -o $1"$i" dummy.ccs
   cat dummy.ccs >> X
done

# calculate the expansion and minimize the result
ccstool2 q -o X X

# perform the actual verification
verify $4 X

# delete temporary file
rm dummy.ccs
rm X
```

Figure 5.3: Macro `direct_verify`

Note that the agents $I_1, \ldots, I_n$ are minimized before the actual expansion and verification is performed. In general, this can speed up the verification considerably (see Chapter 6). Note further that this macro uses the macro `verify` described in Example 1.

Example 4

A possibly better approach to solve the problem stated in Example 3 arises from the property that in general $(I_1 \mid I_2 \mid \ldots \mid I_n) \setminus L$ is observational equivalent to $(\ldots(((I_1 \mid I_2)\setminus L_2) \mid I_3)\setminus L_3)\ldots \mid I_n)\setminus L_n$ for some properly chosen $L_2, \ldots, L_n$. This property enables us to solve the problem *compositionally.*

First, we write a macro `combine` (see figure 5.4) invoked as

```
combine P Q L R
```

which calculates the expansion and minimization of two agents stored in the files P and Q with restrict information in L and returns the resulting agent in R.

```
#!/bin/sh

# concatenate the two components and their restrict information
cat $1 $2 $3 > dummy.ccs

# minimize the separate components
ccstool2 m -o dummy.ccs dummy.ccs

# calculate the expansion and the minimization
ccstool2 q -o dummy.ccs dummy.ccs

# relabel the states
ccstool2 l -s dummy.ccs $4

# delete temporary file
rm dummy.ccs
```

Figure 5.4: Macro `combine`

Next, we use this macro together with the macro `verify` (see Example 1) in `compos_verify` (see figure 5.5). This macro is invoked as

```
compos_verify I L n Spec
```

where I, n, and Spec have the same meaning as in `direct_verify` of Example 3, and L denotes the set of restrictions stored in the files L2, ..., Ln.

```
#!/bin/sh

# combine the components
combine $1'1' $1'2' $2'2' X
i=2
while [ $i != `expr $3` ]
do
   i=`expr $i + 1`
   combine X $1"$i" $2"$i" X
done

# perform the actual verification
verify X $4

# delete temporary file
rm X
```

Figure 5.5: Macro `compos_verify`

We conclude this chapter by remarking again that concatenating two valid fsCCS descriptions always results in a *syntactically* correct fsCCS description. However, the resulting description is not necessarily *semantically* correct due to the fact that concatenation can introduce *double-linked actions* and/or *double-to-be-relabeled actions.* Further, concatenation can introduce *clashes* of state names and action names.

Chapter 6

Performance

The most-used benchmark for determining the performance of verification tools is the minimization modulo observational equivalence of Milner's distributed scheduler [Fer89, GV90, EFT91, CPS93]. This scheduler consists of a starter process and N schedule processes. These processes are expressed in CCS [Mil89] as follows:

$$\begin{array}{ll}
\mathit{Starter} & \stackrel{def}{=} \overline{c_1} \cdot 0 \\
C_1 & \stackrel{def}{=} c_1 \cdot a_1 \cdot (b_1 \cdot \overline{c_2} \cdot C_1 + \overline{c_2} \cdot b_1 \cdot C_1) \\
C_2 & \stackrel{def}{=} c_2 \cdot a_2 \cdot (b_2 \cdot \overline{c_3} \cdot C_2 + \overline{c_3} \cdot b_2 \cdot C_2) \\
\vdots & \quad \vdots \\
C_N & \stackrel{def}{=} c_N \cdot a_N \cdot (b_N \cdot \overline{c_1} \cdot C_N + \overline{c_1} \cdot b_N \cdot C_N)
\end{array}$$

The specification of the scheduler is

$$\mathit{Sched}_N \stackrel{def}{=} (\mathit{Starter} \mid C_1 \mid C_2 \mid \cdots \mid C_N) \setminus \{ c_1, c_2, \cdots, c_N \}$$

N	states	transitions	AUTO	Aldébaran	CCStool2
4	97	241	0.5s	0.26s	0.17s
5	241	721	1.9s	0.88s	0.53s
6	577	2017	8.0s	2.6s	1.7s
7	1345	5377	38s	7.2s	5.9s
8	3073	13825	201s	21s	23s
9	6913	34561	-	56s	101s
10	15361	84481	-	160s	452s
11	33794	326661	-	*	2169s

Table 6.1: Benchmark results of Milner's distributed scheduler

figures available for these two tools of the execution of the complete "filter" process described above, we find it very hard to draw any conclusions with respect to the relative time performances of the execution of the complete process.

The benchmark results described above are obtained using a minimization process in which the starter process and the schedule processes are combined into a single process which subsequently is minimized. In practice, such an approach can be very inefficient in both time and space. As mentioned in Chapter 5, it is often more efficient to minimize systems in a compositional way. We applied a compositional-minimization macro, similar to the verification macro of Example 3 in Chapter 5, to the abstracted scheduler. The results of this benchmark are shown in table 6.3. Furthermore, we used a macro to generate the scheduler processes automatically.

N	states	transitions	CCStool2
4	97	241	0.18s
8	241	721	0.38s
16	2^{20}	2^{23}	0.78s
32	2^{36}	2^{39}	1.8s
64	2^{68}	2^{71}	5.2s
128	2^{132}	2^{135}	21s
256	2^{260}	2^{263}	104s
512	2^{516}	2^{519}	773s
1024	2^{1028}	2^{1031}	5954s

Table 6.3: Benchmark results of Milner's abstract distributed scheduler using compositional minimization

Here, the amounts of states and transitions refer to the expanded scheduler. For $N > 8$ these amounts are estimated. As far as we know no such figures are available for Aldébaran. For a scheduler with 128 schedule processes AUTO needs approximately 30 seconds to compute the minimal observational-equivalent [Bou92].

If we assume that the partitioning algorithm of CCStool2 is of order $\mathcal{O}\left(\mid Q \mid^3\right)$, we estimate that it would take about 10^{1000} *years!* to minimize a scheduler with 1024 components in a non-compositional way.

Chapter 7

Conclusions and Recommendations for Future Extensions

7.1 Conclusions

CCStool2 is an easy-to-use and automated tool which can perform various computations on fsCCS (finite state CCS) descriptions. The tool offers a rather complete set of basic functions including expansion, reduction, minimization, verification, state relabeling, action relabeling, abstraction, and restriction, which is easily extensible to meet possible future requirements.

CCStool2 behaves as a filter taking an input file containing an fsCCS description together with a command indicating the computation to be performed on the fsCCS description, and generating an output file containing the result of the computation in the form of another fsCCS description and/or optionally a report file containing information concerning the performed computation.

An fsCCS description consists of a number of fsCCS equations and a number of operators. The fsCCS equations specify the "pure" behaviour of the finite state CCS agents. The operators specify certain operations to be performed on certain states or actions of these agents.

CCStool2 is efficient in both time and space. Since in practice space efficiency is more important than time efficiency, we especially put effort in the former.

The minimization algorithms are based on the partition-refinement algorithm of Kanellakis and Smolka, and have a worst-case complexity of order $\mathcal{O}\left(\mid Q\mid^3\right)$. The tool also contains two reduction algorithms which have a linear-time complexity. In contrast to other verification tools, graph generation (expansion) is performed very efficiently, probably due to the chosen fsCCS description language.

Most data structures are based on dynamically (re)allocated two-dimensional arrays which support both a fast data manipulation and an efficient memory usage. Further, to improve the search in (very) large amounts of data, an only-recently introduced alternative to balanced trees and self-adjusting trees, called skip lists, is used.

CCStool2 has been implemented in the programming language C to keep it portable to a wide variety of computers and operating systems, to obtain an implementation which is efficient in both time and space, and to be able to implement pointer-based and array-based data structures very efficiently. We have used the GNU C compiler which is available for both MS-DOS and UNIX environments. To circumvent the well-known 640 kbyte memory limit of MS-DOS, a so-called DOS extender has been used.

The fsCCS description language is compositional, the input files and output files are in plain ASCII, and the provided functions can easily be combined to form more complex ones. Therefore, CCStool2 together with file-handling and text-handling commands and tools provide a powerful mechanism to use macros or scripts to define generic higher-order functions and specific complex tasks.

7.2 Recommendations for Future Extensions

The current version of CCStool2 provides only a basic set of functions. A number of practically applicable functions are not yet supported. For instance, it would be useful, especially in the context of digital system design, to support *multi-way synchronization* as in CSP [Hoa85]. To this we could, for example, add a *multi-link* operator and a *multi-way* expansion function.

Currently, the most famous equivalence relation is *observational equivalence*. However, a number of other useful equivalence relations have been defined which could be added, such as *failure equivalence*, *test equivalence*, and *branching bisimulation*. Also *preorder checking* and *model checking* are interesting possibilities to add to CCStool2. Through preorder checking it is verified whether a specification is a potential implementation of another specification. Using model checking certain *temporal properties* of a specification can be proven, such as *deadlock freedom* and *absence of individual starvation*.

Further, a more expressive (fs)CCS description language could be defined, which allows for an easier system specification. This language could be build in CCStool2 itself. An other approach is to develop a front-end which translates this languages into that of the current description language. However, the danger of a *too* powerful description language is that translation could become *enormously* time consuming.

Appendix

fsCCS Description Language of CCStool2

This appendix contains a context-free LL-1 grammar called $G_{CCStool2}$, which defines the syntax of the fsCCS description language implemented in CCStool2.

$G_{CCStool2}$ is defined as

$$G_{CCStool2} = (N_{CCStool2}, \Sigma_{CCStool2}, R_{CCStool2}, S_{CCStool2})$$

$N_{CCStool2}$, the set of nonterminals, is defined as

$$\begin{aligned} N_{CCStool2} = \{ & \mathit{fsCCStext}, \mathit{Equation}, \mathit{GoalState}, \mathit{RestEquation}, \mathit{Start}, \mathit{RestStart}, \\ & \mathit{Link}, \mathit{Invisible}, \mathit{RestInvisible}, \mathit{Relabel}, \mathit{Comment}, \mathit{CommentChars}, \\ & \mathit{CommentChar}, \mathit{State}, \mathit{Action}, \mathit{RestName}, \mathit{Letter}, \mathit{CapitalLetter}, \\ & \mathit{SmallLetter}, \mathit{Digit}\} \end{aligned}$$

The set of terminals $\Sigma_{CCStool2}$ is defined as

$$\Sigma_{CCStool2} = \{\text{"}\mathit{NIL}\text{"}, \text{"}\mathit{start}\text{"}, \text{"}\mathit{link}\text{"}, \text{"}\mathit{invisible}\text{"}, \text{"}\mathit{relabel}\text{"}\} \cup \mathit{ASCII}$$

The start symbol $S_{CCStool2}$ is defined as

$$S_{CCStool2} = \mathit{fsCCStext}$$

$R_{CCStool2}$ consists of the following production rules:

fsCCStext	→	ε
fsCCStext	→	*Equation fsCCStext*
fsCCStext	→	*Start fsCCStext*
fsCCStext	→	*Link fsCCStext*
fsCCStext	→	*Invisible fsCCStext*
fsCCStext	→	*Relabel fsCCStext*
fsCCStext	→	*Comment fsCCStext*
Equation	→	*State* " = " *GoalState*
GoalState	→	*"NIL"*
GoalState	→	*Action* " : " *State RestEquation*
RestEquation	→	ε
RestEquation	→	*Comment RestEquation*
RestEquation	→	" + " *Action* " : " *State RestEquation*
Start	→	*"start"* "(" *State RestStart* ")"
RestStart	→	ε
RestStart	→	"," *State RestStart*
Link	→	*"link"* "(" *Action* "," *Action* ")"
Invisible	→	*"invisible"* "(" *Action RestInvisible* ")"
RestInvisible	→	ε
RestInvisible	→	"," *Action RestInvisible*
Relabel	→	*"relabel"* "(" *Action* "," *Action* ")"
Comment	→	"{" *CommentChars* "}"
CommentChars	→	ε
CommentChars	→	*CommentChar CommentChars*
CommentChar	→	*ASCII* \ { "{", "}" }
State	→	*CapitalLetter RestName*
Action	→	*SmallLetter RestName*

RestName → ε
RestName → *Letter RestName*
RestName → *Digit RestName*
RestName → "|" *RestName*
RestName → "?" *RestName*
RestName → "!" *RestName*
RestName → "_" *RestName*
RestName → "[" *RestName*
RestName → "]" *RestName*
RestName → "<" *RestName*
RestName → ">" *RestName*
RestName → ";" *RestName*
RestName → "^" *RestName*
RestName → "#" *RestName*
RestName → "$" *RestName*
RestName → "%" *RestName*
RestName → "&" *RestName*
RestName → "*" *RestName*
RestName → "−" *RestName*
RestName → "~" *RestName*
RestName → "‘" *RestName*
RestName → "\" *RestName*
RestName → "/" *RestName*
RestName → "’" *RestName*
RestName → """ *RestName*
RestName → "." *RestName*

Letter → *CapitalLetter*
Letter → *SmallLetter*

CapitalLetter → "*A*"
CapitalLetter → ···
CapitalLetter → "*Z*"

SmallLetter → "*a*"
SmallLetter → ···
SmallLetter → "*z*"

Digit → "0"
Digit → ···
Digit → "9"

References

[AHU87] Aho, A.V. and J.E. Hopcroft, J.D. Ullman
Data Structures and Algorithms.
Amsterdam : Addison-Wesley, 1983.

[Bou92] Bouali, A.
Weak and Branching Bisimulation in Fctool.
Le Chesnay : Institut National de Recherche en Informatique et en Automatique (INRIA), 1992.
INRIA Rapport de Recherche no. 1575.

[BRSV89] Boudol, G. and V. Roy, R. de Simone, D. Vergamini
Process Calculi, from Theory to Practice: Verification Tools.
In: Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, June 12–14, 1989. Ed. by J. Sifakis. Berlin : Springer, 1990. Lecture Notes in Computer Science, Vol. 407, p. 1–10.

[BS87] Bolognesi, T. and S.A. Smolka
Fundamental Results for the Verification of Observational Equivalence: a Survey.
In: Proceedings of the IFIP WG6.1 Seventh International Symposium on Protocol Specification, Testing, and Verification, Zuerich, Switzerland, May 5–8, 1987. Ed. by H. Rudin and C.H. West. Amsterdam : North-Holland, 1987. P. 165–179.

[CPS89] Cleaveland, R. and J. Parrow, B. Steffen
The Concurrency Workbench.
In: Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble, France, June 12–14, 1989. Ed. by J. Sifakis. Berlin : Springer, 1990. Lecture Notes in Computer Science, Vol. 407, p. 24–37.

[CPS93] Cleaveland, R. and J. Parrow, B. Steffen
The Concurreny Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems.
ACM Transactions on Programming Languages and Systems, Vol. 15(1993), no. 1, p. 36–72.

[EFT91] Enders, R. and T. Filkorn, D. Taubner
Generating BDDs for Symbolic Model Checking in CCS.
In: Proceedings of the 3rd International Workshop on Computer Aided Verification (CAV'91), Aalborg, Denmark, July 1–4, 1991. Ed. by K.G. Larsen and A. Skou. Berlin : Springer, 1992. Lecture Notes on Computer Science, Vol. 575, p. 203–213.

[Fer89] Fernandez, J.-C.
An Implementation of an Efficient Algorithm for Bisimulation Equivalnce.
Science of Computer Programming, Vol. 13(1989/1990), p. 219–236.

[GS88] Giacalone, A. and S.A. Smolka
Integrated Environments for Formally Well-Founded Design and Simulation of Concurrent Systems.
IEEE Transactions on Software Engineering, Vol. 14(1988), no. 6, p. 787–802.

[GV90] Groote, J.F. and F.W. Vaandrager
An efficient algorithm for branching bisimulation and stuffering equivalence.
Amsterdam : Centre for Mathematics and Computer Science (CWI), 1990.
CWI Report CS-R 9001.

[Hoa85] Hoare, C.A.R.
Communicating Sequential Processes.
Englewood Cliffs : Prentice Hall, 1985.

[Koo91] Koomen, C.J.
The Design of Communicating Systems: A System Engineering Approach.
Dordrecht : Kluwer, 1991.

[Kor91] Korver, H.P.
The Current State of Bisimulation Tools.
Amsterdam : Centre for Mathematics and Computer Science (CWI), 1991.
CWI Report CS-R 9108.

[KS83] Kanellakis, P.C. and S.C. Smolka
CCS Expressions, Finite State Processes, and Three Problems of Equivalence.
In: Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing, Montreal, Canada, August 1983. Ed. by N.A. Lynch. New York : A.C.M., 1983. P. 228–240.

[LD91] Lewis, H.R. and L. Denenberg
Data Structures and their Algorithms.
New York : Harper Collins, 1991.

[Mil89] Milner, R.
Communication and Concurrency.
London : Prentice Hall, 1989.

[MPS92] Munro, J.I. and T. Papadakis, R. Sedgewick
Deterministic Skip Lists.
In: Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms, Orlando, Florida, January 27–29, 1992. New York : A.C.M., 1992. P. 367–375.

[MSGS88] Malhotra, J. and S.A. Smolka, A. Giacalone, R. Shapiro
Winston: A Tool for Hierarchical Design and Simluation of Concurrent Systems.
In: Proceedings of the BCS-FACS Workshop on Specification and Verification of Concurrent Systems, University of Stirling, Scotland, July 6–8, 1988. Ed. by C. Rattray. Berlin : Springer, 1990. P. 140–152.

[PMP92] Papadakis, T. and J.I. Munro, P.V. Poblete
Average Search and Update Costs in Skip Lists.
BIT, Vol. 32(1992), no. 6, p. 316–322.

[Por87] Porro, S.F.
The CCS_TOOL: An information system for the support of CCS and IDP.
Faculty of Mathematics and Computer Science, Eindhoven University of Technology, July 1987.
Master's thesis. Faculty of Math. and Comp. Sci. no. 60.

[PT87] Paige, R. and R.E. Tarjan
Three Partitioning Refinement Algorithms.
SIAM Journal of Computing, Vol. 16(1987), no. 6, p. 973–989.

[Pug90a] Pugh, W.
A Skip List Cookbook.
Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, June 1990.
Technical Report UMIACS-TR-89-72.1/CS-TR-2286.1.

[Pug90b] Pugh, W.
Skip Lists: A Probabilistic Alternative to Balanced Trees.
Communications of the ACM, Vol. 33(1990), no. 6, p. 668–676.

[RS90] Roy. V. and R. de Simone
Auto/Autograph.
In: Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV'90), New Brunswick, New Jersey, June 18–21, 1990. Ed. by E.M. Clarke and R.P. Kurshan. Berlin : Springer, 1990. Lecture Notes on Computer Science, Vol. 531, p. 65–75.

[VV94] Van Rangelrooij, A. and J.P.M. Voeten
CCStool2: An expansion, minimization, and verification tool for finite state CCS descriptions, User's Manual, version 2.4..
Section of Digital Information Systems, Faculty of Electrical Engineering, Eindhoven University of Technology, October 1994.
Internal Sectional Report MAN-EB-9402.

[Tar72] Tarjan, R.
Depth-First Search and Linear Graph Algorithms.
SIAM Journal of Computing, Vol. 1(1972), no. 2, p. 146–160.

(258) Vleeshouwers, J.M.
DERIVATION OF A MODEL OF THE EXCITER OF A BRUSHLESS SYNCHRONOUS MACHINE.
EUT Report 92-E-258. 1992. ISBN 90-6144-258-3

(259) Orlov, V.B.
DEFECT MOTION AS THE ORIGIN OF THE 1/F CONDUCTANCE NOISE IN SOLIDS.
EUT Report 92-E-259. 1992. ISBN 90-6144-259-1

(260) Rooijackers, J.E.
ALGORITHMS FOR SPEECH CODING SYSTEMS BASED ON LINEAR PREDICTION.
EUT Report 92-E-260. 1992. ISBN 90-6144-260-5

(261) Boom, T.J.J. van den and A.A.H. Damen, Martin Klompstra
IDENTIFICATION FOR ROBUST CONTROL USING AN H-infinity NORM.
EUT Report 92-E-261. 1992. ISBN 90-6144-261-3

(262) Groten, M. and W. van Etten
LASER LINEWIDTH MEASUREMENT IN THE PRESENCE OF RIN AND USING THE RECIRCULATING SELF HETERODYNE METHOD.
EUT Report 92-E-262. 1992. ISBN 90-6144-262-1

(263) Smolders, A.B.
RIGOROUS ANALYSIS OF THICK MICROSTRIP ANTENNAS AND WIRE ANTENNAS EMBEDDED IN A SUBSTRATE.
EUT Report 92-E-263. 1992. ISBN 90-6144-263-X

(264) Freriks, L.W. and P.J.M. Cluitmans, M.J. van Gils
THE ADAPTIVE RESONANCE THEORY NETWORK: (Clustering-) behaviour in relation with brainstem auditory evoked potential patterns.
EUT Report 92-E-264. 1992. ISBN 90-6144-264-8

(265) Wellen, J.S. and F. Karouta, M.F.C. Schemmann, E. Smalbrugge, L.M.F. Kaufmann
MANUFACTURING AND CHARACTERIZATION OF GAAS/ALGAAS MULTIPLE QUANTUMWELL RIDGE WAVEGUIDE LASERS.
EUT Report 92-E-265. 1992. ISBN 90-6144-265-6

(266) Cluitmans, L.J.M.
USING GENETIC ALGORITHMS FOR SCHEDULING DATA FLOW GRAPHS.
EUT Report 92-E-266. 1992. ISBN 90-6144-266-4

(267) Józwiak, L. and A.P.H. van Dijk
A METHOD FOR GENERAL SIMULTANEOUS FULL DECOMPOSITION OF SEQUENTIAL MACHINES: Algorithms and implementation.
EUT Report 92-E-267. 1992. ISBN 90-6144-267-2

(268) Boom, H. van den and W. van Etten, W.H.C. de Krom, P. van Bennekom, F. Huijskens, L. Niessen, F. de Leijer
AN OPTICAL ASK AND FSK PHASE DIVERSITY TRANSMISSION SYSTEM.
EUT Report 92-E-268. 1992. ISBN 90-6144-268-0

(269) Putten, P.H.A. van der
MULTIDISCIPLINAIR SPECIFICEREN EN ONTWERPEN VAN MICROELEKTRONICA IN PRODUKTEN (in Dutch).
EUT Report 93-E-269. 1993. ISBN 90-6144-269-9

(270) Bloks, R.H.J.
PROGRIL: A language for the definition of protocol grammars.
EUT Report 93-E-270. 1993. ISBN 90-6144-270-2

N	states	transitions	AUTO	Aldébaran	CCStool2
4	97	1735	0.4s	0.15s	0.16s
5	241	6487	1.1s	0.6s	0.56s
6	577	23335	3.3s	1.9s	2.1s
7	1345	81655	12s	6.9s	7.3s
8	3073	279943	57s	24s	26s
9	6913	944791	-	80s	95s
10	15361	3149287	-	-	338s

Table 6.2: Benchmark results of Milner's abstract distributed scheduler

The benchmark results of schedulers of different sizes are presented in tables 6.1 and 6.2. The latter refers to the scheduler where the actions b_i are hidden (abstracted). The first table shows the amount of states and transitions of the expanded scheduler. The second table shows the amount of states and transitions after the reflexive-transitive closure has been calculated. Both tables show the execution times of AUTO, Aldébaran, and CCStool2. As far as we know the former two are the fastest verification tools currently available for deciding observational equivalence using an approach similar to that of CCStool2. The figures for AUTO and Aldébaran are obtained using a SUN 3/60 with 16 Mbyte RAM and 50 Mbyte RAM respectively and are taken from [GV90]. Those for CCStool2 are obtained using a Silicon Graphics Power Challenge XL with 12 MIPS R4400 processors each running at 150 MHz, 1.5 Gbyte RAM, and 12 Mbyte cache (1 Mbyte per processor). In both tables a "-" indicates that no outcome could be obtained due to a lack of memory, and a "*" means that no outcome is available.

Of course, an absolute comparison of the various figures only yields a very gross estimate of the relative efficiencies of the various tools, since the results were obtained using machines with a large speed difference. Further, the figures of CCStool2 refer to the execution of the complete "filter" process (see Section 2.1) which includes compilation, graph generation (expansion), calculation of the reflexive-transitive closure, calculation of the observational-equivalence classes, and output generation, whereas those of the other tools do not. The figures of AUTO only refer to the calculation of the reflexive-transitive closure and that of the observational-equivalence classes, whereas those of Aldébaran only refer to the calculation of the equivalence classes. In general, graph generation (expansion) and graph transformation (for instance, closure calculation) are the most costly. In comparison with these operations, graph partitioning can be neglected [Kor91]. However, CCStool2 shows the opposite, probably due to the chosen fsCCS description language which enables efficient expansion (graph generation). This means that the figures of CCStool2 give an indication of the performance of the partitioning algorithm (in practice, partitioning takes approximately half of the total time). So, AUTO and Aldébaran are probably more time efficient with respect to *minimization* of state graphs than CCStool2. Since there are no

ISSN 0167-9708
Coden: TEUEDE

(271) Bloks, R.H.J.
CODE GENERATION FOR THE ATTRIBUTE EVALUATOR OF THE PROTOCOL ENGINE GRAMMAR PROCESSOR UNIT.
EUT Report 93-E-271. 1993. ISBN 90-6144-271-0

(272) Yan, Keping and E.M. van Veldhuizen
FLUE GAS CLEANING BY PULSE CORONA STREAMER.
EUT Report 93-E-272. 1993. ISBN 90-6144-272-9

(273) Smolders, A.B.
FINITE STACKED MICROSTRIP ARRAYS WITH THICK SUBSTRATES.
EUT Report 93-E-273. 1993. ISBN 90-6144-273-7

(274) Bollen, M.H.J. and M.A. van Houten
ON INSULAR POWER SYSTEMS: Drawing up an inventory of phenomena and research possibilities.
EUT Report 93-E-274. 1993. ISBN 90-6144-274-5

(275) Deursen, A.P.J. van
ELECTROMAGNETIC COMPATIBILITY: Part 5, installation and mitigation guidelines, section 3, cabling and wiring.
EUT Report 93-E-275. 1993. ISBN 90-6144-275-3

(276) Bollen, M.H.J.
LITERATURE SEARCH FOR RELIABILITY DATA OF COMPONENTS IN ELECTRIC DISTRIBUTION NETWORKS.
EUT Report 93-E-276. 1993. ISBN 90-6144-276-1

(277) Weiland, Siep
A BEHAVIORAL APPROACH TO BALANCED REPRESENTATIONS OF DYNAMICAL SYSTEMS.
EUT Report 93-E-277. 1993. ISBN 90-6144-277-X

(278) Gorshkov, Yu.A. and V.I. Vladimirov
LINE REVERSAL GAS FLOW TEMPERATURE MEASUREMENTS: Evaluations of the optical arrangements for the instrument.
EUT Report 93-E-278. 1993. ISBN 90-6144-278-8

(279) Creyghton, Y.L.M. and W.R. Rutgers, E.M. van Veldhuizen
IN-SITU INVESTIGATION OF PULSED CORONA DISCHARGE.
EUT Report 93-E-279. 1993. ISBN 90-6144-279-6

(280) Li, H.Q. and R.P.P. Smeets
GAP-LENGTH DEPENDENT PHENOMENA OF HIGH-FREQUENCY VACUUM ARCS.
EUT Report 93-E-280. 1993. ISBN 90-6144-280-X

(281) Di, Chennian and Jochen A.G. Jess
ON THE DEVELOPMENT OF A FAST AND ACCURATE BRIDGING FAULT SIMULATOR.
EUT Report 94-E-281. 1994. ISBN 90-6144-281-8

(282) Falkus, H.M. and A.A.H. Damen
MULTIVARIABLE H-INFINITY CONTROL DESIGN TOOLBOX: User manual.
EUT Report 94-E-282. 1994. ISBN 90-6144-282-6

(283) Meng, X.Z. and J.G.J. Sloot
THERMAL BUCKLING BEHAVIOUR OF FUSE WIRES.
EUT Report 94-E-283. 1994. ISBN 90-6144-283-4

(284) Rangelrooij, A. van and J.P.M. Voeten
CCSTOOL2: An expansion, minimization, and verification tool for finite state CCS descriptions.
EUT Report 94-E-284. 1994. ISBN 90-6144-284-2