

SICS/T-89/8910

**Implementation of a Verification Method to
Communication Protocols
by
Ahmed Hussain Khan**



"Implementation of a Verification Method to Communication Protocols"

M.Sc.C.S.E Thesis

by

Ahmed Hussain Khan

E-mail: ahmed@sics.se,ahmed@draken.nada.se

The Royal Institute Of Technology(KTH)

Stockholm

December 1988.



Supervised by

Bengt Jonsson and Joachim Parrow

Swedish Institute Of Computer Science(SICS)

Box 1263

S - 164 28 Kista

Sweden.



IMPLEMENTATION OF A VERIFICATION METHOD TO COMMUNICATION PROTOCOLS

by

Ahmed Hussain Khan

Swedish Institute of Computer Science
Box 1263, S - 164 28 Kista, Sweden
E-mail: ahmed@sics.se

ABSTRACT

It is important to reason about a number of desirable protocol properties to ensure correctness of a designed communicating system. Such properties can be formulated in some temporal logic. If the specification of a communicating system is finite-state, that is, the system has a finite number of distinct states, it is often possible to verify automatically by an efficient algorithm, called a model checking algorithm, whether the system satisfies a property expressed in the logic. We describe a model checking method for Communicating Systems. An implementation of this model-checking facility is obtained by combining two automated tools namely, CWB and EMC. In the CWB[5], Communicating Systems, expressed in CCS-expressions can be represented in the form of labeled transition graphs. In the EMC, there is an efficient model checking algorithm for branching time temporal logic(CTL) with the standard semantics. We change the standard semantics for CTL in terms of communication actions. We implement a transformation of the transition graph into a finite number of distinct states so that it can be fed into EMC to use our logic.

Key Words:Communicating Systems, Model Checking, Concurrency Workbench(CWB), Transition Graph(TG), Alternating Bit Protocol(ABP) and Temporal Logic.

1. INTRODUCTION

When several computer communicate in a communicating system, communication protocols must be specified precisely and correctly. In a complex communication

protocol, the design can have errors. To detect such errors, it is important to reason about a number of desirable properties. Such properties can often be formulated in some temporal logic which expresses properties such as, undefined receptions, freedom from deadlock etc.

If the specification of a communicating system is finite-state, that is, the system has a finite number of distinct states, it is often possible to verify automatically by an efficient algorithm, called a model checking algorithm, whether the system satisfies a property expressed in the logic. An example can be found in [2] and [3], where a model checking algorithm for a propositional branching time temporal logic called CTL(Computation Tree Logic) is developed. The logic that is used in this method expresses properties of sequences of states rather than communication actions.

The Concurrency Workbench(CWB) is a program system that analyzes communication systems expressed in Milner's Calculus of Communicating Systems[4] and it consists of different analysis methods. There already exists a model-checking method for the propositional mu-calculus which express properties of states rather than actions.

The main purpose of this work is to extend CWB with an efficient model checking facility for a temporal logic that can express properties of communication actions and implement it effectively.

Many interesting properties of a communicating system can be specified in propositional temporal logic specification. The example that we used in this report is the Alternating Bit Protocol. A typical requirement for such a system is that every transmitted message must ultimately be received. This can easily be expressed in temporal logic.

Temporal logics exists in two forms [7] namely branching-time temporal logics and linear temporal logics. We have chosen branching-time temporal logics for the following reasons:

- i. it has an efficient model-checking algorithm
- ii. an efficient implementation of a model checker for branching-time temporal logic exists[2], developed at Carnegie Mellon University.

The standard semantics for CTL is defined in terms of sequences of states. We define an alternative semantics for CTL in terms of sequences of communication actions. We have implemented a model-checking method for CTL formulas in our modified semantics that uses a model-checking method for the standard semantics. Our effort in this thesis has been a transformation on communicating systems, which has the property that a given expression in CTL is true of a communicating system in our semantics if and only if the CTL expression is true of the transformed system in the standard semantics. We then use the Extended Model Checker to verify whether the transformed system satisfies the given CTL formula.

Our main implementation in this work is the creation of a model-checking facility for our modified semantics for CTL. It has been obtained by defining a transformation from labeled transition graphs into Kripke Structures. An implementation of this model-checking facility is obtained by combining two automated tools namely CWB and EMC.

The EMC system is written in a combination of lisp and C. The CWB system and our implementation is written in ML.

The outline of our thesis is as follows: Section 2 contains definition of transition graph. In section 3 we describe the syntax and semantics of the temporal logic called Computation Tree Logic(CTL). The definition of Kripke Structures is introduced in section 4. In section 5, we define our transformation of transition graph into a Kripke Structure. Section 6 deals with a description of our implementation and finally section 7 describes the application of our approach in protocol verification.

2. DEFINITION OF TRANSITION GRAPHS

In this section, we introduce labeled transition graphs .

A (finite) labeled transition graph or ltg is a 4-tuple, $TG = (M, s_0, \Sigma, \Delta)$ where

- ◆ M is a finite set of states.
- ◆ $s_0 \in M$ is the initial state.
- ◆ Σ is a finite set of actions, which includes the distinguished silent action ϵ and
- ◆ $\Delta \subseteq M \times \Sigma \times M$ is a transition system.

We use s, s' etc to range over states, α to range over actions and a transition is written as $s \xrightarrow{\alpha} s'$.

Intuitively, the possible future behavior of a transition graph is represented by its states. The states can change in actions. We can also represent a transition graph graphically.

Figure 2.1 represents a transition graph which first performs either a or c and thereafter b or d.

The above figure represents the $TG = (M, s_0, \Sigma, \Delta)$ where

$$M = \{s_0, s_1, s_2, s_f\}.$$

$$\Sigma = \{a, c, b, d\} \text{ and}$$

$$\Delta = \{s_0 \xrightarrow{a} s_1, s_0 \xrightarrow{c} s_2, s_1 \xrightarrow{b} s_f, s_2 \xrightarrow{d} s_f\}.$$

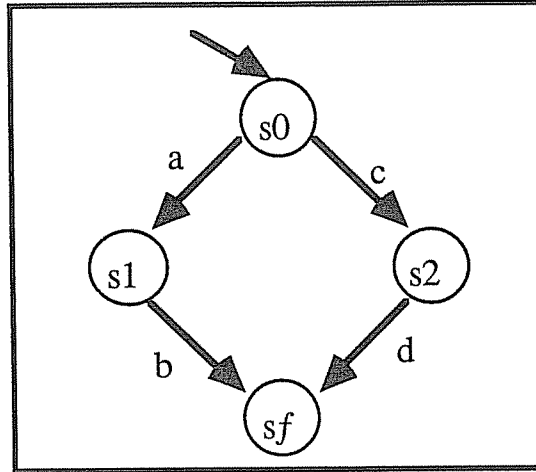


Figure 2.1: A Transition Graph

3. THE SPECIFICATION LANGUAGE, CTL

CTL is a modal logic, originally intended to specify properties of Kripke Structures. See section 4 below for an account of CTL and Kripke Structures. In this section, we define CTL and its interpretation on transition graphs.

CTL contains atomic propositions and a set of temporal operators. When interpreting CTL on transition graphs it is natural to let the atomic propositions be the actions in Σ . The set of CTL formulas is defined inductively as follows:

- (1) Every action $a \in \Sigma$ is a CTL formula.
- (2) if f_1 and f_2 are CTL formulas, then so are $\neg f_1, f_1 \wedge f_2, AX(f_1), EX(f_1), A[f_1 U f_2]$ and $E[f_1 U f_2]$.

The symbols \wedge and \neg have their usual meanings. X is the next time operator; the formula $AX(f_1)(EX(f_1))$ intuitively means that f_1 holds in every(in some) immediate successor of the current state. U is the until operator and

$A[f_1 U f_2](E[f_1 U f_2])$ intuitively means that for every computation path(for some computation path), there exists an initial prefix of the path such that f_2 holds at the last state of the prefix and f_1 holds at all other states along the prefix.

To exactly define truth of CTL-formulas in transition graphs we need the concept of a path.

Definition of a path in a TG:

Let TG be a transition graph and s a state in TG. A path from s is a sequence

$s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n \xrightarrow{a_{n+1}} \dots$ of transitions in TG which is either finite with a final state without any outgoing transitions, or infinite.

Now truth in transition graphs is defined as follows:

Let TG be a transition graph (M, s_0, Σ, Δ) and let $s \in M$ and $a \in \Sigma \cup \{\varepsilon\}$, where $\varepsilon \notin \Sigma$.

- ◆ $\langle\langle a, s \rangle\rangle \models_{TG} p$ iff $a=p$ for all atomic propositions p .
- ◆ $\langle\langle a, s \rangle\rangle \models_{TG} \neg f$ iff $\text{not}(\langle\langle a, s \rangle\rangle \models_{TG} f)$.
- ◆ $\langle\langle a, s \rangle\rangle \models_{TG} f_1 \vee f_2$ iff $(\langle\langle a, s \rangle\rangle \models_{TG} f_1)$ or $(\langle\langle a, s \rangle\rangle \models_{TG} f_2)$.
- ◆ $\langle\langle a, s \rangle\rangle \models_{TG} f_1 \wedge f_2$ iff $(\langle\langle a, s \rangle\rangle \models_{TG} f_1)$ and $(\langle\langle a, s \rangle\rangle \models_{TG} f_2)$.
- ◆ $\langle\langle a, s \rangle\rangle \models_{TG} AX(f)$ iff for all transitions, $(s \xrightarrow{b} s'') \in \Delta$ it holds that $\langle\langle b, s'' \rangle\rangle \models_{TG} f$.
- ◆ $\langle\langle a, s \rangle\rangle \models_{TG} EX(f)$ iff for some transition, $(s \xrightarrow{b} s'') \in \Delta$ it holds that $\langle\langle b, s'' \rangle\rangle \models_{TG} f$.
- ◆ $\langle\langle a, s \rangle\rangle \models_{TG} A[f_1 U f_2]$ iff for all paths $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots$ from s in TG, there exists an $i \geq 0$ such that
 - (1) $\langle\langle a_i, s_i \rangle\rangle \models_{TG} f_2$ and
 - (2) For all $j, 1 \leq j < i$ it holds $\langle\langle a_j, s_j \rangle\rangle \models_{TG} f_1$ and
 - (3) if $i \geq 1$ then $\langle\langle a, s \rangle\rangle \models_{TG} f_1$
- ◆ $\langle\langle a, s \rangle\rangle \models_{TG} E[f_1 U f_2]$ iff for some path $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots$ from s in TG, there exists an $i \geq 0$ such that
 - (1) $\langle\langle a_i, s_i \rangle\rangle \models_{TG} f_2$ and
 - (2) For all $j, 1 \leq j < i$ it holds $\langle\langle a_j, s_j \rangle\rangle \models_{TG} f_1$ and
 - (3) if $i \geq 1$ then $\langle\langle a, s \rangle\rangle \models_{TG} f_1$

If TG is a transition graph with initial state s_0 , then we define

$$\models_{TG} f \text{ iff } \langle\langle \varepsilon, s_0 \rangle\rangle \models_{TG} f$$

Here, ε denotes a dummy action which is not in Σ and is therefore not in the set of atomic propositions. It is meant to signify that "no action occurs". Note that if α is any action, then $\models_{TG} \alpha$ is false for any transition graph TG.

We also use the following abbreviations in writing CTL formulas:

- ◆ $AF(f) \equiv A[\text{True} \cup f]$ intuitively means that f holds in the future along every path from s_0 , i.e. f is inevitable.
- ◆ $EF(f) \equiv E[\text{True} \cup f]$ means that f holds in the future along some path from s_0 , i.e. f is potentially true.
- ◆ $AG(f) \equiv \neg EF(\neg f)$ means that f holds at every state on every path from s_0 , i.e. f holds globally.
- ◆ $EG(f) \equiv \neg AF(\neg f)$ means that there is a path along which f holds at every state.

Example:

In the transition graph in section 2, we have

$$\begin{aligned} \langle a, s_1 \rangle &\models_{TG} a. \\ \langle c, s_2 \rangle &\models_{TG} c. \\ &\models_{TG} AX(a \vee c). \end{aligned}$$

4. DEFINITION OF KRIPKE STRUCTURES

In this section, we introduce the mathematical definition of Kripke Structures and show how CTL- formulas are interpreted in such structures.

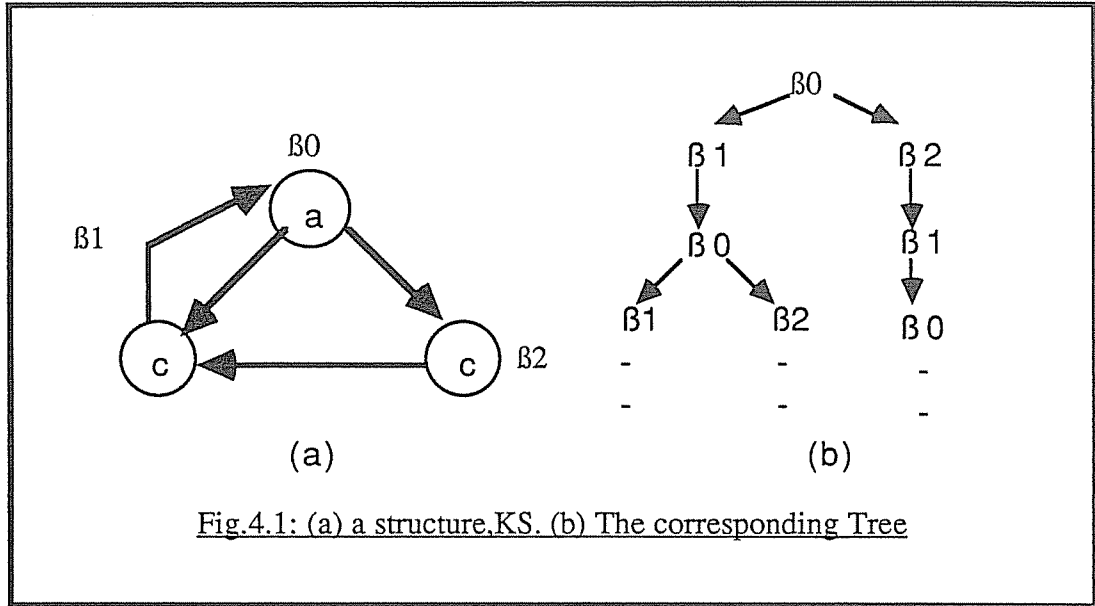
We assume AP is a set of atomic propositions.

Formally, a Kripke Structure is a quadruple $KS = (S, \beta_0, R, P)$ where:

- ◆ S is a finite set of states.
- ◆ β_0 is an initial state
- ◆ R is a binary relation on S ($R \subseteq S \times S$) which gives the possible transitions between states and must be total, i.e. $\forall x \in S \exists y \in S [(x, y) \in R]$.
- ◆ P is an assignment of atomic propositions to states i.e. $P: S \rightarrow 2^{AP}$.

A path is an infinite sequence of states $(\beta_0, \beta_1, \dots)$ where each transition (β_{i-1}, β_i) belongs to R . To each structure, KS , there is an infinite computation tree with β_0 as root and the arcs representing transitions in R . Figure 4.1 shows a simple Kripke Structure and the corresponding tree for the start state β_0 .

Here the assignment P is shown by writing $P(\beta)$ in the node corresponding to β_i for each β_i .



It is easy to see, in the figure above, that $AF(c)$ is true at state β_0 and that $AF(b)$ is false in state β_1 .

Truth of a CTL formula in a structure is indicated with the validity relation \models_{KS} . The definition is inductively as follows:

- ◆ $\beta_0 \models_{KS} p$ iff $p \in P(\beta_0)$ for all atomic proposition p .
- ◆ $\beta_0 \models_{KS} \neg f$ iff not $(\beta_0 \models_{KS} f)$.
- ◆ $\beta_0 \models_{KS} f1 \wedge f2$ iff $\beta_0 \models_{KS} f$ and $\beta_0 \models_{KS} f$.
- ◆ $\beta_0 \models_{KS} f1 \vee f2$ iff $\beta_0 \models_{KS} f$ or $\beta_0 \models_{KS} f$.
- ◆ $\beta_0 \models_{KS} AX(f)$ iff for all states t such that $(\beta_0, t) \in R$, $t \models_{KS} f$.
- ◆ $\beta_0 \models_{KS} EX(f)$ iff for some state t such that $(\beta_0, t) \in R$, $t \models_{KS} f$.
- ◆ $\beta_0 \models_{KS} A[f1 U f2]$ iff for all paths $(\beta_0, \beta_1, \dots)$,
 $\exists i \geq 0 [\beta_i \models_{KS} f2 \wedge \forall 0 \leq j < i [\beta_j \models_{KS} f1]]$.
- ◆ $\beta_0 \models_{KS} E[f1 U f2]$ iff for some path $(\beta_0, \beta_1, \dots)$,
 $\exists i \geq 0 [\beta_i \models_{KS} f2 \wedge \forall 0 \leq j < i [\beta_j \models_{KS} f1]]$.

We also define $\models_{KS} f$ to mean that $\beta_0 \models_{KS} f$ where β_0 is the initial state at KS. The defined operators, AF, EF, AG, EG are defined as in section 3.

5. TRANSFORMATION FROM TRANSITION GRAPH TO KRIPKE STRUCTURES

In this section, we define the transformation of transition graphs into Kripke Structures.

The main ideas of our construction is that every action in the Transition graph will be represented by a single distinct state in the Kripke Structure.

Let $TG = (M, s_0, \Sigma, \Delta)$ be a transition graph. The Kripke Structure corresponding to TG , written $KS(TG)$, is (S, β_0, R, P) where

$$\diamond S = \{\beta_0, \beta_f\} \cup \Delta \text{ where } \beta_0 \text{ and } \beta_f \text{ are distinct states and } \{\beta_0, \beta_f\} \notin \Delta :$$

$$\diamond R = \{((A -a \rightarrow A'), (A' -b \rightarrow A'')) : (A -a \rightarrow A'), (A' -b \rightarrow A'') \in \Delta\}$$

$$\cup \{((A -a \rightarrow A'), \beta_f) : (A -a \rightarrow A') \in \Delta \text{ and } A' \text{ has no outgoing transitions in } \Delta\}$$

$$\cup \{(\beta_0, (s_0 -a \rightarrow S)) : (s_0 -a \rightarrow S) \in \Delta\}$$

$$\cup \{(\beta_f, \beta_f)\}.$$

$$\diamond P(A -a \rightarrow A') = \{a\} \text{ for } (A -a \rightarrow A') \in \Delta \text{ and}$$

$$\diamond P(\beta_0) = P(\beta_f) = \emptyset.$$

Intuitively, the states in S are the transitions of TG plus an extra initial state β_0 and final state β_f . Due to the requirement that R be total, we must in R add a self-loop at β_f .

The main theoretical result that this transformation is compatible with the definition of truth of CTL formulas is shown in [6].

6. IMPLEMENTATION

In this section, we describe the architecture of our implementation of an algorithm for checking that a labeled transition graph satisfies a CTL formula. Our approach is a combination of two existing automatic tools: The Concurrency Workbench(CWB), and the Extended Model-Checker(EMC). We use CWB for generating transition graphs from CCS-expressions. We use the existing model

checking algorithm of EMC to check CTL formulas after they have been transformed.

We change the standard semantics for CTL in terms of communication actions and transform the transition graph into Kripke Structures so that it can be fed into EMC to use our logic to specify and verify desirable properties for communication protocol.

6.1 CCS

CCS, a calculus of communicating systems is widely used to specify and verify distributed computer systems. CCS can be regarded as a formalisation of the traditional finite state machine approach into an algebraic framework.

In CCS, processes are called agents. Agents are defined from communication events, internal events and a set of operators like action prefix, which is a limited form of sequential composition ($.$), parallel composition ($!$) and non-deterministic choice ($+$). Recursion is used to define cyclic behaviours.

The restriction operator (\backslash) is used to hide ports which are not to be available for external interaction. The syntax and semantics are described in [8].

6.2 CWB

The Concurrency Workbench is an automated tool for the analysis of communicating system that is developed at the University of Edinburgh. Any CCS-expression can be analyzed and defined in it. For example, deadlock and some other communication properties can be checked in this system. It is also possible to transform a given CCS-expression into a labeled transition graph. Our extension to the CWB is that we have defined a transformation from the labeled transition graph to Kripke Structures, as defined in section four, so that we can feed it into EMC to verify the desirable protocol properties.

6.3 EMC

The Extended Model Checker is an efficient model checking facility that does automatic verification of communication protocol properties using CTL-formula. A session with EMC starts with the loading of a description of the desired protocol in a Kripke Structures. Then CTL formulas are input, and EMC checks whether they are satisfied by the properties of the protocol. For formulas that are not satisfied, EMC attempts to output a counter-example path that illustrates why the formula does not hold.

6.4 OUR APPROACH

Our approach deals with the conversion of a transition graph into a finite state

Kripke Structure so that we can use the efficient algorithm described in the Extended Model Checker[2] for verifying that a CCS-agent meets a specification expressed in a propositional branching time temporal logic. Our approach is given in figure 6.1 and can be described as follows:

We created a converter in Standard ML, which takes any CCS-agent in the form of Transition Graph(described in section 2) as input from the CWB and produces a finite state Kripke Structure(described in section 4) so that we can feed it into the Extended Model Checker(EMC). Then CTL formulas are input to the EMC, and EMC verifies whether they are satisfied by the protocol.

6.5 DETAILS OF IMPLEMENTATION

The CWB consists of several modules with different functions which can be compiled separately. We need to look into the datastructures for the transition graph described in the module PolyGraph, which is our main input. Our transformation is done by adding some functions to the module Top. These two modules are described as follows:

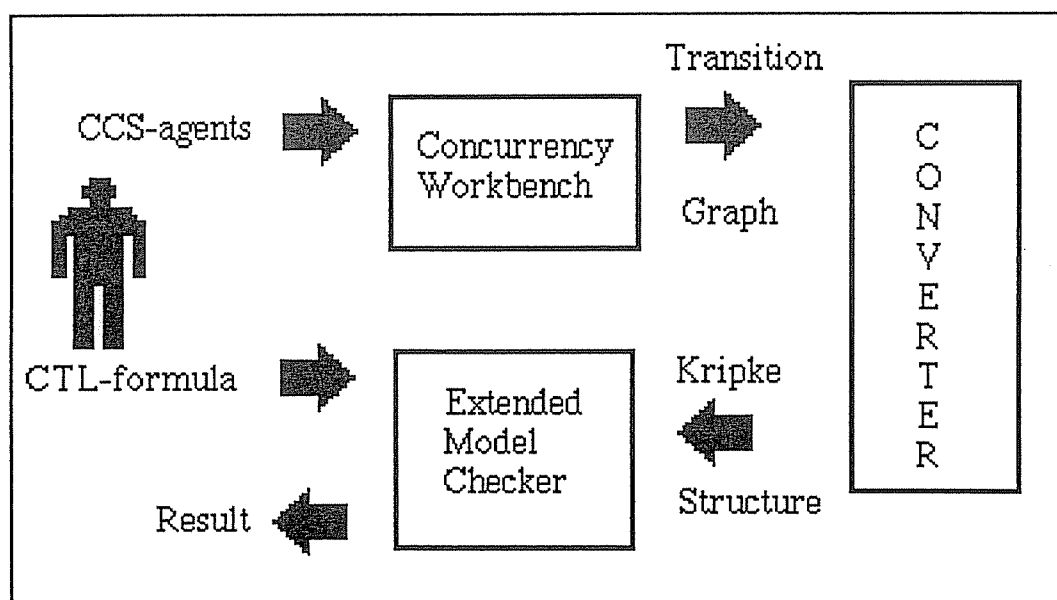


Figure 6.1: Architecture of our Approach

Module PolyGraph:

The transition graph that we need for our transformation to Kripke Structure is generated by the module PolyGraph.

The "mkgraph" function of this module takes an agent and creates a transition graph

which contains a finite number of distinct states as described in section two.

A transition graph in the PolyGraph is defined as a pair where the first element is a pointer to a state (the initial state), and the second element is a list of the states in the graph.

Each state is a record, defined as follows:

```
datatype 'a state =
  S of { name: agent,           ( Agent corresponding to state)
        id   : int,           ( Unique identifier)
        suc  : (act * ('a state list ref)) list ref, ( Successor of states)
        ..
        ..
        .. }
```

We are interested in every suc parameter of each state in a transition graph.

For example the suc parameter of the transition graph in figure 2.1 can be represented as follows:

```
Suc of s0 = ref [ (a * ref [s1]), (c * ref [s2])].
Suc of s1 = ref [ (b * ref [sf])].
Suc of s2 = ref [ (d * ref [sf])].
Suc of s3 = ref [].
```

The construction of the Kripke Structure starts with an initial state, β_0 , which contains no action. The successor to β_0 , is completely dependent upon the number of actions that the suc parameter of the initial state, s_0 contains. For example, there are two successors to β_0 as s_0 contains two actions (a and c) as stated by the suc parameter of s_0 . Any nil agent in transition graph (s_3 in this example) is always represented by the final state, β_f , in the Kripke Structure with a successor to itself. To reach our goal, we need only manipulation of such suc parameters of all the states created in the transition graph.

Module TOP:

We change the original module name "Top" to "MyTop" in our implementation. The module "MyTop" contains the following functions to interact with the user:

```
-- readcommand: read a command interactively.
   We added a new command "ctl" for the production of Kripke Structures.
-- minimize(unchanged): takes a graph and returns a minimal graph i.e. a bisimilar
```

graph where all bisimilar states have been collapsed to single states.

We introduced two new main functions in module MyTop for our implementation as follows:

i) The function **chan** :

The input to this function is the `suc` parameter of the datatype 'a state (described in module PolyGraph). It is a reference list of pairs. If the second element of a pair has several states then the function **chan** partitions the list into a new list in which every second element of the pair will have a reference list with only one state.

For example, if the `suc` of any state contains two states with same action as follows:

$$\text{Suc of any state} = \text{ref} [(a * \text{ref} [D1, D2])] .$$

Then the function `chan` splits it into a list of single reference as follows:

$$\text{ref} [(a * \text{ref} [D1]) , (a * \text{ref} [D2])] .$$

We need this transformation because every action in an agent becomes a state in our finite-state graph.

ii) The function **kripke_propocsuc**:

This function does several activities as follows:

1. it calls `mkgraph` of the Polygraph module and creates a list of states called "strct" which is formed from the CCS-expression. The list forms in the ascending order.
2. from the sorted list "strct", this function creates three lists.
 - the first list contains a list of pairs. The first element of the pair is the identifier number of a state and the second element is the total number of actions in the corresponding state.

For example, the first list for the agent described in figure 2.1 can be written as:

$$[(0,2),(1,1),(2,1),(3,0)] .$$

Let us call this list L1.

Let n be the total number of actions in the list L1. Then the total number of states in the finite graph will be $n+2$. If the CCS-expression contains the nil agent then the

total number of states in the finite graph will be $n + 1$.

- the second list contains a list of triples which are as follows:
 - i) the first element indicates the state number
 - ii) the second element represents the individual action and
 - iii) the list of the action of the successor states is denoted by the third element.

For figure 2.1, the second list can be written as :

$$[(2,a,[b]),(3,c,[d]),(4,b,[]),(5,d,[]),(6,\varepsilon,[])].$$

Let us call this list L2.

- the third list is also a list of triples as follows:
 - i) the first element denotes the state number
 - ii) the second element represents the individual action and
 - iii) the third element indicates the list of the successor states.

From L2, the third list can be viewed as :

$$[(2,a,[4]),(3,c,[5]),(4,b,[6]),(5,d,[6]),(6,\varepsilon,[6])].$$

Let us call this list L3.

The empty action denotes the final state in the graph if the agent nil exists.

Once the list L3 is created, we have all the information to establish a Kripke Structure. We denote the first element as a state number in the Kripke Structures which is labeled by the second element (a single action). The third element is the successor to the corresponding state number in the Kripke Structures.

The Kripke Structure generated by the CCS-expression (a.b.nil+c.d.nil) from the transition graph in section 2 consists of six states as shown in fig. 6.4.1.

3. it produces Input to EMC.

The function *kripke_propocsuc* creates a list from L3 with a special format which can be directly fed into the EMC. The format which is formed from the transition graph from figure 2.1 is shown below:

```
(setq nstates 6)
(store (prop 1) #())
```

```
(store (prop 2) #(a))  
(store (prop 3) #(c))  
(store (prop 4) #(b))  
(store (prop 5) #(d))  
(store (prop 6) #())  
(store (suc 1) #(2 3))  
(store (suc 2) #(4))  
(store (suc 3) #(5))  
(store (suc 4) #(6))  
(store (suc 5) #(6))  
(store (suc 6) #(6)).
```

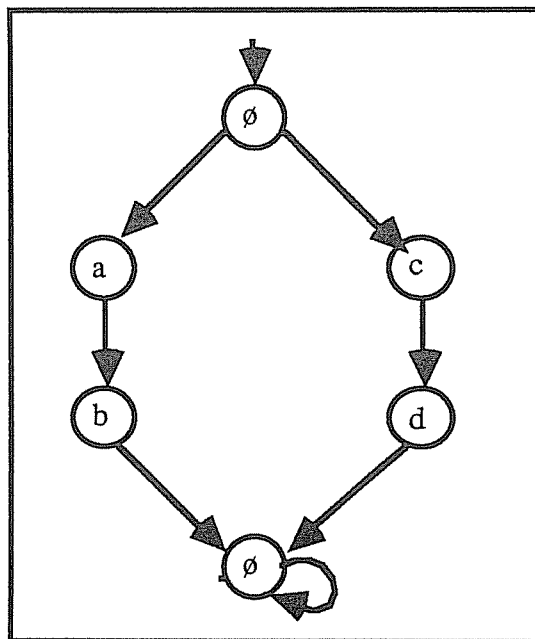


Figure 6.4.1: A Kripke Structure resulting from transformation.

Note: All the necessary files exists in the directory `/home/nuada/ahmed/mycwb/src`.
See Appendix 2 for operating instructions.
See Appendix 3 for Kripke Structures generated from the ABP.

6.6 MODIFICATIONS TO EMC

The lexical analyser of the Extended Model Checker has been changed to a certain extent from the original one to adjust the syntax of the CCS-expressions. For example, the original EMC used to accept ‘ ’ in the front of the action list which is also available in CCS-expressions to denote output event. So it has been changed to “#” as shown above in the front of the action list.

The modified version also presents an understandable prompt when a session starts.

7. USING EMC TO VERIFY THE ALTERNATING BIT PROTOCOL

In this section we consider an example to illustrate how the Extended Model Checker system might be used. The example that we selected is the Alternating Bit Protocol and that it is chosen from [1], which is a simple data link protocol and guarantees one way communication across a faulty medium. The service specification is that of a one place buffer and shown in the figure 7.1.

The protocol specification (fig. 7.2 and 7.3) consists of a sender and a receiver, communicating over two media M1 and M2 (in a lower layer) that may lose or corrupt (but not reorder) messages. The operation of the protocol is as follows.

The sender accepts a message to be transmitted on the channel send. It adds a one bit sequence number to the message (starting with 0 for the first message) and transmits it through the medium M1. The sender then awaits an acknowledgment with the same sequence number on the medium M2. After reception of the correct acknowledgment the procedure is repeated: a new message can be accepted for transmission. This time the sequence number is inverted. If the sender receives an acknowledgment, or no acknowledgment at all within a specified time, the sender assumes medium failure and retransmits the message on M1 (with the same sequence number). Retransmissions are repeated until a correct acknowledgment arrives.

The receiver acknowledges all messages from M1 by transmitting an acknowledgment on M2 with the same sequence number as the message. Each message with a sequence different from that of the immediately preceding one is reported on the channel rec. In addition, if a corrupt message arrives, medium failure is assumed and the last acknowledgment is retransmitted.

The media are one place one way buffers. A message (or acknowledgment) is accepted for transmission, whereafter one of three things may happen: the message is delivered intact, the message is delivered corrupt, or the message is lost (i.e. nothing happens at all).

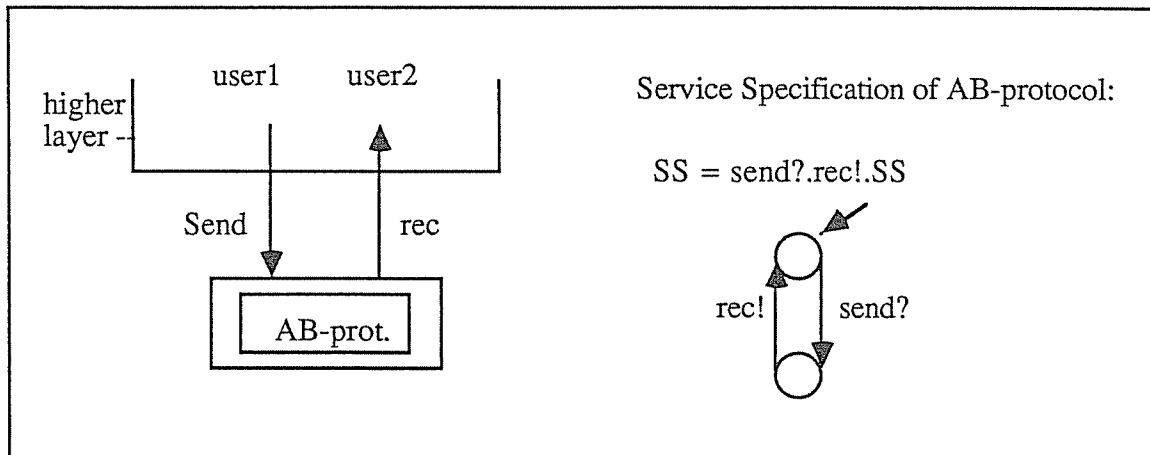


Figure 7.1: Service Specification of AB-protocol

In the formal protocol specification, some mechanism to detect message corruption is assumed. Thus, the generic event e is used to indicate a corrupt message, and ea indicates a corrupt acknowledgment. There is no need to represent the message contents, but sequence numbers are important for synchronisation. Thus, the events sm_0 and sm_1 indicate message transmissions from the sender to M1 with the sequence number 0 and 1 respectively. Similarly, mri,rai and asi indicate transmissions from M1 to receiver, from receiver to M2 and from M2 to sender respectively ($i \in \{0,1\}$).

The architecture of the protocol is depicted in fig. 7.2.

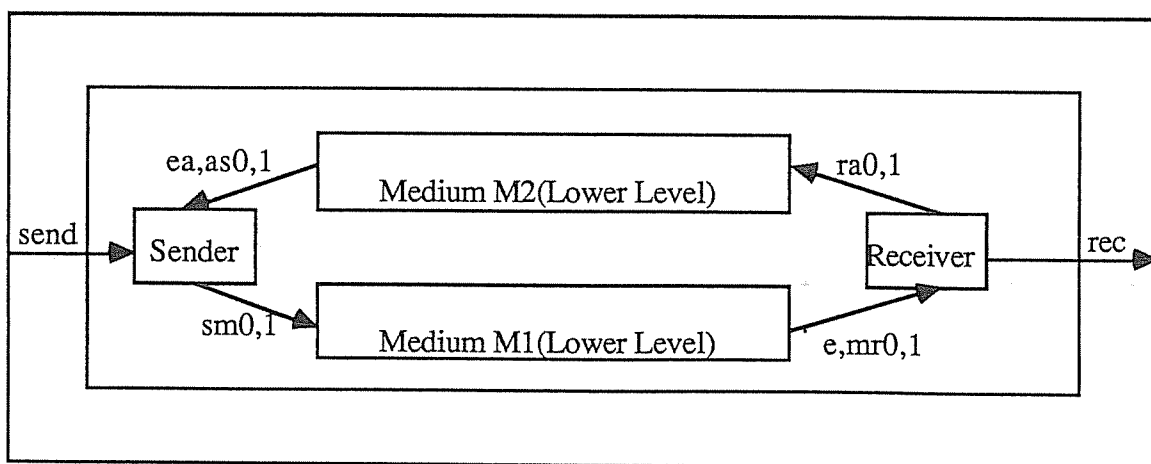


Figure 7.2: Architecture of AB-protocol Specification.

The timeout in the sender is modelled as a τ event (resulting in a state where a transmission can be made). Similarly, message losses in the media are modelled as τ

events.

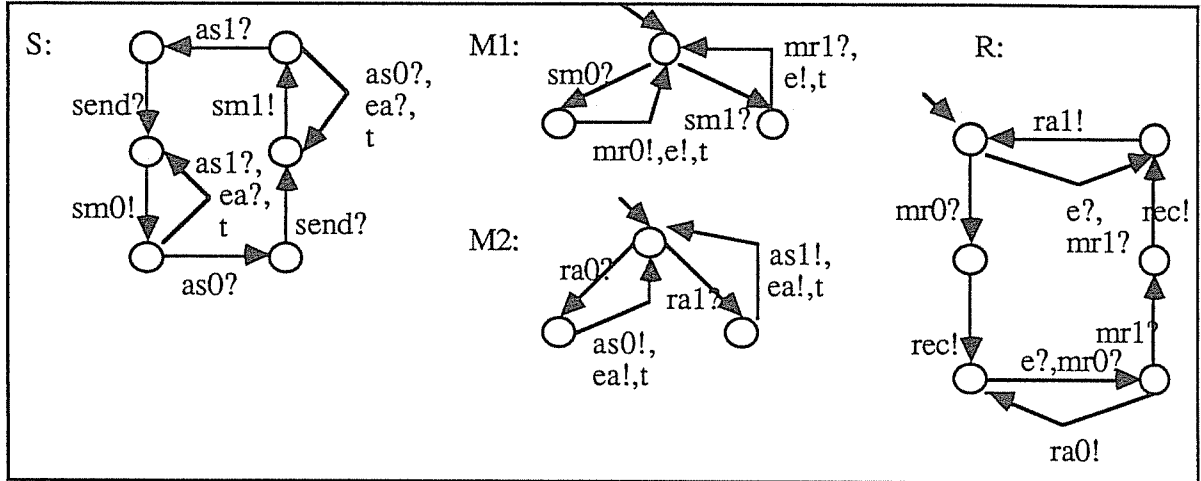


Figure 7.3: AB-protocol specification.

The protocol entities S (sender), R (receiver), M1 and M2 are modelled in fig. 7.3.

Recall the convention that for all events a , we write $a!$ for an output and $a?$ for an input.

The ABP in fig. 7.3 will become as follows when specified in CCS:

$$\begin{aligned}
 S &= \text{send?}.S' \\
 S' &= \text{sm0!}.(as1?.S'+ea?.S'+t.S'+as0?.\text{send?}.S'') \\
 S'' &= \text{sm1!}.(as0?.S''+ea?.S''+t.S''+as1?.S) \\
 M1 &= \text{sm0?}.(mr0!.M1+e!.M1+t.M1)+\text{sm1?}.(mr1!.M1+e!.M1+t.M1) \\
 M2 &= \text{ra0?}.(as0!.M2+ea!.M2+t.M2)+\text{ra1?}.(as1!.M2+ea!.M2+t.M2) \\
 R &= e?.\text{ra1!}.R+\text{mr1?}.\text{ra1!}.R+\text{mr0?}.\text{rec!}.R' \\
 R' &= \text{ra0!}.(mr0?.R'+e?.R'+\text{mr1?}.\text{rec!}.\text{ra1!}.R)
 \end{aligned}$$

Let L be the set $\{sm0, mr0, ra0, as0, sm1, mr1, ra1, as1, e, ea\}$ of internal communication channels in the AB-protocol. Then the protocol specification can be written as in CCS-expression, $P = (S \mid M1 \mid M2 \mid R) \setminus L$.

A global state graph is generated from the CCS-agent P , and the CTL expressions described in section 3 can be used to determine if the protocol satisfies its specifications. In the case of the ABP we can now test CTL-formulas such as:

$$\begin{aligned}
 &AG(\text{send} \rightarrow AF(\text{rec})). \\
 &AG(\text{send} \rightarrow EF(\text{rec})).
 \end{aligned}$$

Here the first formula intuitively states that every data message that is generated by

the sender is eventually accepted by the receiver. The second formula intuitively states that every data message that is generated by the sender is possibly accepted by the receiver. The first formula is not satisfied by the global state graph obtained from ABP because of infinite paths on which a message is lost or garbled each time that it is retransmitted. For this reason, we consider only fair paths i.e. paths in which a message cannot be lost each time transmitted. The fairness constraints are formulated as a set of states, in this case the state labeled send. Any infinite path has to pass this state infinitely many times. With this restriction, our model checking algorithm correctly determine that the state graph of the ABP satisfies its specification.

See Appendix1 (Transcript of Model checker Execution).

8. CONCLUSION

We have presented a temporal logic for communicating systems, and its integration into an existing automated tool: The Concurrency Workbench[5]. We have illustrated how one can change the semantics of an existing logic, and how one can adapt existing automated tools to implement decision procedures such as model checking.

ACKNOWLEDGMENTS

I want to thank Joachim Parrow and Bengt Jonsson and Fredrik Orava for enlightning fruitful discussions and comments, and to Ed Clarke for helping me understand the EMC.

REFERENCES

- [1] Joachim Parrow: Fairness Properties in Process Algebra, Ph. D. thesis, Department of Computer System, Uppsala University, Sweden. Uppsala 1985.
- [2] E.M. Clarke, E.A. Emerson, A.P. Sistla: Automatic Verification of finite-state Concurrent Systems Using Temporal Logic specifications, ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, April 1986, Pages 244-263.
- [3] M.C. Browne: An Improved Algorithm for the Automatic Verification of finite-State Systems Using Temporal Logic, Department of computer Science, Carnegie-Mellon University. CMU-CS-86-156.
- [4] Robin Milner: A Calculus of Communicating Systems. Volume 92 of Lecture Notes of Computer Science, Spring Verlag, 1980.

[5] Rance Cleaveland, Joachim Parrow and B. Steffan: The Concurrency Workbench. To be published in: 9th IFIP International Symposium on Protocol Specification, Testing, and Verification.

[6] Bengt Jonsson, Ahmed Hussain Khan, Joachim Parrow: Implementing a model Checking Algorithm by Adapting Existing Automated Tools, SICS, Kista, Sweden. Presented at the Workshop on Automatic verification methods for Finite State Systems organised at Grenoble, France, June 12-14, 1989.

[7] M.C. Browne, E.M. Clarke, O. Grumberg: Characterizing Kripke Structures In Temporal Logic, CMU, January 1987. CMU-CS-87-104.

[8] David Walker: Introduction to a calculus of Communicating Systems, University of Edinburgh, March 1987. ECS-LFCS-87-22.

APPENDIX 1: Transcript of Model Checker Execution

[Time is measured in 1/60 of a second. The first component measures total user CPU time. The second component measures total system CPU time.]

```
ahmed@anubis:> emc test.result
                CTL MODEL CHECKER (version A2.5)
```

Taking input from test.result...

```
EMC> .
Fairness constraint: .
```

```
time: (176 3)
```

```
|= AG(send -> AF('rec')).
The equation is FALSE.
```

```
time: (195 4)
```

```
|= AG(send -> EF('rec')).
The equation is TRUE.
```

```
time: (201 4)
```

```
|= .
End of Session.
```

```
ahmed@anubis:> emc test.result
                CTL MODEL CHECKER (version A2.5)
```

Taking input from test.result...

```
EMC> .
Fairness constraint: send.
Fairness constraint: .
```

```
time: (176 7)
```

```
|= AG(send -> AF('rec')).
The equation is TRUE.
```

```
time: (197 8)
```

```
|= AG( send -> EF('rec')).
The equation is TRUE.
```

```
time: (182 8)
```

```
|= .
End of Session.
ahmed@anubis:> exit
exit
```

script done on Wed Mar 29 11:16:01 1989

APPENDIX 2:
OPERATING INSTRUCTIONS FOR CONCURRENCY WORKBENCH
AND EXTENDED MODEL CHECKER

by

Ahmed Hussain Khan

25th january, 1989

1. Introduction

This note simply describes how to use Concurrency Workbench(CWB) for the production of finite state Kripke Structures so that Extended Model Checker can be used for the Computational Tree Logic formulas. It is assumed that readers of this note are acquainted with the syntax and symantics of Calculus of Communicating Systems and CTL-formulas.

Section one describes the operating Instructions of CWB and section two contains the user commands for the Extended Model Checker(EMC) written by Michael Browne.

Section one: Operating the CWB

The user operates the CWB by giving commands. There are commands for activating different analyses and for manipulating the global environments. Most commands require parameters; for example, the command *ctl* for producing Kripke Structures takes one agent. The user is always explicitly prompted for such parameters by the workbench. Parameters are terminated by the "return" key.

To produce Kripke Structure, it needs two file handling commands which provide for the saving and restoration of global environments between two sessions of the workbench and one environment commaand as follows:

- i. input file (*if*) Parameter: a file name: The file contains CCS-agents and workbench commands which updates the global environment. To add bindings to the agent the commands *bi* must appear in the file exactly as one would enter them in the workbench. The input file may contain an arbitrary mixture of such commands.
- ii. output file (*of*) Parameter:a file name. Directs most of the output which is usually sent to the terminal to the specified file.
- iii. *ctl* , Parameter: an agent. The result of this command is to produce a finite state Kripke Structure which can also be printed to the terminal unless output file is specified.

To terminate a workbench session, one executes the command *quit* , which may be abbreviated as *qu* .

Section Two : Operating EMC**NAME**

`emc` – extended model checker for CTL formulas

SYNOPSIS

`emc [-c] [-e] [-l] [-a] [filename]`

DESCRIPTION

Emc is an implementation of the extended model checker described in *Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach* by E.M.Clarke, E.A.Emerson, and A.P.Sistla.

The `-c` option is used to invoke a counter example facility. If this option is used, whenever the user inputs a formula that does not hold in the model, *emc* will attempt to find a path that demonstrates that the formula is false.

The `-e` option turns on expression echoing. This can be used to examine macro expansions.

The `-l` option allows *emc* to warn the user whenever a proposition that has not been declared is used. This doesn't prevent typos, but it does allow the user to find out about them.

The `-a` option enables the automatic declaration of propositions used while building the state machine.

The *filename* parameter is a file to be loaded silently before the user is prompted for input.

When execution begins, *emc* prompts the user for input with

>

indicating that there is no state machine in memory. The user must input a series of lisp-like commands (described in the COMMANDS section) to build a state machine and label each state with the propositions that are true there. Once the machine is completed, the user should enter a period to indicate completion.

Emc will then prompt the user with

Fairness constraint:

indicating that there are no fairness constraints in memory. The user may now enter one or more fairness constraints, each one terminated by a period. After all of the fairness constraints have been entered, the user must indicate this by entering a period.

After a slight delay, *emc* will print out the elapsed time and prompt for a CTL expression to verify with

⊨

The user should enter a formula to be tested, terminated by a period. *Emc* will test the formula, print the result, and prompt for another expression. This process continues until the user enters a period to indicate that he is done using the program.

COMMANDS

In the following command description, an *atom* can be either

1. a string of alphanumeric characters beginning with a letter,
2. a string of alphanumerics, "/", and "." preceded by a backquote (`),

3. any string enclosed in double quotes ("), or
4. any single printing character that has no special meaning. (Characters with special meaning are the quote characters, the CTL operators, all parentheses and brackets, comma, colon, and period.)

As mentioned in the DESCRIPTION section, *Emc* has three different prompts; a building prompt (">"), a fairness prompt ("Fairness constraint:"), and a normal prompt ("="). Different commands are accepted at each prompt.

At the building prompt, the program is expecting the user to enter a series of lisp-like commands that will construct a state machine. (Since the original implementation of the model checker was written in lisp, many of the commands used in this implementation are in the form of lisp statements.) The commands allowed at the ">" prompt are:

(setq nstates <number>)

Space is allocated for a new machine with <number> states.

(array prop t <number>)

(array suc t <number>)

These commands don't do anything. They are provided for compatibility with the earlier lisp version of the model checker.

(store (prop <number>) '(<list of atoms>))

The atoms in the list are defined to be the set of true propositions in state <number>. Any previous declarations for this state are deleted.

(store (suc <number>) '(<list of numbers>))

The numbers in the list are defined to be the successor states of state <number>. Any previous declarations for this state are deleted.

(new) The current state machine is deleted, and the user is prompted for commands to build a new state machine.

(load <atom>)

Input is taken from the file <atom> instead of the standard input.

At the fairness prompt, the user is expected to enter zero or more CTL formulas, each one terminated by a period. In addition to entering fairness constraints, the following commands can also be entered:

(restart) The current set of fairness constraints are deleted, and *emc* begins to prompt for a new set of constraints.

(new) The current state machine is deleted, and the user is prompted for commands to build a new state machine.

(load <atom>)

Input is read from the file <atom> instead of the standard input.

<atom> (<f1>,<f2>,...) := <expression>.

This defines <atom> to be a macro with <expression> as the body. <f1>, <f2>, ... are formal parameters which will be replaced with the actual expression parameters when the macro is actually called. Macros can be redefined by quoting the macro name (either by preceding it with a backquote or surrounding it in double quotes). Not quoting the atom when redefining a macro results in a syntax error.

infix <atom> (<f1>,<f2>) := <expression>.

This is a macro definition similar to the one above. The only difference is that this macro is a binary infix operator.

At the normal prompt, the user is expected to enter CTL formulas terminated by periods, to be verified. In addition, the commands allowed at the fairness prompt are also allowed at the normal prompt.

CTL EXPRESSIONS

Allowable CTL expressions are:

true

false

<atom> Atomic propositions and logical constants.

<atom2> / <atom2>

The atoms are concatenated to produce an atomic proposition.

<expr1> -> <expr2>

Logical implication

<expr1> | <expr2>

Logical disjunction.

<expr1> & <expr2>

Logical conjunction.

~ <expr>

Logical negation.

AX <expr>

X <expr>

<expr> holds in all successor states that are fair.

EX <expr>

<expr> holds in some successor state that is fair.

AF <expr>

F <expr>

Along all fair paths, <expr> holds at some future time.

EF <expr>

Along some fair path, <expr> holds at some future time.

AG <expr>

G <expr>

Along all fair paths globally, <expr> holds.

EG <expr>

Along some fair path globally, <expr> holds.

A[<expr1> U <expr2>]

[<expr1> U <expr2>]

Along all fair paths, <expr1> holds until <expr2> holds.

E[<expr1> U <expr2>]

Along some fair path, <expr1> holds until <expr2> holds.

A{ <atom>: <v1>, <v2>,... } <expr>

<expr> with <atom> replaced by <v1>, or <v2>, or... is always true (universal quantification macro).

E{ <atom>: <v1>, <v2>,... } <expr>

<expr> with <atom> replaced by <v1>, or <v2>, or... is true at least once (existential quantification macro).

<macro> (<expr1>, <expr2>,...)

This is a normal macro call. <expr1> replaces the first parameter in the macro definition, etc.

<expr1> <macro> <expr2>

This is also macro call. In this case, *<macro>* must have been defined as a binary infix operator by preceding the macro definition with *infix*. *<expr1>* replaces the first parameter in the macro definition, and *<expr2>* replaces the second parameter.

(*<expr>*)

Expressions can be parenthesised to change the order of evaluation.

All binary operators associate from left to right. The operators have the following precedence (from lowest to highest):

```
infix macro calls
->
|
&
~, AX, EX, AF, EF, AG, EG, A{ }, E{ }
/
normal macro calls
```

A[f U g] and E[f U g] are not on this list, because they are fully parenthesised.

Expressions can be several lines long. In all cases, expressions can only be terminated by a period.

EXAMPLE

Here is an example using the model checker to verify a flipflop. Comments enclosed in "{}" have been added and are not part of the input.

```
% emc
                                CTL MODEL CHECKER (C version 1.7)

> (setq nstates 4)
Space for 4 states allocated.
> (array prop t 5)
Array PROP dimensioned.
> (store (prop 1) '())           { Flipflop clear, no input }
Propositions of 1 added.
> (store (prop 2) '(R))         { Flipflop being reset }
Propositions of 2 added.
> (store (prop 3) '(S Q))       { Flipflop being set }
Propositions of 3 added.
> (store (prop 4) '(Q))         { Flipflop set, no input }
Propositions of 4 added.
> (array suc t 5)
Array SUC dimensioned.
> (store (suc 1) '(1 2 3))
Successors to 1 added.
> (store (suc 2) '(1 2))
Successors to 2 added.
> (store (suc 3) '(3 4))
Successors to 3 added.
> (store (suc 4) '(2 3 4))
Successors to 4 added.
> .
Fairness constraint: .

time: (10 18)

⊨ S -> A[Q U R & ~Q]. { Once set, the flipflop is set until reset }
```

The equation is TRUE.

time: (12 20)

$\models \text{EF } (S \ \& \ \sim Q)$. { S might not set the flipflop }

The equation is FALSE.

time: (13 20)

$\models \text{setres}(x,y,z) := x \rightarrow A[z \ \cup \ y]$.

Macro setres defined.

time: (16 20)

$\models \text{setres}(S,R,Q)$. { $S \rightarrow A[Q \ \cup \ R]$ }

The equation is TRUE.

time: (18 21)

$\models \text{setres}(R,S,\sim Q)$. { $R \rightarrow A[\sim Q \ \cup \ S]$ }

The equation is TRUE.

time: (19 25)

$\models \text{infix } = (x,y) := (x \rightarrow y) \ \& \ (y \rightarrow x)$. { Defining an equality operator }

Macro = defined.

time: (21 29)

$\models \text{EF } (S = Q)$. { Is it possible for S to equal Q? }

The equation is TRUE.

time: (23 30)

$\models \text{AG } (S = Q)$. { Is S always equal to Q? }

The equation is FALSE.

time: (24 32)

$\models .$

End of Session.

%

BUGS

The counter example facility has had problems in the past, and probably will in the future.

HISTORY

- 15-Nov-84 Michael Browne (mcb) at Carnegie-Mellon University
Fixed several bugs in the counter example facility. Replaced malloc with xmalloc to increase the speed. Added code to use breadth-first search to find counter examples for A[f U g] or E[f U g].
- 29-Oct-84 Michael Browne (mcb) at Carnegie-Mellon University
Added primitive error recovery. (At least the program doesn't die if you miscount parentheses!)
- 29-Jun-84 Michael Browne (mcb) at Carnegie-Mellon University
Fixed A[f U g] counter example bug so paths are fair. Altered load command so that ksp(1), dte(1), and lte(1) are invoked when necessary.
- 22-May-84 Michael Browne (mcb) at Carnegie-Mellon University
Added building prompt, infix macros, and (new) command. Added -e switch for debugging purposes. Fixed LEX grammar so that space is permitted between the letter ("A" or "E") and the bracket ("[" or "{") in the until expressions and quantification macros.
- 17-May-84 Michael Browne (mcb) at Carnegie-Mellon University
Created.

APPENDIX 3: Kripke Structures generated from the Alternating Bit Protocol.

```
(setq nstates 295)
(store (prop 1) #())
(store (prop 2) #(send))
(store (prop 3) #(t))
(store (prop 4) #(t))
(store (prop 5) #(t))
(store (prop 6) #(t))
(store (prop 7) #(t))
(store (prop 8) #(t))
(store (prop 9) #(t))
(store (prop 10) #(t))
(store (prop 11) #('rec))
(store (prop 12) #(t))
(store (prop 13) #('rec))
(store (prop 14) #(t))
(store (prop 15) #(t))
(store (prop 16) #('rec))
(store (prop 17) #(t))
(store (prop 18) #(t))
(store (prop 19) #(t))
(store (prop 20) #(t))
(store (prop 21) #(t))
(store (prop 22) #(t))
(store (prop 23) #(t))
(store (prop 24) #(t))
(store (prop 25) #(t))
(store (prop 26) #(t))
(store (prop 27) #(t))
(store (prop 28) #(t))
(store (prop 29) #(t))
(store (prop 30) #(t))
(store (prop 31) #(t))
(store (prop 32) #(t))
(store (prop 33) #(t))
(store (prop 34) #(t))
(store (prop 35) #(t))
(store (prop 36) #(t))
(store (prop 37) #(t))
(store (prop 38) #(t))
(store (prop 39) #(t))
(store (prop 40) #(t))
(store (prop 41) #(t))
(store (prop 42) #(t))
(store (prop 43) #(t))
(store (prop 44) #(t))
(store (prop 45) #(t))
(store (prop 46) #(t))
(store (prop 47) #(t))
(store (prop 48) #(t))
(store (prop 49) #(t))
(store (prop 50) #(t))
(store (prop 51) #(t))
(store (prop 52) #(t))
(store (prop 53) #(t))
(store (prop 54) #(t))
(store (prop 55) #(t))
(store (prop 56) #(t))
(store (prop 57) #(t))
(store (prop 58) #(send))
(store (prop 59) #(t))
```

```
(store (prop 60) #(t))
(store (prop 61) #(t))
(store (prop 62) #(t))
(store (prop 63) #(t))
(store (prop 64) #(t))
(store (prop 65) #(t))
(store (prop 66) #(t))
(store (prop 67) #(t))
(store (prop 68) #(t))
(store (prop 69) #(t))
(store (prop 70) #(t))
(store (prop 71) #(t))
(store (prop 72) #(t))
(store (prop 73) #(t))
(store (prop 74) #('rec))
(store (prop 75) #(t))
(store (prop 76) #(t))
(store (prop 77) #('rec))
(store (prop 78) #(t))
(store (prop 79) #(t))
(store (prop 80) #(t))
(store (prop 81) #(t))
(store (prop 82) #('rec))
(store (prop 83) #(t))
(store (prop 84) #('rec))
(store (prop 85) #(t))
(store (prop 86) #('rec))
(store (prop 87) #(t))
(store (prop 88) #(t))
(store (prop 89) #('rec))
(store (prop 90) #(t))
(store (prop 91) #(t))
(store (prop 92) #(t))
(store (prop 93) #(t))
(store (prop 94) #(t))
(store (prop 95) #(t))
(store (prop 96) #(t))
(store (prop 97) #(t))
(store (prop 98) #(t))
(store (prop 99) #(t))
(store (prop 100) #(t))
(store (prop 101) #(t))
(store (prop 102) #(t))
(store (prop 103) #(t))
(store (prop 104) #(t))
(store (prop 105) #(t))
(store (prop 106) #(t))
(store (prop 107) #(t))
(store (prop 108) #(t))
(store (prop 109) #(t))
(store (prop 110) #(t))
(store (prop 111) #(t))
(store (prop 112) #(send))
(store (prop 113) #(t))
(store (prop 114) #(t))
(store (prop 115) #(t))
(store (prop 116) #(t))
(store (prop 117) #(t))
(store (prop 118) #(t))
(store (prop 119) #(t))
(store (prop 120) #(t))
(store (prop 121) #(t))
(store (prop 122) #(t))
```

```
(store (prop 123) #(t))
(store (prop 124) #(t))
(store (prop 125) #(t))
(store (prop 126) #(t))
(store (prop 127) #(t))
(store (prop 128) #'(rec))
(store (prop 129) #(t))
(store (prop 130) #(t))
(store (prop 131) #'(rec))
(store (prop 132) #(t))
(store (prop 133) #(t))
(store (prop 134) #(t))
(store (prop 135) #(t))
(store (prop 136) #'(rec))
(store (prop 137) #(t))
(store (prop 138) #(t))
(store (prop 139) #(t))
(store (prop 140) #(t))
(store (prop 141) #(t))
(store (prop 142) #(t))
(store (prop 143) #(t))
(store (prop 144) #(t))
(store (prop 145) #(t))
(store (prop 146) #(t))
(store (prop 147) #(t))
(store (prop 148) #(t))
(store (prop 149) #(t))
(store (prop 150) #'(rec))
(store (prop 151) #(t))
(store (prop 152) #(t))
(store (prop 153) #(t))
(store (prop 154) #'(rec))
(store (prop 155) #(t))
(store (prop 156) #(t))
(store (prop 157) #(t))
(store (prop 158) #(t))
(store (prop 159) #(t))
(store (prop 160) #(t))
(store (prop 161) #(t))
(store (prop 162) #(t))
(store (prop 163) #(t))
(store (prop 164) #(t))
(store (prop 165) #(t))
(store (prop 166) #(t))
(store (prop 167) #(t))
(store (prop 168) #(t))
(store (prop 169) #(t))
(store (prop 170) #(t))
(store (prop 171) #(t))
(store (prop 172) #(t))
(store (prop 173) #(t))
(store (prop 174) #(t))
(store (prop 175) #(t))
(store (prop 176) #(t))
(store (prop 177) #(t))
(store (prop 178) #(t))
(store (prop 179) #(t))
(store (prop 180) #(t))
(store (prop 181) #(t))
(store (prop 182) #(t))
(store (prop 183) #(t))
(store (prop 184) #(t))
(store (prop 185) #(t))
```

```
(store (prop 186) #(send))
(store (prop 187) #(t))
(store (prop 188) #(send))
(store (prop 189) #(t))
(store (prop 190) #(send))
(store (prop 191) #(t))
(store (prop 192) #(t))
(store (prop 193) #(t))
(store (prop 194) #(send))
(store (prop 195) #(t))
(store (prop 196) #(send))
(store (prop 197) #(t))
(store (prop 198) #(t))
(store (prop 199) #(t))
(store (prop 200) #(t))
(store (prop 201) #(t))
(store (prop 202) #(t))
(store (prop 203) #(t))
(store (prop 204) #(t))
(store (prop 205) #(t))
(store (prop 206) #(t))
(store (prop 207) #(t))
(store (prop 208) #(t))
(store (prop 209) #(t))
(store (prop 210) #(t))
(store (prop 211) #(t))
(store (prop 212) #(t))
(store (prop 213) #(t))
(store (prop 214) #(t))
(store (prop 215) #(t))
(store (prop 216) #(t))
(store (prop 217) #(t))
(store (prop 218) #(t))
(store (prop 219) #(t))
(store (prop 220) #(t))
(store (prop 221) #(t))
(store (prop 222) #('rec))
(store (prop 223) #(t))
(store (prop 224) #(t))
(store (prop 225) #(t))
(store (prop 226) #(t))
(store (prop 227) #(t))
(store (prop 228) #(t))
(store (prop 229) #(t))
(store (prop 230) #(t))
(store (prop 231) #(t))
(store (prop 232) #(t))
(store (prop 233) #(t))
(store (prop 234) #(t))
(store (prop 235) #(t))
(store (prop 236) #('rec))
(store (prop 237) #(t))
(store (prop 238) #(t))
(store (prop 239) #(t))
(store (prop 240) #(t))
(store (prop 241) #(t))
(store (prop 242) #(t))
(store (prop 243) #(t))
(store (prop 244) #(t))
(store (prop 245) #(t))
(store (prop 246) #(t))
(store (prop 247) #(t))
(store (prop 248) #(t))
```

```
(store (prop 249) #(t))
(store (prop 250) #(t))
(store (prop 251) #(t))
(store (prop 252) #(t))
(store (prop 253) #(t))
(store (prop 254) #(t))
(store (prop 255) #(t))
(store (prop 256) #(t))
(store (prop 257) #(t))
(store (prop 258) #(t))
(store (prop 259) #(t))
(store (prop 260) #(t))
(store (prop 261) #(t))
(store (prop 262) #(t))
(store (prop 263) #(t))
(store (prop 264) #(t))
(store (prop 265) #(t))
(store (prop 266) #(t))
(store (prop 267) #(t))
(store (prop 268) #(t))
(store (prop 269) #(t))
(store (prop 270) #(t))
(store (prop 271) #(t))
(store (prop 272) #(t))
(store (prop 273) #(t))
(store (prop 274) #(t))
(store (prop 275) #(t))
(store (prop 276) #(t))
(store (prop 277) #(t))
(store (prop 278) #(send))
(store (prop 279) #(t))
(store (prop 280) #(send))
(store (prop 281) #(t))
(store (prop 282) #(send))
(store (prop 283) #(send))
(store (prop 284) #(t))
(store (prop 285) #(t))
(store (prop 286) #(t))
(store (prop 287) #(send))
(store (prop 288) #(t))
(store (prop 289) #(send))
(store (prop 290) #(t))
(store (prop 291) #(t))
(store (prop 292) #(t))
(store (prop 293) #(t))
(store (prop 294) #(t))
(store (prop 295) #(t))
(store (suc 1) #( 2 ))
(store (suc 2) #(3 ))
(store (suc 3) #(4 5 6 7 ))
(store (suc 4) #(180 ))
(store (suc 5) #(167 168 ))
(store (suc 6) #(154 155 ))
(store (suc 7) #(8 9 10 ))
(store (suc 8) #(3 ))
(store (suc 9) #(117 118 ))
(store (suc 10) #(11 12 ))
(store (suc 11) #(20 21 ))
(store (suc 12) #(13 14 15 ))
(store (suc 13) #(22 23 24 ))
(store (suc 14) #(154 155 ))
(store (suc 15) #(16 17 ))
(store (suc 16) #(18 19 ))
```

```
(store (suc 17) #(11 12 ))
(store (suc 18) #(46 47 48 ))
(store (suc 19) #(20 21 ))
(store (suc 20) #(38 39 ))
(store (suc 21) #(22 23 24 ))
(store (suc 22) #(40 41 42 43 44 45 ))
(store (suc 23) #(25 26 ))
(store (suc 24) #(18 19 ))
(store (suc 25) #(27 28 29 30 ))
(store (suc 26) #(20 21 ))
(store (suc 27) #(37 ))
(store (suc 28) #(282 ))
(store (suc 29) #(38 39 ))
(store (suc 30) #(31 ))
(store (suc 31) #(32 33 34 ))
(store (suc 32) #(37 ))
(store (suc 33) #(25 26 ))
(store (suc 34) #(35 36 ))
(store (suc 35) #(31 ))
(store (suc 36) #(20 21 ))
(store (suc 37) #(31 ))
(store (suc 38) #(31 ))
(store (suc 39) #(40 41 42 43 44 45 ))
(store (suc 40) #(32 33 34 ))
(store (suc 41) #(27 28 29 30 ))
(store (suc 42) #(292 293 294 295 ))
(store (suc 43) #(289 290 291 ))
(store (suc 44) #(46 47 48 ))
(store (suc 45) #(35 36 ))
(store (suc 46) #(35 36 ))
(store (suc 47) #(38 39 ))
(store (suc 48) #(49 50 ))
(store (suc 49) #(20 21 ))
(store (suc 50) #(51 52 53 54 55 ))
(store (suc 51) #(22 23 24 ))
(store (suc 52) #(292 293 294 295 ))
(store (suc 53) #(58 59 60 ))
(store (suc 54) #(56 57 ))
(store (suc 55) #(18 19 ))
(store (suc 56) #(18 19 ))
(store (suc 57) #(49 50 ))
(store (suc 58) #(61 62 ))
(store (suc 59) #(283 284 285 286 ))
(store (suc 60) #(278 279 ))
(store (suc 61) #(273 274 275 ))
(store (suc 62) #(63 64 ))
(store (suc 63) #(265 266 ))
(store (suc 64) #(65 66 67 ))
(store (suc 65) #(267 268 269 270 271 272 ))
(store (suc 66) #(251 252 ))
(store (suc 67) #(68 69 ))
(store (suc 68) #(70 71 72 73 ))
(store (suc 69) #(63 64 ))
(store (suc 70) #(261 262 263 ))
(store (suc 71) #(265 266 ))
(store (suc 72) #(240 241 ))
(store (suc 73) #(74 75 76 ))
(store (suc 74) #(227 228 ))
(store (suc 75) #(84 85 ))
(store (suc 76) #(77 78 79 80 81 ))
(store (suc 77) #(229 230 231 232 ))
(store (suc 78) #(86 87 88 ))
(store (suc 79) #(236 237 238 239 ))
```

```
(store (suc 80) #(222 223 224 ))
(store (suc 81) #(82 83 ))
(store (suc 82) #(98 99 ))
(store (suc 83) #(84 85 ))
(store (suc 84) #(93 94 ))
(store (suc 85) #(86 87 88 ))
(store (suc 86) #(95 96 97 ))
(store (suc 87) #(89 90 ))
(store (suc 88) #(82 83 ))
(store (suc 89) #(91 92 ))
(store (suc 90) #(84 85 ))
(store (suc 91) #(218 219 220 221 ))
(store (suc 92) #(93 94 ))
(store (suc 93) #(203 204 ))
(store (suc 94) #(95 96 97 ))
(store (suc 95) #(205 206 207 208 209 210 ))
(store (suc 96) #(91 92 ))
(store (suc 97) #(98 99 ))
(store (suc 98) #(100 101 102 ))
(store (suc 99) #(93 94 ))
(store (suc 100) #(211 212 ))
(store (suc 101) #(203 204 ))
(store (suc 102) #(103 104 ))
(store (suc 103) #(93 94 ))
(store (suc 104) #(105 106 107 108 109 ))
(store (suc 105) #(95 96 97 ))
(store (suc 106) #(199 200 201 202 ))
(store (suc 107) #(112 113 114 ))
(store (suc 108) #(110 111 ))
(store (suc 109) #(98 99 ))
(store (suc 110) #(98 99 ))
(store (suc 111) #(103 104 ))
(store (suc 112) #(115 116 ))
(store (suc 113) #(190 191 192 193 ))
(store (suc 114) #(186 187 ))
(store (suc 115) #(181 182 183 ))
(store (suc 116) #(117 118 ))
(store (suc 117) #(172 173 ))
(store (suc 118) #(119 120 121 ))
(store (suc 119) #(174 175 176 177 178 179 ))
(store (suc 120) #(167 168 ))
(store (suc 121) #(122 123 ))
(store (suc 122) #(124 125 126 127 ))
(store (suc 123) #(117 118 ))
(store (suc 124) #(8 9 10 ))
(store (suc 125) #(172 173 ))
(store (suc 126) #(156 157 ))
(store (suc 127) #(128 129 130 ))
(store (suc 128) #(141 142 ))
(store (suc 129) #(11 12 ))
(store (suc 130) #(131 132 133 134 135 ))
(store (suc 131) #(143 144 145 146 ))
(store (suc 132) #(13 14 15 ))
(store (suc 133) #(150 151 152 153 ))
(store (suc 134) #(136 137 138 ))
(store (suc 135) #(16 17 ))
(store (suc 136) #(139 140 ))
(store (suc 137) #(16 17 ))
(store (suc 138) #(128 129 130 ))
(store (suc 139) #(18 19 ))
(store (suc 140) #(141 142 ))
(store (suc 141) #(20 21 ))
(store (suc 142) #(143 144 145 146 ))
```

```
(store (suc 143) #(22 23 24 ))
(store (suc 144) #(147 148 149 ))
(store (suc 145) #(139 140 ))
(store (suc 146) #(18 19 ))
(store (suc 147) #(25 26 ))
(store (suc 148) #(141 142 ))
(store (suc 149) #(20 21 ))
(store (suc 150) #(147 148 149 ))
(store (suc 151) #(154 155 ))
(store (suc 152) #(128 129 130 ))
(store (suc 153) #(11 12 ))
(store (suc 154) #(25 26 ))
(store (suc 155) #(11 12 ))
(store (suc 156) #(117 118 ))
(store (suc 157) #(158 159 160 161 ))
(store (suc 158) #(119 120 121 ))
(store (suc 159) #(164 165 166 ))
(store (suc 160) #(162 163 ))
(store (suc 161) #(122 123 ))
(store (suc 162) #(122 123 ))
(store (suc 163) #(156 157 ))
(store (suc 164) #(167 168 ))
(store (suc 165) #(156 157 ))
(store (suc 166) #(117 118 ))
(store (suc 167) #(169 170 171 ))
(store (suc 168) #(117 118 ))
(store (suc 169) #(180 ))
(store (suc 170) #(172 173 ))
(store (suc 171) #(3 ))
(store (suc 172) #(3 ))
(store (suc 173) #(174 175 176 177 178 179 ))
(store (suc 174) #(4 5 6 7 ))
(store (suc 175) #(169 170 171 ))
(store (suc 176) #(164 165 166 ))
(store (suc 177) #(150 151 152 153 ))
(store (suc 178) #(124 125 126 127 ))
(store (suc 179) #(8 9 10 ))
(store (suc 180) #(3 ))
(store (suc 181) #(184 185 ))
(store (suc 182) #(172 173 ))
(store (suc 183) #(156 157 ))
(store (suc 184) #(3 ))
(store (suc 185) #(117 118 ))
(store (suc 186) #(117 118 ))
(store (suc 187) #(188 189 ))
(store (suc 188) #(172 173 ))
(store (suc 189) #(2 ))
(store (suc 190) #(181 182 183 ))
(store (suc 191) #(196 197 198 ))
(store (suc 192) #(188 189 ))
(store (suc 193) #(194 195 ))
(store (suc 194) #(156 157 ))
(store (suc 195) #(186 187 ))
(store (suc 196) #(184 185 ))
(store (suc 197) #(2 ))
(store (suc 198) #(186 187 ))
(store (suc 199) #(91 92 ))
(store (suc 200) #(186 187 ))
(store (suc 201) #(103 104 ))
(store (suc 202) #(93 94 ))
(store (suc 203) #(213 ))
(store (suc 204) #(205 206 207 208 209 210 ))
(store (suc 205) #(214 215 216 ))
```

```
(store (suc 206) #(218 219 220 221 ))
(store (suc 207) #(199 200 201 202 ))
(store (suc 208) #(196 197 198 ))
(store (suc 209) #(100 101 102 ))
(store (suc 210) #(211 212 ))
(store (suc 211) #(213 ))
(store (suc 212) #(93 94 ))
(store (suc 213) #(214 215 216 ))
(store (suc 214) #(217 ))
(store (suc 215) #(91 92 ))
(store (suc 216) #(211 212 ))
(store (suc 217) #(213 ))
(store (suc 218) #(217 ))
(store (suc 219) #(2 ))
(store (suc 220) #(203 204 ))
(store (suc 221) #(213 ))
(store (suc 222) #(225 226 ))
(store (suc 223) #(82 83 ))
(store (suc 224) #(74 75 76 ))
(store (suc 225) #(98 99 ))
(store (suc 226) #(227 228 ))
(store (suc 227) #(93 94 ))
(store (suc 228) #(229 230 231 232 ))
(store (suc 229) #(95 96 97 ))
(store (suc 230) #(233 234 235 ))
(store (suc 231) #(225 226 ))
(store (suc 232) #(98 99 ))
(store (suc 233) #(91 92 ))
(store (suc 234) #(227 228 ))
(store (suc 235) #(93 94 ))
(store (suc 236) #(233 234 235 ))
(store (suc 237) #(89 90 ))
(store (suc 238) #(74 75 76 ))
(store (suc 239) #(84 85 ))
(store (suc 240) #(63 64 ))
(store (suc 241) #(242 243 244 245 ))
(store (suc 242) #(65 66 67 ))
(store (suc 243) #(248 249 250 ))
(store (suc 244) #(246 247 ))
(store (suc 245) #(68 69 ))
(store (suc 246) #(68 69 ))
(store (suc 247) #(240 241 ))
(store (suc 248) #(251 252 ))
(store (suc 249) #(240 241 ))
(store (suc 250) #(63 64 ))
(store (suc 251) #(253 254 255 ))
(store (suc 252) #(63 64 ))
(store (suc 253) #(264 ))
(store (suc 254) #(265 266 ))
(store (suc 255) #(256 ))
(store (suc 256) #(257 258 259 260 ))
(store (suc 257) #(264 ))
(store (suc 258) #(251 252 ))
(store (suc 259) #(89 90 ))
(store (suc 260) #(261 262 263 ))
(store (suc 261) #(256 ))
(store (suc 262) #(63 64 ))
(store (suc 263) #(84 85 ))
(store (suc 264) #(256 ))
(store (suc 265) #(256 ))
(store (suc 266) #(267 268 269 270 271 272 ))
(store (suc 267) #(257 258 259 260 ))
(store (suc 268) #(253 254 255 ))
```

(store (suc 269) #(248 249 250))
(store (suc 270) #(236 237 238 239))
(store (suc 271) #(70 71 72 73))
(store (suc 272) #(261 262 263))
(store (suc 273) #(276 277))
(store (suc 274) #(265 266))
(store (suc 275) #(240 241))
(store (suc 276) #(256))
(store (suc 277) #(63 64))
(store (suc 278) #(63 64))
(store (suc 279) #(280 281))
(store (suc 280) #(265 266))
(store (suc 281) #(282))
(store (suc 282) #(256))
(store (suc 283) #(273 274 275))
(store (suc 284) #(289 290 291))
(store (suc 285) #(280 281))
(store (suc 286) #(287 288))
(store (suc 287) #(240 241))
(store (suc 288) #(278 279))
(store (suc 289) #(276 277))
(store (suc 290) #(282))
(store (suc 291) #(278 279))
(store (suc 292) #(25 26))
(store (suc 293) #(278 279))
(store (suc 294) #(49 50))
(store (suc 295) #(20 21))