Contents

Preface

This document is intended to be a practical guide to the arts and crafts of specifying and verifying asynchronous systems in CCS. We assume that you are familiar with the CCS notation (either through Walker's introductory paper [Wal87] or Milner's teaching text [Mil89]) and that you have used CWB — the Concurrency Workbench [Mol91] — a little.

What we want to put across is what you don't learn from these documents: how to test specifications and gain confidence in their correctness — if you like, the methodology behind CCS.

Proving the equivalence of processes is a hard business. It is important to realise that the methodology is not to just specifying a design, provide an implementation and then using the workbench to prove their equivalence. Getting a specification "correct" is very hard in its own right and that's where the bulk of the effort should be put. The proper way to proceed is to use CWB to test the consequences of a specification thoroughly before attempting an implementation. The right tools, namely Hennessy-Milner Logic (HML) and the modal-μ calculus, are built into the CWB but there is a dearth of introductory material suitable for beginners. For further reading, we suggest [Win91, SW91].

It is the purpose of these notes to give you the background to these logics, their intuition, and how to use them. All of the examples are taken from asynchronous hardware design. Asynchronous circuits do very little else other than communicate and thus they form very concentrated objects for study. In addition, since CCS is not especially tailored to hardware, the style we teach transfers immediately to network protocols and real time software.

In chapter 1 we introduce you to some basic asynchronous circuits. In chapter 2 we introduce Hennessy-Milner Logic and how to apply it. We cover modal logics, HML, and some modal characterisation theorems. In

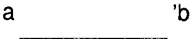
chapter 3 we introduce, motivate, and give the intuition behind recursive equations and minimum and maximum fix points. We then introduce the syntax and semantics of the modal-μ calculus. How can we use it? we explain the concepts of safety and liveness, show how to write "property" macros in the modal-μ calculus, and wind up with a host of illustrative examples.

Chapter 1

Some Hardware Circuits in CCS

In this chapter we specify a number of simple asynchronous circuits in CCS. The syntax and semantics of CCS are summarized as an appendix. For a more detailed exposition see [Mil89]. We observe the following naming conventions: each internal and external wire in the design is given a unique name. When we use that name in a process description, it will be interpreted as an event (a stimulus or a signal transition) on that line. External wires are connected to the environment. Each external wire may either receive signals from the environment or send signals to the environment but not both. We interpret an action **var** as an input event on line **var**, and an action **'var** as an output event on line **var**. Internal lines lead from one device to another. If they are named **'var** at the output end they are named **var** at the input end. Circuits take unpredictable and arbitrary times to fire; signals take unpredictable and arbitrary times to travel along wires.

1. `WIRE = a.'b.WIRE`



WIRE

A wire transmits a signal (one at a time) from one end to the other with an unpredictable delay. More formally, it receives an input on **a**, and later fires an output on **'b** before evolving into a wire again.

It may not receive another event on a until after it has output on 'b.

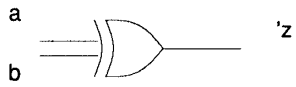
2. `IWIRE = 'b.a.IWIRE`

An iwire fires its output then behaves like a wire. Iwires are useful fixes for ensuring that specific lines "go first".

3. `FORK = a.('b.'c.FORK + 'c.'b.FORK)`

The fork first receives an input on a then "copies" it on b and on c but in random (unpredictable) order. It evolves back into a fork after both copies have been sent. Forks are used to duplicate signals.

4. `MERGE2 = a.'z.MERGE2 + b.'z.MERGE2`



MERGE

The merge2 has two inputs and a single output. The output fires after a signal on either input.

5. `C2 = a.b.'z.C2 + b.a.'z.C2`

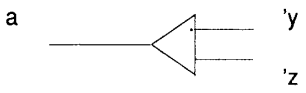
The two-input Muller C-element acts as a rendezvous. It fires when its both its inputs have fired. Once an input has fired it must remain stable until the output has fired.

It is easy to describe a "wobbly-input" C-element which may accept and then cancel and input.

```
C2' = a.(b.'z.C2' + a.C2') + b.(a.'z.C2' + b.C2')
```

Whether you actually want a device that behaves like this is another point.

6. `TOGGLE = a.'y.a.'z.TOGGLE`



TOGGLE

The toggle fires its outputs alternately after receiving an input. It outputs on **y** after receiving odd numbered input stimuli; and on **z** after receiving even numbered input stimuli.

7.
```
SEM = 'g.p.SEM
U1  = INITIAL1.g.USE1.'p.FINAL1
U2  = INITIAL2.g.USE2.'p.FINAL2
```

Here we have a description of a semaphore and two users. The semaphore is not a hardware device, but understanding how it works and how it should be used is crucial to our definitions of the join and the arbiter below.

A semaphore is used to protect a resource from multiple users. It contains a token which is granted to a single user before he uses the resource, kept whilst he uses it, and then put back when his use of the resource is completed. The semaphore sees to it that only one user at a time may have the token. Each user is honour bound to follow the protocol.

It is instructive to see what can happen if malicious users enter the fray, for example

```
W1  = g.Nil
W2  = 'p.Nil
W3  = 'p.g.Nil
```

8.
```
JOIN2 = ( A1 | A2 | N ) \{g, p}
where
   A1 = a1.g.'b1.'p.A1
   A2 = a2.g.'b2.'p.A2
    N = n.'g.p.N
```

The JOIN2 circuit has two inputs and two outputs. On receiving an **a1** it will eventually output a **b1**. On receiving an **a2** it will eventually output a **b2**. But it can only attend to one of these requests at a time, hence the semaphore. The device is ready for action when it receives stimulus on **n**.

Notice the style of definition: we have written a protocol for each line (A1 and A2), added a semaphore to ensure one user at a time, and then composed them in parallel hiding the semaphore accesses. The style generalises to joins of any size.

9.
```
ARBITER = ( A1 | A2 | N ) \{g, p}
where
   A1 = r1.g.'g1.d1.p.'a1.A1
   A2 = r2.g.'g2.d2.p.'a2.A2
    N = 'g.'p.N
```

The arbiter is slightly more complicated than the join. Users are competing for a resource which permits only one user at a time. The resource is guarded by an arbiter. A user initiates things by sending a request. When accepted by the arbiter, it sends out a grant. When the user is finished he sends a done signal, which is acknowledged by the arbiter. The typical user sequence is thus:

```
U1 = INITIAL.r1.g1.USE.f1.a1.FINAL
```

The arbiter may free the semaphore before sending an acknowledgement. It may then start serving another user whilst the first is awaiting the acknowledgement.

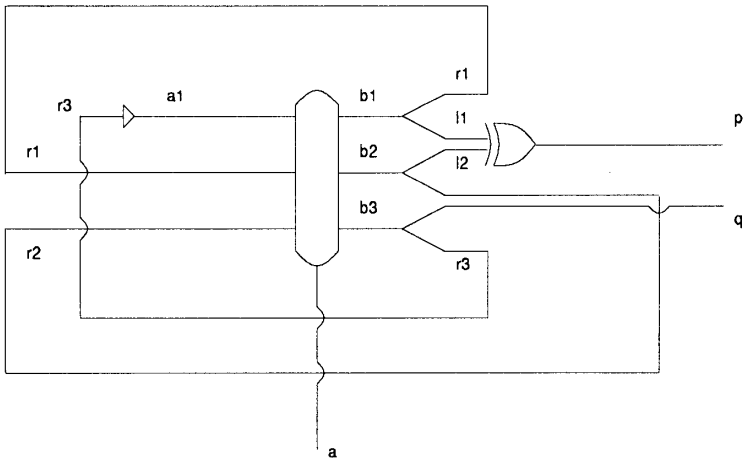
Example

Here is the specification of a modulo-3 counter and Jo Ebergen's one-hot implementation. The IWIRE makes sure that a1 fires first.

```
JOIN3   = ( A1 | A2 | A3 | N ) \{g, p}
where
   A1   = a1.g.'b1.'p.A1
   A2   = a2.g.'b2.'p.A2
   A3   = a3.g.'b3.'p.A3
    N   = n.'g.p.N

M3_spec = a.'p.a.'p.a.'q.M3_spec

M3_imp  = ( IWIRE  [ r3/a, a1/b ] \
          | JOIN3  [  a/n, r1/a2, r2/a3 ] \
          | FORK   [ b1/a, l1/b, r1/c ] \
          | FORK   [ b2/a, l2/b, r2/c ] \
          | FORK   [ b3/a,  q/b, r3/c ] \
          | MERGE2 [ l1/a, l2/b,  p/z ] \
          ) \{ a1, b1, b2, b3, l1, l2, r1, r2, r3}
```



Mod-3 Counter

1.1 Equivalence

Now that we have specified some basic circuits in CCS, we will want to see if our implementation meets the specification. We will use the Concurrency Workbench and the notion of equivalence in CCS for this purpose.

The Concurrency Workbench is an automatic tool for semantic analysis of CCS specifications.

CCS has three main notions of equivalence: strong equivalence (strong bisimulation), observation equivalence (weak bisimulation or, simply, bisimulation) and observation congruence (equality). Roughly, strong bisimulation treats the silent action τ (communication) as any other action; bisimulation ignores τ; equality ignores τ only under certain conditions. Please refer to [Mil89] for further details.

We will be using the workbench command "eq" (observation equivalence or bisimulation) for checking equivalence of circuit behaviour.

Let us see if two wires composed in parallel will behave as a single wire.

```
*Print the definition of a wire.
Command: pi
Identifier: WIRE

  WIRE = a.'b.WIRE

*Two wires in parallel, restricting the c-action which is internal.
Command: bi
Identifier: WIRE_2
Agent: (WIRE[c/b]|WIRE[c/a])\{c}

*Are they bisimilar?
Command: eq
Agent: WIRE
Agent: WIRE_2
false
```

They are not bisimilar because of the fact that the two-wire model acts as a buffer: it is possible for the two-wire system to accept another "a"

input before the "b" output has occured. The following workbench session will illustrate this.

```
* Visible (excluding internal tau actions) sequences of length 4.
Command: vs
Number: 4
Agent: WIRE
=== a 'b a 'b ===>

Command: vs
Number: 4
Agent: WIRE_2
=== a a 'b a ===>
=== a a 'b 'b ===>
=== a 'b a a ===>
=== a 'b a 'b ===>

Command: min
Agent: WIRE_2
Save result in identifier: WIRE_2'

WIRE_2' has 3 states.

Command: pi
Identifier: WIRE_2'

  WIRE_2' = WIRE_2'0  \
      where WIRE_2'0 = a.WIRE_2'2  \
        and WIRE_2'2 = a.WIRE_2'3 + 'b.WIRE_2'0  \
        and WIRE_2'3 = 'b.WIRE_2'2  \
      end
```

Therefore we have to enforce a condition on the two-wire system (and on most circuits, in fact) that when it is placed in an appropriate environment the behaviour will be as required. In this case, the environment is such that it will not supply an "a" input before it has seen a "'b".

This translates to the following with the appropriate renamings:

```
Command: bi
```

```
Identifier: ENV
Agent: x.'a.b.'y.ENV

Command: bi
Identifier: WSYS
Agent: (WIRE[c/b]|WIRE[c/a]|ENV)\{a,b,c}

Command: min
Agent: WSYS
Save result in identifier: WSYS'

WSYS' has 2 states.

Command: pi
Identifier: WSYS'

  WSYS' = WSYS'0  \
      where WSYS'0 = x.WSYS'4  \
        and WSYS'4 = 'y.WSYS'0  \
      end


Command: eq
Agent: WSYS'
Agent: (WIRE|ENV)\{a,b}
true
```

The above shows that the two-wire system and the single wire have the same behaviour (in the sense of bisimulation) if they are placed in an appropriate environment.

We have shown the appropriate environments for specific circuit behaviours for various simple asynchronous circuits.

Let us look at the Modulo-3 counter. We can just construct an environment (like in the previous example) by taking the "inverse" of the specification.

```
bi
ENV
```

```
x.'a.p.'y.x.'a.p.'y.x.'a.q.'z.ENV

Command: eq
Agent: (M3_spec|ENV)\{a,p,q}
Agent: (M3_imp|ENV)\{a,p,q}
true

Command: min

Agent: (M3_imp|ENV)\{a,p,q}
Save result in identifier: M3'

M3' has 6 states.

Command: pi
Identifier: M3'

  M3' = M3'0  \
      where M3'0 = x.M3'69  \
        and M3'57 = x.M3'58  \
        and M3'58 = 'z.M3'0  \
        and M3'69 = 'y.M3'70  \
        and M3'70 = x.M3'71  \
        and M3'71 = 'y.M3'57  \
      end
```

We can also construct a more generous environment which requires just that the environment wait for an output (of "p" or "q") before it sends an input.

```
Command: bi
Identifier: ENV'
Agent: x.'a.(p.'y.ENV' + q.'z.ENV')

Command: eq
Agent: (M3_spec|ENV')\{a,p,q}
Agent: (M3_imp|ENV')\{a,p,q}
true
Command: min
```

```
Agent: (M3_imp|ENV'')\{a,p,q}
Save result in identifier: M3''

M3'' has 6 states.

Command: pi
Identifier: M3''

  M3'' = M3''0  \
      where M3''0 = x.M3''69  \
        and M3''57 = x.M3''58  \
        and M3''58 = 'z.M3''0  \
        and M3''69 = 'y.M3''70  \
        and M3''70 = x.M3''71  \
        and M3''71 = 'y.M3''57  \
      end
```

The above environment may not work in general and may allow additional behaviours that may not be desirable.

Chapter 2

Hennessy-Milner Logic

2.1 An Introduction to Modal Logic

While CCS has different notions of equivalence of two agents, there is no notion of why two agents differ. Modal Logic is a powerful method for describing properties of agents, together with a definition of what each property means. Some examples of properties which we would like to prove of a system, might be:

- There is at most one user of an Arbiter at a time;
- It is not possible to input another value to a `WIRE` until after the `WIRE` has output;
- If a `WIRE` has an output, i.e. the environment must be willing to accept the output, the environment can not demand that the `WIRE` "buffer" the value.
- If a `FORK` has a value in it, the only reason that the value can not be output is because the environment is not ready.

CCS can only express equivalence of states; two states are equivalent if all of their properties are equivalent. If they differ, Modal Logic can tell us which properties are the same, in a given state, and which properties are different. The next sections will show:

- Modal Logic can provide a syntax to state these properties formally;

- Modal Logic has a semantics which gives meaning to these properties;
- examples to illustrate the points being made.

Following this, Hennessy-Milner Logic is introduced as a special type of a Modal Logic.

2.1.1 Definition of a Modal Logic

A modal logic, like all other logics, comprises:

- a set of formulae;
- a model over which the formulae are interpreted.

however, a modal logic is distinguished from other logics by two operators call possibily and necessity. Although some purely syntactic proofs may be performed over a logic, it is necessary to introduce a model to give a semantics or meaning to the logic.

2.1.2 Formulae of a Modal Logic

This section describes "formulae" of modal logic and describes the grammar which the logic obeys. It does not describe the "meaning" of these formulae; that is done in the next section.

The set of formulae of a modal logic are inductively defined as:

$$A ::= Q \mid \neg A \mid A \wedge A \mid \Diamond A \mid (A)$$

where:

- A is a formulae of modal logic;
- Q is one of a denumerable set of atomic propositions;
- $\neg$ is the negation of a formula;
- $\wedge$ is the conjunction of two formulae;

- $\Diamond$ is a modal operator called "possibility";
- () are parenthesis which give precedence to operations inside them.

The atomic formula T, or "true" is always a member of Q, and in Hennessy-Milner Logic is the only member of Q. These formulae are usually extended with the definition of the following derived operators:

- The atomic formula F is defined as $\neg$T.
- A$\vee$B is defined as $\neg(\neg A \wedge \neg B)$
- $\Box$A is defined as $\neg\Diamond\neg$A, and is called "necessity";

Informal examples of atomic propositions for the agent Wire, might be:

- *Hot*; this is true if the wire has a value which it could output.
- *Cold*; this is true if the wire is ready to input a value.

Some examples of formulae made up of atomic propositions might be:

- *Hot*
- $\neg$*Cold*
- $Hot \wedge \neg Cold \vee \neg Hot \wedge Cold$

These formulae are treated as formulae in the predicate calculus, and the operators have the same binding, in order of precedence the operators are $\neg$, $\wedge$, $\vee$.

The use of the modal operator $\Diamond$ is used to refer to what may "possibly" be true. Some examples of modal formulae might be:

- $\Diamond$*Hot*, which is read as "Possibly Hot";
- $\neg\Box$*Cold*, which is read as "Not Necessarily Cold";
- $(\Diamond Hot \wedge \neg Cold) \vee (\Diamond \neg Hot \wedge Cold)$, which is read as "Possibly Hot And Not Cold, Or Possibly Not Hot And Cold"

We will adopt the convention that the modal operators $\Diamond$ and $\Box$ have the same precedence and bind more strongly than $\wedge$, or $\vee$. Thus,

$$(\Diamond Hot \wedge \neg Cold) \vee (\Diamond \neg Hot \wedge Cold)$$
$$= ((\Diamond Hot)\ (\wedge(\neg Cold))) \vee ((\Diamond(\neg Hot) \wedge Cold))$$

This has a different meaning than,

$$(\Diamond(Hot \wedge \neg Cold)) \vee (\Diamond(\neg Hot \wedge Cold))$$

which would make more sense.

The next section discusses precisely what we mean by something being "possibly" true or "necessarily" true.

2.1.3 Models for a Modal Logic

The previous section described formulae for a modal logic, but it didn't give valuations or interpretations for them. The formula "Hot" is merely a symbol; conjunctions and negations of these symbols are merely expressions unless it is possible to assign values to these symbols, and deduction rules for computing values from expressions. Formulae are either "true" or "false" within a *model.* Models are useful because they have associated with them a valuation function, which maps from a formula to the set of all states where the formula is true. Thus, we can say a formula is true in some state if and only if that state is a member of the set to which the valuation function maps.

The Model for a modal logic which we shall use consists of:

- a set of states P;
- a relation between the states R, where R $\subseteq$ P $\times$ P; we call R the "next state" relation. It is possible for there to be more than one "next state".
- a mapping from atomic propositions Q, to subsets of states in P where for each $Q_i \in$Q, the mapping is described as $||Q_i||$. We say that an atomic proposition Q_i, is "true" in all states which are members of the set $||Q_i||$ and false otherwise.

This defines the valuation function for atomic propositions. The valuation function can be defined for the basic logic:

- $||T|| = P$;
- $||A \wedge B|| = ||A|| \cap ||B||$;
- $||\neg A|| = P - ||A||$
- $||\Diamond A|| = \{x \mid (x,a) \in R \text{ where } a \in ||A|| \}$

Thus,

- $||T||$ is the set of all states;
- $||A \wedge B||$ is the set of states where both A and B are true;
- $||\neg A||$ is the set of states where A is not true;
- $||\Diamond A||$ is the set of states which have a "next state" where A is true.

It is possible to extend the the valuation function to the derived operations. The reader should check that the valuation is consistent with the definition of these derived operations.

- $||F||$ is the empty set;
- $||A \vee B|| = ||A|| \cup ||B||$;
- $||\Box A||$ is the set of states where A is true in all the immediate successor states. It is vacuously true when the state has no successors.

In this interpretation the necessity or possibility of a proposition being true refers to the "next states" but not the current state. Other interpretations are possible (where the current state is included) but these will not be discussed. We write "proposition P is true at state S in Model M" as $\models^M_S P$. Thus, $\models^M_S P$, iff $S \in ||P||$. It is possible to drop the index "M" if there is no ambiguity about the model.

As an example, we can derive the following facts from these definitions:

- $||F|| = ||\neg T|| = P - ||T|| = P - P = \emptyset$;
  Therefore, False corresponds to the empty set.
- $||T \wedge A|| = ||T|| \cap ||A|| = P \cap ||A|| = ||A||$
  Since True corresponds to P, the set of all states, the set of states where the proposition (True$\wedge$A) is true is precisely the states where the proposition "A" is true.

- $||\Box F|| = ||\neg\Diamond\neg F|| = ||\neg\Diamond T||$
  $= P - ||\Diamond T||$
  $= P - \{x \mid (x,a) \in R \text{ where } a \in ||T|| \}$
  $= P - \{x \mid (x,a) \in R \text{ where } a \in P \}$
  $= P - \{x \mid x \text{ has a "next state"} \}$
  $= \{y \mid y \text{ does not have a "next state"} \}$
  Therefore "Necessarily False" corresponds to the set of all states which do not have successor states.

- $||\Diamond T|| = \{x \mid (x,a) \in R \text{ where } a \in ||T|| \}$
  $= \{x \mid (x,a) \in R \text{ where } a \in P \}$
  $= \{x \mid x \text{ has a "next state"} \}$
  Therefore "Possibly True" corresponds to the set of all states which have successor states. Necessity was defined as the dual of Possibility ($\Box A = \neg\Diamond\neg A$), and as a consequence of this, the current example is the dual of the previous example.

- $||\Diamond F|| = \{x \mid (x,a) \in R \text{ where } a \in ||F|| \}$
  $= \{x \mid (x,a) \in R \text{ where } a \in ||\neg T|| \}$
  $= \{x \mid (x,a) \in R \text{ where } a \in (P - ||T||) \}$
  $= \{x \mid (x,a) \in R \text{ where } a \in (P - P) \}$
  $= \{x \mid (x,a) \in R \text{ where } a \in (\emptyset) \}$
  $= \emptyset$
  $= ||F||$
  Therefore, "Possibly False" corresponds to the set of states which has a successor state which is in the empty set (as False corresponds to the empty set). Since there is no such state, it corresponds to the empty set, which corresponds to the predicate False.

- $||\Box T|| = ||\neg\Diamond\neg T||$
  $= P - ||\Diamond\neg T||$
  $= P - \{x \mid (x,a) \in R \text{ where } a \in ||\neg T|| \}$
  $= P - \{x \mid (x,a) \in R \text{ where } a \in (P - ||T||) \}$
  $= P - \{x \mid (x,a) \in R \text{ where } a \in (P - P) \}$
  $= P - \{x \mid (x,a) \in R \text{ where } a \in (\emptyset) \}$
  $= P - \emptyset$
  $= P$
  $= ||T||$
  Therefore, "Necessarily True" corresponds to the set of states which has all successor states which are in the set of all states (as True corresponds to the set of all states, P). Since this is true of all states, even those which have no successor states, it is the set of all states, P, which corresponds to the predicate True.

Again, this example and the previous example may be considered as duals of each other, as a consequence of the duality of the definition of Necessity and Possibility.

In summary,

- the model for a modal logic associates a set of states with each modal predicate;
- a predicate is true at a state if and only if the state is a member of this set of states associated with the predicate;
- Predicates are true in a model; it does not make sense to discuss truth without reference to a model.
- a predicate is "possibly" true in a state, if there is a "next state" where the predicate is true;
- a predicate is "necessarily" true in a state if the predicate is true in all next states. If there are no "next states", then the predicate is vacuously true.

2.2 Examples of Modal Predicates

Let us examine some simple modal predicates, and use as an example the Wire which is defined as:

```
bi Wire
Cold

bi Cold
in.Hot

bi Hot
'out.Cold
```

This CCS agent defines two states, Hot and Cold, and the relations between the two states. We arbitrarily define two predicates which corresponds to each state. Thus, a modal logic which corresponds to this system comprises:

- two states, Cold and Hot;
- a relation with two elements, {(Cold,Hot), (Hot,Cold)};
- two predicates, *Cold* and *Hot* where $||Cold|| = \{\text{Cold}\}$, and $||Hot|| = \{\text{Hot}\}$.

Note the different font used to distinguish the predicates *Cold* and *Hot* from the states Cold and Hot.

In this system, we examine the statements $\models_{Cold} \Diamond Hot$ and $\models_{Hot} \Box Hot$

Example: $\models_{Cold} \Diamond Hot$

This statement says that in the state Cold, the predicate possibly *Hot* is true. Remember that this means possibly hot in the next state.

- $||\Diamond Hot||$ is the set of states which have a successor state which is a member of $||Hot||$.
- But, $||Hot||$ is equal to {Hot}, and the set of states which have a successor state to {Hot} is {Cold}.
- $\models_{Cold} \Diamond Hot$ iff Cold $\in$ {Cold}.
- Thus, $\models_{Cold} \Diamond Hot$, or the state Cold satisfies the modal proposition $\Diamond Hot$, and therefore the proposition $\models_{Cold} \Diamond Hot$ is true.

Example: $\models_{Hot}\Box Hot$

This statement says that in the state Hot, it is necessary that the predicate *Hot* be true *in all next states.* The predicate says nothing about the current state.

- $\models_{Hot}\Box Hot$ is true iff Hot $\in ||\Box Hot||$.
- But, $||\Box Hot||$ is equal to the set of states which have all successor states a member of *Hot.*
- This is equal to the set {Cold}.
- Since Hot $\notin$ {Cold}, the state Hot does not satisfy $\Box Hot$ and the proposition $\models_{Hot}\Box Hot$ is False.

2.2.1 Wire with an Environment

Let us define an environment for this wire,

```
bi ENV
'in.E1 + a.Thinking

Thinking
'b.ENV

bi E1
out.ENV

bi System
(Wire | ENV) \{in,out}
```

The CCS agent System is a more complex agent than Wire as it has more states and more relations. We arbitrarily define predicates corresponding to this agent as well. Thus, a modal logic which corresponds to this system comprises:

- three states which come from the | composition,
  - (Cold|ENV);
  - (Cold|Thinking);
  - (Hot|E1).

These states are really (Cold|ENV) \{in, out}, etc. but for brevity, the restrictions of {in, out} will be assumed.

- a relation with four elements,
  - ((Cold|ENV), (Cold|Thinking));
  - ((Cold|Thinking), (Cold|ENV));
  - ((Cold|ENV), (Hot|E1))
  - ((Hot|E1), (Cold|ENV))
- two predicates,
  - *Cold* where $||Cold|| = \{(\text{Cold}|*)\}$ or {(Cold|ENV), (Cold|Thinking)};
  - *Hot*, where $||Hot|| = \{(\text{Hot}|\text{E1})\}$

In this system, we examine the following predicates:

- $\models_{(Cold|ENV)} \Diamond Hot$ and
- $\models_{(Hot|E1)} \Diamond Cold$.

Example: $\models_{(Cold|ENV)} \Diamond Hot$

- $||\Diamond Hot||$ is the set of states which have a successor state which is a member of $||Hot||$.
- But, in this system, $||Hot||$ is equal to {(Hot |E1)}.
- The set of states which have a successor state to {(Hot |E1)} is {(Cold |ENV)}.
- $\models_{(Cold|ENV)} \Diamond Hot$ iff (Cold|ENV) $\in$ {(Cold|ENV)}.
- Thus, $\models_{(Cold|ENV)} \Diamond Hot$, or the state (Cold|ENV) satisfies the modal proposition $\Diamond Hot$.

Example: $\models_{(Hot|E1)} \Diamond Cold$

- The set of states where $\Diamond Cold$ is true is $||\Diamond Cold||$.
- This is equal to the set of states which have a successor state where *Cold* is true.
- Since $||Cold||$ is equal to {(Cold|ENV), (Cold|Thinking)}, we find that $||\Diamond Cold||$ is equal to {(Cold|ENV), (Cold|Thinking), (Hot|E1)} which is equal to ||T||.
- Since (Hot|E1) is a member of this set, we conclude that (Hot|E1) satisfies $\Diamond Cold$ and the predicate $\models_{(Hot|E1)} \Diamond Cold$ is true.

2.3 Hennessy-Milner Logic

The previous section on Modal Logic took no account of labels on relations between states. Hennessy-Milner Logic is a modal logic which uses labelled transition systems as a model. Recommended reading for this section and the next chapter includes [Sti91]. HML differs from the modal logic introduced in the earlier section in the following areas:

- HML has only one predefined atomic formula, T;
- HML uses labels on transitions to index the modal operators Box and Diamond;
- HML has some additional notation to deal with sets of labels on transitions;
- HML has a notion called "weak necessity" and "weak possibility" for dealing with composed relations;
- A notion of "fixpoint" has been borrowed from the Modal μ-calculus.

Only the first two points will be dealt with in this section leaving the other points for future sections.

Labeled Modalities

Although HML may seem to be fatally weakened by having "T" as the only atomic predicate, the use of labels on modalities regains most of the lost power.

The indexing, written as $\langle b \rangle$ where "b" is a label and $||\langle b \rangle A||$ is defined as: $\{x \mid (x,a) \in R$ where $a \in ||A||$ and (x,a) bears label b $\}$.
Thus, $||\langle a \rangle B||$ is the set of states which are related to a state in $||B||$ by an "a" transition.

Likewise, [a]B is defined as $\neg\langle a \rangle \neg B$. Similarly, $||[a]B||$ is the set of states which do not have an "a" transition to a state which is not a member of $||B||$.

Some examples of this logic are:

- Wire$\models\langle in \rangle T$;
  Agent "Wire" has the ability to perform an "in" action.

- (Cold|ENV)$\models\langle$in$\rangle$T$\wedge$['out]F;
  Agent "(Cold|ENV)" has the ability to perform an "in" action and can not perform an "'out" action.

- (Cold|Thinking)$\models$[in]F;
  Agent "(Cold|Thinking)" does not have the ability to perform an "in" action.

2.3.1 HML Satisfaction

In the previous section we gave a semantics for Modal Logic based on a model for Modal Logic and gave a denotation for "T", "$\wedge$" and "$\neg$". HML inductively defines a "satisfaction" relation which is entirely consistent with the model in the previous section. It should be noted that the "satisfaction relation" defines a set of "rules of deduction", yet they use the "Model" symbol $\models$ instead of the "Deduction" symbol $\vdash$. We state these rules of deduction, and omit the proof that these rules are sound with respect to the model.

- A$\models$T;

- A$\models\neg$P iff A $\not\models$ P;

- A$\models$P1$\wedge$P2 iff A$\models$P1 and A$\models$P2;

- A$\models\langle$x$\rangle$P iff for some A', A $\xrightarrow{x}$ A' and A'$\models$P.

It is possible to prove that these deduction rules are sound, and consistent with respect to the semantics of the model defined in the previous section. Instead of giving a proof, the next example will be worked with both the deduction rules and the model.

Example 1. Given that:

$$\begin{array}{lcll} X & = & a.(b.nil + c.nil) & (a) \\ Y & = & a.b.nil + a.c.nil & (b) \\ E & = & \langle a\rangle\,(\langle b\rangle T \;\&\; \langle c\rangle T) & \end{array}$$

show that X $\models$ E and Y $\not\models$ E where E expresses the condition that after an a-action it is possible to do a b-action and it is possible to do a c-action.

Proof by Model

Show that $\models_X$ E. We know that this is true if X $\in$||E||. We must find ||E||. First, define the model.

- Let the set of states be {X, X1, X2, X3}.
- Let the relations R be {(X, a, X1), (X1, b, X2), (X1, c, X3)}.
- The only predicate is the constant T where ||T|| = {X, X1, X2, X3}.

We show that $\models_X$ E as follows:

- ||E|| = ||⟨a⟩(⟨b⟩T∧⟨c⟩T)||;
- ||⟨a⟩(⟨b⟩T∧⟨c⟩T)|| = {X |(X,a,Y) ∈R and Y ∈||(⟨b⟩T∧⟨c⟩T)|| };
- ||(⟨b⟩T∧⟨c⟩T)|| = ||(⟨b⟩T||∩||⟨c⟩T)||;
- ||(⟨b⟩T|| = {y |(Y,b,Y1) and Y1 ∈T }; but this equals {X1};
- ||⟨c⟩T)|| = {y |(Z,c,Z1) and Z1 ∈T }; but this equals {X1};
- therefore, ||(⟨b⟩T||∩||⟨c⟩T)|| = {X1} ∩ {X1} = {X1};
- therefore, ||(⟨b⟩T∧⟨c⟩T)|| = {X1};
- therefore {X |(X,a,Y) ∈R and Y ∈||(⟨b⟩T∧⟨c⟩T)|| }
  = {X |(X,a,Y) ∈R and Y ∈{X1} } = {X}.
- Since X ∈{X} we conclude that $\models_X$ E.

Disproof by Model

Show that $\not\models_Y$ E. We know that this is true if Y$\notin$||E||. We must find ||E||. First, define the model.

- Let the set of states be {Y, Y1, Y2, Y3, Y4 }.
- Let the relations R be
  {(Y, a, Y1), (Y1, b, Y2), (Y, a, Y3),
  (Y3, c, Y4)}.
- The only predicate is the constant T where
  ||T|| = {Y, Y1, Y2, Y3, Y4}.

We show that $\not\models_y$ E as follows:

- $||E|| = ||\langle a\rangle(\langle b\rangle T \wedge \langle c\rangle T)||$;
- $||\langle a\rangle(\langle b\rangle T \wedge \langle c\rangle T)|| = \{X \mid (X,a,Y) \in R \text{ and } Y \in ||(\langle b\rangle T \wedge \langle c\rangle T)|| \}$;
- $||(\langle b\rangle T \wedge \langle c\rangle T)|| = ||(\langle b\rangle T|| \cap ||\langle c\rangle T)||$;
- $||(\langle b\rangle T|| = \{y \mid (Y,b,Y1) \text{ and } Y1 \in T \}$; but this equals $\{Y1\}$;
- $||\langle c\rangle T)|| = \{y \mid (Z,c,Z1) \text{ and } Z1 \in T \}$; but this equals $\{Y3\}$;
- therefore, $||(\langle b\rangle T|| \cap ||\langle c\rangle T)|| = \{Y1\} \cap \{Y3\} = \{\} = \emptyset$;
- therefore, $||(\langle b\rangle T \wedge \langle c\rangle T)|| = \{\} = \emptyset$;
- therefore $\{X \mid (X,a,Y) \in R \text{ and } Y \in ||(\langle b\rangle T \wedge \langle c\rangle T)|| \}$ $= \{X \mid (X,a,Y) \in R \text{ and } Y \in \emptyset \} = \emptyset$.
- Since $X \notin \emptyset$ we conclude that $\not\models_X$ E.

Next, we do the same proof using the deduction rules rather than the model. These examples with a little thought should convince the reader that the model and the deduction rules will always produce the same result.

2.3.2 Proof by Deduction Rules

Example 1. Given that

$$\begin{array}{lcll} X & = & a.(b.nil + c.nil) & (a) \\ Y & = & a.b.nil + a.c.nil & (b) \\ E & = & \langle a\rangle(\langle b\rangle T \;\&\; \langle c\rangle T) & \end{array}$$

show that $X \models E$ and $Y \not\models E$ where E expresses the condition that after an a-action it is possible to do a b-action and it is possible to do a c-action.

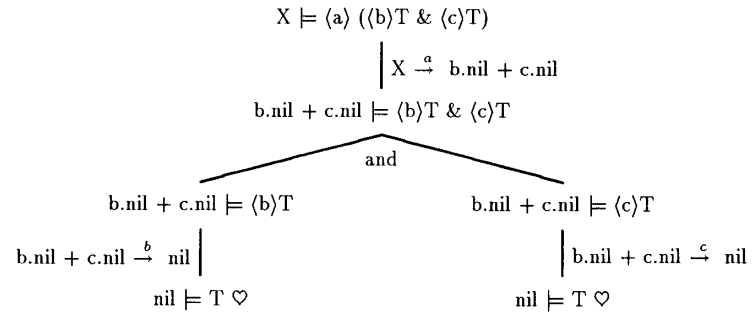


Figure 2.1 Proof of X ⊨ ⟨a⟩ (⟨b⟩T & ⟨c⟩T)

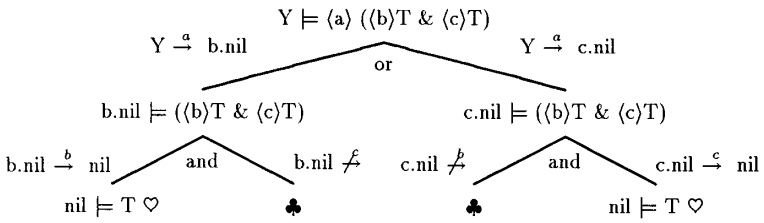


Figure 2.2 Proof of Y⊭⟨a⟩(⟨b⟩T&⟨c⟩T)

Example 3. Given that

X	=	a.(b.nil + c.nil)	(a)
Y	=	a.b.nil + a.c.nil	(b)
E	=	⟨a⟩ (⟨b⟩T & ⟨c⟩T)	

Find a formula G not containing c such that X ⊨ G and Y ⊭ G.

The condition we seek to express is "after an a it is always possible to get a b" which comes out as G = [a]⟨b⟩T.

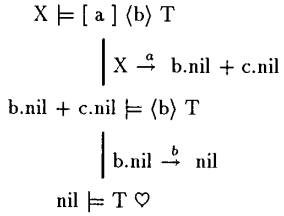


Figure 2.3 Proof of X ⊨ [a] ⟨b⟩ T

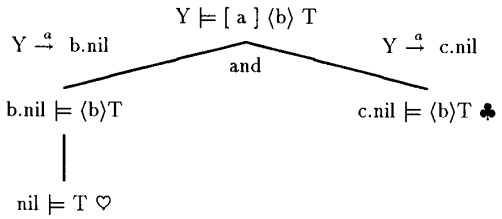


Figure 2.4 Proof of Y ⊭ [a] ⟨b⟩ T

2.4 Multiple actions

It is convenient to add some more notation dealing with "sets" of actions as it makes predicates and proofs more concise. For example, if we wished to state that actions $a_1, a_2, ...a_n$ were possible after which predicate "A" held, using the current notation we would write:

$$\langle a_1\rangle A \vee \langle a_2\rangle A \vee ... \langle a_n\rangle A$$

A more convenient notation for the same predicate would be:

$$\langle a_1, a_2 ... a_n\rangle A$$

The set notation is extended as follows:

- Rather than explicitly spell out the set of actions each time they are referenced, it is possible to define a set and use the set reference. For example, if the set $\mathcal{K}$ is defined as $\{a_1, a_2, ...a_n\}$ then $\langle\mathcal{K}\rangle$ would mean $\langle a_1, a_2 ... a_n\rangle$.
- Rather than define sets explicitly each time they are used, it is possible to use "set arithmetic" to define sets. For example, the set $\mathcal{X} - \mathcal{K}$ is the set of elements of $\mathcal{X}$ which are not in the set $\mathcal{K}$. If $\mathcal{X}$ is the set $\{b_1, b_2, a_1, a_2\}$, and $\mathcal{K}$ is the set $\{a_1, a_2, ...a_n\}$, then the set expression $\mathcal{X}$ - $\mathcal{K}$ is equal to the set $\{b_1, b_2\}$. Similarly, the predicate $\langle\mathcal{X} - \mathcal{K}\rangle$A is equal to $\langle b_1, b_2\rangle$A which is equal to $\langle b_1\rangle$A$\vee\langle b_2\rangle$A.
- Rather than explicitly state the complement of a set, it is possible to omit the set of all actions. For example, if $\mathcal{A}$ is the set of all actions, and $\mathcal{K}$ is as before, then -$\mathcal{K}$ is equal to $\mathcal{A}$ - $\mathcal{K}$. If the set of all actions, $\mathcal{A}$ is $\{a_1, a_2, ...a_n, b_1, b_2, ...b_m\}$ then -$\mathcal{K}$ is the set $\{b_1, b_2, ...b_m\}$.
- Rather than explicitly stating the set of all actions, it is possible to use "-". For example, to the predicate ⟨-⟩T says that some action is possible, and [-]F says that no action is possible.

We have defined set operations for the possibility operator but have not yet done so for the necessity operator. Recall that when dealing with unlabelled relations, possibility and necessity were defined as duals, where □A is defined as ¬◇¬A. This notion of duality was extended to labelled relations where [a]A is defined as ¬⟨a⟩¬A. Since we wish to retain this notion of duality for sets of actions, we define [K]A is defined as ¬⟨K⟩¬A.

- but $\neg\langle \mathrm{K}\rangle\neg \mathrm{A}$ is equal to $\neg\langle a_1, a_2, ...a_n\rangle\neg \mathrm{A}$
- which equals $\neg(\langle a_1\rangle\neg \mathrm{A} \vee \langle a_2\rangle\neg \mathrm{A}$ ... $\vee\langle a_n\rangle\neg \mathrm{A})$
- which equals $\neg\langle a_1\rangle\neg \mathrm{A} \wedge \neg\langle a_2\rangle\neg \mathrm{A}$ ... $\wedge\neg\langle a_n\rangle\neg \mathrm{A}$
- which equals $[a_1]$ A$\wedge[a_2]$ A ... $\wedge[a_n]$ A
- Hence, [K]A = $[a_1, a_2, ...a_n]$A = $[a_1]$A$\wedge[a_2]$A ... $\wedge[a_n]$A

2.4.1 Useful Facts

Here are some common formulae and their interpretations:

P$\models$[a]T	always true
P$\models$[a]F	P cannot do an a action
P$\models\langle$a$\rangle$T	it is possible for P to do an a action
P$\models\langle$a$\rangle$F	always false
P$\models$[a]A	is true for any A if P cannot perform an a action
P$\models$[-]F	if P cannot do any action
P$\models\langle$-$\rangle$T$\wedge$[-a]F	P can do an a action and nothing else

2.5 Examples

In this section, some useful properties of the C-Element are checked using HML on the CWB.

The Workbench has a model checker for the purpose of checking propositions about a particular CCS agent. The command is "cp" (check proposition). The reader is reminded that the workbench uses the set notation defined earlier, and also writes $\wedge$ as "&" and $\vee$ as "+".

```
* Print the definition of CEL.
Command: pi
Identifier: CEL

  CEL = a.b.'c.CEL + b.a.'c.CEL

* It is possible to do an "a" action.
Command: cp
Agent: CEL
Proposition: <a>T
true

* It is possible to do an "a" or a "b".
Command: cp
Agent: CEL
Proposition: <a,b>T
true

* After an "a" action it is not possible to do another "a" but
* it is possible to do a "b".
Command: cp
Agent: CEL
Proposition: [a]([a]F & <b>T)
true

* It is not possible to do two "b"'s in a row.
Command: cp
Agent: CEL
Proposition: <b><b>T
false

* An "a" may not be possible after "[a,b]".
* "[a,b]<a>T" expands to "[a]<a>T & [b]<a>T"
Command: cp
Agent: CEL
Proposition: [a,b]<a>T
false

* After a "b" some action is possible and this action cannot
* be anything other than an "a".
Command: cp
Agent: CEL
```

```
Proposition: [b](<->T & [-a]F)
true

* The third action must be a "'c".
Command: cp
Agent: CEL
Proposition: <->T & [-](<->T & [-](<->T & [-'c]F))
true
```

The truth values of expressions with multiple actions are not always intuitively obvious.

2.6 Two Modal Logics for CCS

There are two modal logics for CCS associated with two transitions defined in conjunction with the two bisimulation equivalences (weak and strong). If τ is observable, the associated transition relation is $\longrightarrow$ and the modalities are as mentioned before. If τ is not observable, then the transition relation is $\Longrightarrow$ and the associated modalities are expressed as [[K]] and $\langle\langle K \rangle\rangle$.

The τ action can occur only with strong modalities and the corresponding transition associated with weak bisimulation is $\stackrel{\varepsilon}{\Longrightarrow}$ and the modalities are $[[\varepsilon]]$ and $\langle\langle\varepsilon\rangle\rangle$ where the ε denotes zero or more τ transitions.

A small example will serve to illustrate the fact that one has to remember the definition of ε when it is used in conjunction with sets in weak modalities on the workbench.

```
Command: pi
Identifier: WIRE

  WIRE = a.'b.WIRE

* Is it possible to do something other than an "a"?
Command: cp
Agent: WIRE
Proposition: <-a>T
false

* With weak modalities.
Command: cp
Agent: WIRE
Proposition: <<-a>>T
true

* The previous proposition was true because it is possible to do
* an epsilon transition which is zero or more tau actions.
Command: cp
Agent: WIRE
Proposition: <<-a,eps>>T
false
```

2.7 Examples

To show some aspects of difference in behavior between WIRE and two-wires composed together:

```
* Print the definition of WIRE.
Command: pi
Identifier: WIRE

  WIRE = a.'b.WIRE

* Bind WIRE_2 to the composition of two wires.
Command: bi
Identifier: WIRE_2
Agent: (WIRE[c/b]|WIRE[c/a])\{c}

* It is possible to do two "a"s in a row.
Command: cp
Agent: WIRE
Proposition: <<a>><<a>>T
false

* It is possible to do two "a"s in a row.
Command: cp
Agent: WIRE_2
Proposition: <<a>><<a>>T
true

* After an "a" action, it is possible to do a "'b".
Command: cp
Agent: WIRE
Proposition: [a](<'b>T)
true

* After an "a" action, it is possible to do a "'b"
* (using weak modalities, ignoring tau).
Command: cp
Agent: WIRE
Proposition: [[a]](<<'b>>T)
true

* After an "a" action, it is possible to do a "'b"
* (using weak modalities, ignoring tau).
Command: cp
Agent: WIRE_2
Proposition: [[a]](<<'b>>T)
true

* Minimize WIRE_2 thereby getting rid of tau.
Command: min
Agent: WIRE_2
Save result in identifier: WIRE_2'
```

```
WIRE_2' has 3 states.

* Print the minimized agent WIRE_2'.
Command: pi
Identifier: WIRE_2'

  WIRE_2' = WIRE_2'0  \
      where WIRE_2'0 = a.WIRE_2'2  \
        and WIRE_2'2 = a.WIRE_2'3 + 'b.WIRE_2'0  \
        and WIRE_2'3 = 'b.WIRE_2'2  \
      end

* Strong modalities can be used now since the tau's are gone.
Command: cp
Agent: WIRE_2'
Proposition: [a](<'b>T)
true
```

2.8 Modal Characterization Theorems

We know that if two agents are bi-similar they are denotatively congruent; one may be substituted for another in any context. It follows that since there is no test which can be performed to distinguish the two, that all of the modal properties must likewise be equivalent. For, if there were some modal predicate which could distinguish them it would not be difficult to build a context in which the agents differed. Similarly, if all of the modal properties of two agents are identical, the two agents are bi-similar.

We state without proof, that two agents are strongly bi-similar if and only if there is no modal proposition which distinguishes them using the strong Hennessy-Milner Logic.

Similarly, two agents are weakly bi-similar if and only if there is no modal proposition which distinguishes them using the weak Hennessy-Milner Logic.

Chapter 3

Modal μ-calculus

3.1 Motivation

In CCS it is possible to define "recursive" agents such as Wire.

```
bi Wire
a.'b.Wire
```

This agent can satisfy the HML modal predicate ⟨a⟩⟨'b⟩T. It can also satisfy the predicates ⟨a⟩⟨'b⟩⟨a⟩⟨'b⟩T and ⟨a⟩⟨'b⟩⟨a⟩⟨'b⟩⟨a⟩⟨'b⟩⟨a⟩⟨'b⟩...

Since CCS has a method to express recursive agents, it is desirable to have a method of expressing recursive properties properties in Modal Logic. We could express the equation that wire satisfies as:
Y = ⟨a⟩⟨'b⟩Y
To see that this works as advertised, read the equation as "Y is the modal property that it is possible to do an "a" action, then possible to do a "'b" action and arrive in a state where property Y holds".

We call such equations which refer to themselves in their definition *fixpoint* equations. Their general form is Y = F(Y) where F is some function or relation. Solutions to fixpoints equations need not be unique nor even exist. For example, the equation X = ¬X does not have a solution. However, if the function F has the special property that

if $X \subseteq Y$, then $F(X) \subseteq F(Y)$

we call F monotonic, and solutions to the fixpoint equations are guaranteed to exist. Syntactically, this corresponds the restriction that there must be an even number of negations prefixing the reference to the variable Y in the function F. This is not the case in the equation $Y = \neg Y$.

The rest of this chapter discusses these fixpoint equations and their solutions.

3.2 Fixpoints in the Model

It is possible to describe a fixpoint in terms of the deduction rules or in terms of the model. This section describes the fixpoint's characteristics in terms of the model.

3.2.1 Non Uniqueness of Fixpoints

Solutions to fixpoint equations need not be unique. This is illustrated in this section by finding several solutions to a fixpoint equation for a given system.

As an example, consider the following contrived system:

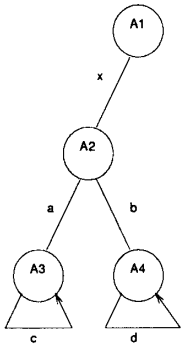
```
bi A1
x.A2

bi A2
a.A3 + b.A4

bi A3
c.A3

bi A4
d.A4
```

The following diagram represents the same agent.



The modal logic which corresponds to this is:

- States = {A1, A2, A3, A4};

- Relations R = {(A1, x, A2), (A2, a, A3), (A2, b, A4), (A3, c, A3), (A4, d, A4)};
- Predicates
  - $||T|| = \{A1, A2, A3, A4\}$;
  - $||F|| = \{\}$;
  - $||A1|| = \{A1\}$;
  - $||A2|| = \{A2\}$;
  - $||A3|| = \{A3\}$;
  - $||A4|| = \{A4\}$.

For convenience, four predicates corresponding to the four states have been defined. As in the previous chapter, the predicate *A1* has a different font than the state "A1".

Example: A1 is a fixpoint

To go along with the contrived system, a contrived fixpoint equation is introduced:

$$Y = ([-]Y \vee \langle x \rangle T)$$

It would be helpful if there were some intuitive meaning which could be associated with this, but as will be shown in future sections there is more than one meaning associated with this equation and this system. It suffices that it is a fixpoint equation; this section checks whether $||Y|| = \{A1\}$ is a solution. This is done by using the methods of the previous chapter, and showing that $||[-]A1 \vee \langle x \rangle T||$ is equal to $||A1||$ where *A1* is the proposition which is true only of state "A1". As was done in the previous chapter:

- $||[-]A1 \vee \langle x \rangle T|| = ||[-]A1|| \cup ||\langle x \rangle T||$
- $||[-]A1|| = \{X \mid (X, -, A1) \in R\} = \{\} = \emptyset$
- $||\langle x \rangle T|| = \{Y \mid (Y, x, Z) \in R \text{ for any } Z\} = \{A1\}$
- therefore, $||[-]A1|| \cup ||\langle x \rangle T|| = \emptyset \cup \{A1\} = \{A1\}$
- therefore it is true that $Y = \{A1\}$ is a solution to the fixpoint equation $Y = ([-]Y \vee \langle x \rangle T)$.

Example: {A1,A3} is a fixpoint

Using the methods of the previous chapter, check whether {A1,A3} is a solution to the fixpoint equation Y = ([-]Y∨⟨x⟩T). For brevity, we will refer to the proposition which corresponds to the set {A1,A3} as S2 in this example.

- ||([-]S2∨⟨x⟩T)|| = ||[-]S2)|| ∪ ||⟨x⟩T||
- ||⟨x⟩T|| = {A1}
- ||[-]S2)|| = {X |(X, -, R) implies R ∈S2 where S2 = {A1,A3} } = {A3} Please note that A2 is not included; it has a derivative (A4) which is not in the set {A1, A3}
- ||([-]S2∨⟨x⟩T)|| = {A3}∪{A1} = {A1,A3}

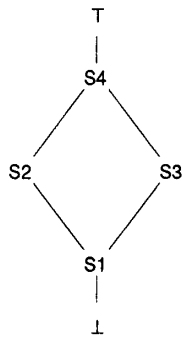
and therefore *S2*, where ||S2|| = {A1,A3} is also a solution to the fixpoint equation.

List of Fixpoints

A list of other solutions follows, and it is suggested that the reader check that they do satisfy the equation.

- S1 = {A1}
- S2 = {A1,A3}
- S3 = {A1,A4}
- S4 = {A1,A2,A3,A4}

It will be shown that the following diagram forms a lattice. Since lattices correspond to a partial order, we note that elements which are greater in the lattice are supersets.



3.2.2 Example of a Non Fixpoint

It will be shown that the set Z = {A1, A3, A4} is not a fixed point of the equation F(Y) = ([-]Y∨⟨x⟩T) with the system in the previous section.

- ||([-]Z∨⟨x⟩T)|| = ||[-]Z)|| ∪ ||⟨x⟩T||
- ||⟨x⟩T|| = {A1}
- ||[-]Z)|| = {X |(X, -, R) implies R ∈Z where Z = {A1,A3,A4} } = {A2,A3,A4}
- ||([-]Z∨⟨x⟩T)|| = {A2,A3,A4} ∪ {A1} = {A1,A2,A3,A4}

and therefore solution ||Z|| = {A1,A3,A4} is not a solution to the equation. When we assumed that F(Y) = ([-]Y∨⟨x⟩T) is true at states {A1, A3, A4}, a logical consequence is that it is also true at A2. This is reasonable, as states are included if all of their derivatives are included, and if we assume that A3 and A4 are included, then A2 must also be included.

Fixpoint as Deductive Closure

A solution to a fixed point equation can be thought of as a truth assignment to a set of states which is deductively closed under the formula. Thus, if we assume that the fixed point formula F(y) is true for every y in fixed point set X, then testing this assumption by applying F(y) to each member of X will not only confirm the assumption by asserting that F(y) is true, but it

will also assert that it is not possible to show that it is true for any state except those in X.

In this sense X is a "consistent set of beliefs" given F(y) as a deduction rule.

3.2.3 Unions of Fixpoints and the Maximum Fixpoint

In the previous section three examples were worked with the same fixpoint equation, and it was shown that:

- {A1, A3} is a fixpoint;
- {A1, A4} is a fixpoint;
- {A1, A3, A4} is not a fixpoint;

Thus, it is not necessarily true that the union of any two solutions to a fixpoint equation is a solution to a fixpoint equation. However, it is true that the union of any two solutions may be extended to a solution. Further, for fixed points A and B, there is a unique solution X which has the following properties;

- X is a super set of A, and X is a super set of B;
- for any Y which is a solution to the fixpoint equation, such that Y is a super set of A and B, either X = Y or X is a subset of Y.

This can be seen from the example cited above; {A1, A2, A3, A4} is a super set of {A1, A3} and of {A1, A4} and it is the least such super set that is also a solution to the fixpoint equation. This solution X may be the union of A and B, however if it is not, X is the deductive closure of the union of A and B. We say that Y is a prefixed point of F if $F(Y) \subseteq Y$.

To see that this is the case, consider:

- If F is monotonic, then $||F(A \cup B)|| \supseteq ||F(A)||$ and $||F(A \cup B)|| \supseteq ||F(B)||$
- Therefore, $||F(A \cup B)|| \supseteq ||F(A)|| \cup ||F(B)||$ but since A and B are fixpoints, $||F(A)|| \cup ||F(B)|| = A \cup B$;

- but by monotonicity, $\|F(F((A \cup B))\| \supseteq \|F(A \cup B)\|$;
- and therefore by induction, $\|F^{n+1}(A \cup B)\| \supseteq \|F^n(A \cup B)\| \supseteq A \cup B$;
- this process must come to an end since, $P \supseteq F^n(A \cup B)$
- and therefore, there is a unique fixpoint which contains $A \cup B$

It has been shown that if there are solutions to the fixpoint equation, then there is a greatest solution, since:

- the union of all prefixed points can be extended to a solution;
- it is greater than any other solution;
- it is unique.

The maximum fixpoint corresponds to the deductive closure of the union of all fixpoints. It includes everything except that which can not possibly be a member of the fixpoint.

3.2.4 Intersections of Fixpoints and the Minimum Fixpoint

To show that the intersection of two solutions is not always a solution consider the previous system with the following contrived fixpoint equation:

$$Y = (\langle - \rangle Y \vee \langle x \rangle T)$$

It has the following solutions:

- S1 = {A1}
- S2 = {A1,A2,A3}
- S3 = {A1,A2,A4}
- S4 = {A1,A2,A3,A4}

The solutions are not closed under intersection. To see that solutions are not closed under intersection, consider $S2 \cap S3$ which is {A1, A2}. This is not a solution as for $\|Z\|$ = {A1, A2} :

- $||(\langle\text{-}\rangle Z \vee \langle x \rangle T)|| = ||\langle\text{-}\rangle Z|| \cup ||\langle x \rangle T||$
- $||\langle x \rangle T|| = \{A1\}$
- $||\langle\text{-}\rangle Z|| = \{X \mid (X, \text{-}, R) \text{ where } R \in Z \text{ where } Z = \{A1,A2\}\} = \{A1\}$
- $||(\langle\text{-}\rangle Z \vee \langle x \rangle T)|| = \{A1\} \cup \{A1\} = \{A1\}$

and therefore $||Z|| = \{A1,A2\}$ is not a solution to the equation. This is reasonable, as A2 does not have any derivatives which are a member of the set {A1, A2}. Thus, the greatest solution set which is a subset of {A1, A2} is {A1}.

Dual to the case of unions, the intersection of any two solutions is a superset of a unique solution. We say that Y is a postfixed point of F if $Y \subseteq F(Y)$.

To see that this is the case, consider:

- If F is monotonic, then $||F(A \cap B)|| \subseteq ||F(A)||$ and $||F(A \cap B)|| \subseteq ||F(B)||$
- Therefore, $||F(A \cap B)|| \subseteq ||F(A)|| \cap ||F(B)||$ but since A and B are fixpoints, $||F(A)|| \cap ||F(B)|| = A \cap B$;
- but by monotonicity, $||F(F((A \cap B))|| \subseteq ||F(A \cap B)||$;
- and therefore by induction, $||F^{n+1}(A \cap B)|| \subseteq ||F^n(A \cap B)|| \subseteq A \cap B$;
- this process must come to an end since, $\emptyset \subseteq F^n(A \cap B)$
- and therefore, there is a unique largest fixpoint which is contained in $A \cap B$

It has been shown that if there are solutions to the fixpoint equation, then there is a least solution, since:

- the intersection of all postfixed points contain a solution;
- it is less than any other solution;
- it is unique.

This solution is called the minimum fixed point. Frequently, the minimum fixed point is $\emptyset$. None of the contrived fixpoint equations have allowed for this possibility, as an empty set as a solution may be confusing.

3.2.5 Some Intuition

In the example with the unions of fixpoints, the deductive closure kept getting bigger, while in the example with the intersection of fixpoints the deductive closure kept getting smaller. In the example with the unions, we could consider {A3,A4} as being facts, while {A2} is a consequence of these facts. In the example with the intersections, {A2} is a consequence of one of the facts {A3} or {A4}. After the intersection, the facts were eliminated but the conclusion remained. After iterating a few times the consequences of the facts were eliminated. Similarly, when the union of two fixpoints was taken, facts were added to the system and by taking the deductive closure consequences were added.

We can give an interpretation to the fixpoint equation,

$$Y = (\langle x \rangle T \vee [-]Y)$$

which we introduced earlier. Under the minimum interpretation, this corresponds to the property that it is possible to perform an **x** action, or every action will inevitably lead to the possibility of an **x** action. This can be seen by expanding the equation by substituting the right side of the equation for Y:

$$\langle x \rangle T \vee [-](\langle x \rangle T \vee [-]Y)$$

Expanding twice yields:

$$\langle x \rangle T \vee [-](\langle x \rangle T \vee [-](\langle x \rangle T \vee [-]Y))$$

Similarly, under the minimal interpretation, the equation:

$$Y = (\langle x \rangle T \vee \langle - \rangle Y)$$

corresponds to the property that it is possible to perform an **x** action, or there is a derivative (or derivative of a derivative) which can perform an **x** action. Under the maximal interpretation, the first fixpoint equation denotes the the set of all agents. The second denotes the set of agents which can eventually perform an **x** action, or which can perform some infinite sequence of actions.

3.2.6 Duality of Maximum and Minimum Fixpoints

There is a simple way in which the maximum and minimum fixpoints are duals of each other.

> The maximum includes everything except that which can not possibly be true, while the minimum includes only that which must necessarily be true.

This can be expressed in the following famous theorem:

> If Y is the minimum fixpoint of the equation F(y),
> and Z is the maximum fixpoint of the equation $\neg F(\neg y)$,
> then $||Y|| = P - ||Z||$, where P is the set of all states.

This will be demonstrated by means of an example. Consider the fixpoint equation of the previous section:

$$Y = \langle - \rangle Y \vee \langle x \rangle T$$

We have already calculated the minimum fixpoint for this equation, it is {A1}.

To show the duality between the maximum and minimum fixpoints, we must find the maximum fixpoint of the dual equation, i.e. find $\neg F(\neg y)$.

- The dual of $\langle - \rangle Y \vee \langle x \rangle T$ is $\neg(\langle - \rangle(\neg Y) \vee \langle x \rangle T$;
- which equals, $\neg\langle - \rangle(\neg Y) \wedge \neg\langle x \rangle T$;
- which by duality of $\Diamond$ and $\Box$ equals, $[-]Y \wedge \neg\langle x \rangle T$;
- The maximum fixpoint of this equation is the set {A2, A3, A4}; (The proof is left as an exercise for the reader)
- P - {A2, A3, A4} = {A1}
- Hence, since {A1} is also the minimum fixpoint of $\langle - \rangle Y \vee \langle x \rangle T$, we conclude that for this example, the maximal and minimal fixpoints are related by the duality equation.

It should be noted that the maximum fixpoint is usually defined as the dual of the minimum, and a theorem is used to show that:

- if Y is the minimum fixpoint of F(y) then $Y = \cap\{S \mid F(S) \subseteq S\}$
- if Z is the maximum fixpoint of F(y) then $Z = \cup\{S \mid S \subseteq F(S)\}$

This presentation has motivated the notion of maximal fixpoint being a super set of all other fixpoints, and minimal fixpoint being a subset of all other fixpoints.

3.2.7 Conclusions

This section gave a brief introduction to fixpoints in models of modal logic:

- a fixpoint equation is of the form y = F(y);
- a fixpoint equation will have a solution set if there are an even number of negations preceding each reference to y in the equation F(y).
- the solutions need not be unique;
- solutions may be thought of as being "consistent beliefs"; if we assume that y is a solution and everything outside y isn't we have a consistent system.
- since the intersection and union of solutions are subsets and supersets of solutions, there is a greatest and least solution called the maximum and minimum fixpoints.
- the maximum fixpoint corresponds to everything except that which can not possibly be a part of the solution; the minimum fixpoint corresponds to only that which must be a a part of the solution; other solutions correspond to consistent beliefs; that which must be part of the solution plus that which might be part of the solution.

3.2.8 An Exercise

In the introduction to this chapter, a definition of a monotonic function was given. In this chapter, the fixpoint equations were of the form Y = F(Y), where F is a function over a set of states. Show that this is true of each of the functions used in this chapter:

- $\|Y = \langle\text{-}\rangle Y \vee \langle x \rangle T\|$

- $\|Y = [-]Y \vee \langle x \rangle T\|$

It was stated that provided there were an even number of negations prefixing the reference to the variable Y. Check whether the following function is monotonic, and see if there are any solutions to the fixpoint equation.

- $\|Y\| = \|\langle - \rangle \neg Y \vee \langle x \rangle T\|$
- $\|Y\| = \|\neg [-] Y \vee \langle x \rangle T\|$

3.3 Logical Description of Fixpoint

In the previous section we described the model for a fixpoint as being a set of states which satisfy a fixpoint equation. It is also possible to give a characterization of the maximum and minimum fixpoints under the rules of deduction called "satisfies" which were introduced in the previous section.

3.3.1 Minimum Fixpoint

The minimum fixed point of an equation may be calculated as follows:

- assume that the fixpoint equation is false for all states;
- calculate the result of the fixpoint function based on this assumption;
- if the result is different from the assumption, repeat the process using the new values.

As an example, consider the system from the previous section and the fixpoint function $Y = \langle d \rangle T \vee \langle - \rangle Y$. The minimum fixpoint of this function includes all states which can perform a "d" action or any state which can by some sequence of actions eventually reach such a state.

Calculating the fixpoint using the method above we find:

- Assuming $Y_0 = F$, the set of states for which the equation $\langle d \rangle T \vee \langle - \rangle Y$ is true is $||Y_1|| = \{A4\}$.
- Assuming $||Y_1|| = \{A4\}$ and substituting in the equation, we find that $\langle d \rangle T \vee \langle - \rangle (A4)$ is true for $Y_2 = \{A4, A2\}$.
- Assuming $||Y_2|| = \{A4,A2\}$ and substituting in the equation, we find that $\langle d \rangle T \vee \langle - \rangle (A4 \vee A2)$ is true for $Y_3 = \{A4, A2, A1\}$.
- Assuming $||Y_3|| = \{A4,A2,A1\}$ and substituting in the equation, we find that $\langle d \rangle T \vee \langle - \rangle (A4 \vee A2 \vee A1)$ is true for $Y_4 = \{A4, A2, A1\}$.

There is no point in continuing the iteration, as we have reached the minimum fixed point and further iterations will only yield the same answer.

3.3.2 Maximum Fixpoint

The maximum fixpoint may be calculated as follows:

- assume that the fixpoint equation is true for all states;
- calculate the result of the fixpoint function based on this assumption;
- if the result is different from the assumption, repeat the process using the new values.

As an example, consider the system from the previous section and the fixpoint function Z = [d]F∧[-]Z. The maximum fixpoint of this function includes all states which can not perform a "d" action and any state which can not eventually reach a state where a "d" action is possible.

Calculating the fixpoint using the method above we find:

- Assuming Z_0 = T, the set of states for which the equation [d]F∧[-]Z is true is $||Z_1||$ = {A1,A2,A3}.
- Assuming $||Z_1||$ = {A1,A2,A3} and substituting in the equation, we find that [d]F∧[-]Z) is true for Z_2 = {A1, A3}.
- Assuming $||Z_2||$ = {A1, A3} and substituting in the equation, we find that [d]F∧[-]Z is true for Z_3 = {A3}.
- Assuming $||Z_3||$ = {A3} and substituting in the equation, we find that [d]F∧[-]Z is true for Z_4 = {A3}.

There is no point in continuing the iteration, as we have reached the fixed point and further iterations will only yield the same answer.

3.3.3 Conclusions

In general, minimum fixpoints express "liveness" properties which state that some desired property is possible or will eventually happen. Maximum fixpoints are used to express "safety" properties which state that some undesired property is not possible or will not happen. Due to the duality of the maximum and minimum fixpoints, we can express either safety or liveness in terms of either fixpoint, but if we restrict ourselves to fixpoints without negations, then least fixpoints will, in general express liveness. Some examples are:

- Property P holds everywhere is expressed as a maximal fixpoint: ||Z|| = ||P∧[-]Z||
- Property P eventually holds (i.e. it holds in this state, or if the state has at least one derivative and the property holds in every derivative is a minimal fixpoint: ||Z|| = ||P∨[-]Z∧¡-¿T||

3.4 Notation for Fixpoints in Modal Mu

We have already given the semantics for what it means to the maximum or minimum fixpoint to be true of some state. It means the state is a member of the set of states which make up that fixpoint. This section defines the syntax which is used to discuss fixpoints.

In the previous section a fixpoint equation was written as "Y = ⟨d⟩T ∨⟨-⟩Y". This implicitly defines a function together with arguments for it. The syntax for this logic uses the expressions μX.formula and νX.formula to represent the minimal and maximal fixpoints and to bind the variable X which is free in the formula. Thus, the minimal and maximal fixpoints of this equation would be written as

- μY.⟨d⟩T ∨⟨-⟩Y
- νY.⟨d⟩T ∨⟨-⟩Y

It is now possible to describe the syntax of HML, with labelled modalities, sets of events, and fixpoints, and it is worth while to do so.

$$A ::= T \mid \neg A \mid A \wedge A \mid \langle \mathrm{K} \rangle A \mid (A) \mid Z \mid \mu Z.A$$

where:

- A is a formula of Hennessy-Milner Logic;
- T, or "true" is the predefined proposition whose valuation includes every state;
- ¬ is the negation of a formula;
- ∧ is the conjunction of two formulae;
- ⟨K⟩ is a modal operator called "possibility" which operates on either a single action or a set of actions;
- () are parenthesis which give precedence to operations inside them.
- Z is a free propositional variable used as an argument to the fixpoint operator;
- μZ.A is the minimal fixpoint of the formula "A".

As earlier, these formulae are usually extended with the definition of the following derived operators:

- The atomic formula F is defined as $\neg$T.
- A$\vee$B is defined as $\neg(\neg A \wedge \neg B)$
- [K]A is defined as $\neg\langle K\rangle\neg A$, and is called "necessity";
- νZ.A is defined as $\neg\mu(\neg Z).A$, and is called the maximal fixpoint of A.

3.4.1 Notation on the Workbench

The workbench uses the following notation for maximum and minimum fixpoints:

```
max(Z.<a>T & [-]Z)
min(Y.<a>T + [-]Y)
```

3.4.2 Some Common Facts

Some common formulae and their interpretations:

$\mu Z.\langle K\rangle Z$	always false for any K
$\nu Z.[K]Z$	always true for any K
$\nu Z.\langle\tau\rangle Z$	divergence

It is again easy to see why $\mu Z.\langle K\rangle Z$ is always false. This is the minimal solution to the equation $Z = \langle K\rangle Z$. Since we are trying to find the minimal fixpoint, we assume that $\emptyset$ is a solution to the equation in which case we can replace $\langle K\rangle Z$ by $\langle K\rangle F$ which is always false (see Chapter 3). Since $\langle K\rangle F$ is always false irrespective of the agents $\emptyset$ is the only solution to the equation which represents "always false". Similarly, when solving for $\nu Z.[K]Z$ we start will the set of all states. If a member of K is not a possible action in that state, the proposition is vacuously true. If a member of K is a possible action, then all that the propostion says is that after this action we end up in some state that is a member of the set of all states. For the proposition, $\nu Z.\langle\tau\rangle Z$ we start with the set of all states. The proposition reads, "the property Z is the ability to do a tau action and end up in a state with the property Z". Thus, the states which belong the this fixpoint have the ability to do an infinite number of tau transitions.

3.5 Use of Fixpoints

This section uses a simple hardware device called a sequencer to demonstrate some of the concepts of minimum and maximum fixpoints, and to familiarize the reader with the CWB. First, define the sequencer.

```
Command: bi
Identifier: A
Agent: a.x1.'p.x2.A

Command: bi
Identifier: B
Agent: b.x1.'q.x2.B

Command: bi
Identifier: N
Agent: n.'x1.'x2.N

Command: bi
Identifier: SEQ
Agent: (A | B | N) \{x1, x2}
```

We want the sequencer to output a `'p` after receiving an `a`, or output a `'q` after receiving a `b`, but both of these outputs are gated by the input `n`. To show the behavior of this device we print out the *visible sequences* `vs` of length `4`.

```
Command: vs
Number: 4
Agent: SEQ
=== a b n 'q ===>
=== a b n 'p ===>
=== a n 'p a ===>
=== a n 'p b ===>
=== a n 'p n ===>
=== a n b 'p ===>
=== a n b 'q ===>
=== b a n 'q ===>
=== b a n 'p ===>
=== b n a 'q ===>
=== b n 'q a ===>
=== b n 'q b ===>
=== b n 'q n ===>
=== b n a 'p ===>
=== n a 'p a ===>
=== n a 'p b ===>
=== n a 'p n ===>
=== n a b 'p ===>
=== n a b 'q ===>
=== n b a 'q ===>
=== n b 'q a ===>
```

```
=== n b 'q b ===>
=== n b 'q n ===>
=== n b a 'p ===>
```

One property which is of interest has to do with delay insensitive hardware. A delay insensitive device obeys a protocol such that:

- If the delay insensitive device is permitted by its environment to produce an output, the environment will remain receptive and will not attempt to block the output;
- After generating the output, the delay insensitive device will not generate the same output until after the environment has signalled its willingness to accept it, by some input action;
- Dually, if the environment of the delay insensitive device is permitted to generate an input for the device, the device must remain receptive until after the environment has either generated the input or has signalled its agreement to not produce such an input via some other input signal.

Delay insensitive devices characterize the same problem as flow control in protocols. In order to check for delay insensitivity, a macro called CYCLIC is generated. It checks that the stream of traces generated or allowed by the device is an action, followed by an enabling action. This is done with the `bmi` (bind macro identifier) command.

```
Command: bmi
Identifier: CYCweak

Enter parameters, separated by commas or spaces.
   -- Proposition and action set parameters begin with an uppercase letter,
   -- action parameters otherwise.

Parameters: action enabling_action
Body: max(X.<<action>>T & [[enabling_action]]F & [[-action]]X & [[action]](\
  max(Y.[[action]]F & [[enabling_action]]X & [[-enabling_action]]Y)))

Command: bmi
Identifier: CYCstrong

Enter parameters, separated by commas or spaces.
   -- Proposition and action set parameters begin with an uppercase letter,
   -- action parameters otherwise.

Parameters: action enabling_action
Body: max(X.<action>T & [enabling_action]F & [-action]X & [action](\
  max(Y.[action]F & [enabling_action]X & [-enabling_action]Y)))
```

Two such macros have been defined; one using the strong and one the weak modalities. In the strong modality, tau actions count as a transition. In the weak modality, tau actions are considered to be part of a single atomic action.

The strong macro may be read as:

> in state **X** , an **action** is possible, and an **enabling_action** is not possible, and this will continue to be the case no matter what happens as long as it isn't an **action**. Further, if an **action** does happen, the agent must go to state Y, where an **action** is not possible, and all non **enabling_action** actions, will take us back to this state, and if an **enabling_action** happens the agent must go to state X,

Below, the workbench checks the agent SEQ for the property that:

> An input **a** is possible, and output **'p** is not possible, and this will continue to hold until the input **a** happens. Once this input occurs, the input may not re-occur despite any non **'p** actions. If the output **'p** occurs input **a** is possible again.

This is done for both the weak and strong versions using the **cp** (check proposition) command.

```
Command: cp
Agent: SEQ
Proposition: CYCstrong a 'p
false
Command: cp
Agent: SEQ
Proposition: CYCweak a 'p

[Major collection... 39% used (840824/2109360), 490 msec]

[Major collection... 39% used (824752/2098516), 460 msec]

[Major collection... 39% used (822020/2095356), 440 msec]

[Major collection... 38% used (800740/2097640), 440 msec]
true
```

The proposition **CYCstrong a 'p** failed on agent **SEQ** because after doing a **'p** action it is not possible to do an **a** action. It is necessary to do the

x2 action first. **CYCweak a 'p** succeeds on agent **SEQ** because all of the tau actions have been ignored. The rational is that an action like **'p** followed by a tau is really one atomic action, and the description of this as being a **'p** followed by tau is an aid to understanding. It should be noted that checking for the strong modality took a few seconds and checking for the weak modality took about a minute.

CYCweak has all actions as weak modalities; many of these could be changed to strong modalities. It is left as an exercise for the reader to figure out which.

In order to speed matters up, the agent **SEQ** is minimized with the command **min**. To show that the agent is truly smaller, the **size** command shows the number of states.

```
Command: min
Agent: SEQ
Save result in identifier: SEQ'

SEQ' has 12 states.

Command: size SEQ

SEQ has 16 states.

Command: size SEQ'

SEQ' has 12 states.
```

The proposition **CYCstrong a 'p** is computed on the minimized agent **SEQ'**. This agent has the transitory tau transitions removed, and the strong proposition succeeds on this agent even though it fails on the non minimized agent. If strong proposition succeeds on minimized agents which have the transitory states removed, the weak proposition will succeed on non minimized agents; but, it will do so much faster. If there are tau transitions at the beginning of a choice, the strong proposition may fail though the weak succeeds.

```
Command: cp SEQ'
Proposition: CYCstrong a 'p
true
```

Note that above the command **cp** and its first argument **SEQ'** were put on the same line. More experienced users of the workbench may find it more convenient to anticipate the prompts. Later in this example, this is also done with the the **bi** command.

Is it justified to use weak modalities instead of strong ones? That depends on whether the **x2** action is just an operator on two states or whether

there is real processing going on. If there is real processing going on, (and hence real delay) then the environment may not present another `a` to the sequencer until after this processing has finished; but the environment can not know when the processing has finished, as the only signal which it received was `'p` which was output before the processing started. Hence, if there is real processing this is not a delay insensitive agent. Let us suppose that there is real processing going on. Below the agent `A` is changed to make it a delay insensitive agent `Adi`.

```
Command: bi Adi
Agent: a.Adi1

Command: bi Adi1
Agent: x1.'p.(x2.Adi + a.x2.Adi1)

Command: bi Bdi
Agent: b.Bdi1

Command: bi Bdi1
Agent: x1.'q.(x2.Bdi + b.x2.Bdi1)

Command: bi SEQdi
Agent: (Adi | Bdi | M) \{x1, x2}
```

Here, the agent is willing to accept an `a` after outputting a `'p`. Below, this is checked on the workbench and the similar property for `b` and `'p`.

```
Command: cp SEQdi
Proposition: CYCstrong a 'p

[Major collection... 36% used (764596/2094184), 360 msec]
true
Command: cp SEQdi
Proposition: CYCstrong b 'q
true
```

After analysing the actions `a`, `b`, `'p` and `'q` for delay insensitivity, we next examine `n`. What are the necessary conditions for `n` being possible? We would expect that input `n` should be allowed after a `'p`, or a `'q` action. The macros which we introduced previously only allowed single actions. In this example, sets of operations are used. Sets are denoted with an uppercase letter to the `bmi` command, and are defined by the `bsi` (bind set identifier) command.

```
Command: bmi CYCset_weak

Enter parameters, separated by commas or spaces.
   -- Proposition and action set parameters begin with an uppercase letter,
```

```
   -- action parameters otherwise.

Parameters: action ENABLING_ACTIONS
Body: max(X.<<action>>T & [[ENABLING_ACTIONS]]F & [[-action]]X & [[action]](\
  max(Y.[[action]]F & [[ENABLING_ACTIONS]]X & [[-ENABLING_ACTIONS]]Y)))

Command: bmi CYCset_strong

Enter parameters, separated by commas or spaces.
   -- Proposition and action set parameters begin with an uppercase letter,
   -- action parameters otherwise.

Parameters: action ENABLING_ACTIONS
Body: max(X.<action>T & [ENABLING_ACTIONS]F & [-action]X & [action](\
  max(Y.[action]F & [ENABLING_ACTIONS]X & [-ENABLING_ACTIONS]Y)))
Command: bsi
Identifier: PQ
Enter action list: 'p 'q
```

Below, the agent `SEQdi` is checked to see whether it can accept another `n` after outputting a `'p` or a `'q`.

```
Command: cp SEQdi
Proposition: CYCset_strong n PQ
false
Command: cp SEQ
Proposition: CYCset_weak n PQ
true
Command: cp SEQ'
Proposition: CYCset_strong n PQ
true
```

Unsurprisingly, it exhibits the same problems as the original sequencer. If the action `x2` represents real processing, then the agent is not delay insensitive. We will not continue along these lines, as making the semaphore delay insensitive will complicate matters without providing more insight. This point of the last exercise was to demonstrate the set capabilities of the workbench.

3.5.1 Exercise for the Reader

All of the macros used the maximal fixpoint rather than the minimal fixpoint.

- Check that an agent which input an `a` and halted (`a.nil`) would satisfy the proposition `CYCstrong a 'p`.

- Determine what should be done to guarantee that output will eventually be generated. HINT: what does

```
max(X.<action>T & [enabling_action]F & [-action]X & [action](\
  max(Y.[action]F & [enabling_action]X & [-enabling_action]Y)))
```

describe?

Bibliography

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

[Mol91] F. G. Moller. The Edinburgh Concurrency Workbench, Version 6.0. Tech Report, Computer Science Department, University of Edinburgh, 1991.

[Sti91] C. Stirling. An Introduction to Modal and Temporal Logics for CCS. In *Proceedings of the 1989 Joint UK/Japan Workshop on Concurrency*, pages 2–20, New York, 1991. Lecture Notes in Computer Science 491, Springer Verlag.

[SW91] C. Stirling and D. Walker. Local model checking in the modal μ-calculus. *Theoretical Computer Science*, 89:161–177, 1991.

[Wal87] D. Walker. Introduction to a Calculus of Communicating Systems. Technical Report ECS-LFCS-87-22, Laboratory for the Foundations of Computer Science, University of Edinburgh, 1987.

[Win91] G. Winskel. A note on model checking the modal ν-calculus. *Theoretical Computer Science*, 83:157–167, 1991.