Specification and Verification
of Distributed Systems

Peter Ladkin

Office: D6-131; Tel: (106)-5326
Computer Lab: D6-129; Tel: (106)-5325
Sprechstunden: Wednesday 11.00-12.00
`ladkin@rvs.uni-bielefeld.de`
`http://www.rvs.uni-bielefeld.de`

All course notes are available *only* via WWW under my home page or RVS course page.

In the course, we study how to perform verifications, using the TLA+ specification language and the TLA logic. We use an implementation of a bounded buffer and part of a Lazy Caching algorithm as examples.

In the Seminar, we will work through a set of preliminary specifications of the Radio Link Protocol, a form of sliding window protocol, that will be used in the TDMA cellular phone technology.

Formal proofs in *TLA+* exist:—

- A buffer defined as an array implements an abstract buffer:

  Formal but Lively Buffers in TLA+

- The *Lazy Caching* algorithm of *Afek*, *Brown* and *Merritt* (ACM Transactions on Programming Languages and Systems, Jan 1993) implements a *sequentially consistent* memory

  Lazy Caching: An Assertional View
  Ladkin, Lamport, Roegel and Olivier
  Distributed Computing, to appear.

Section 1: General Background

What is a specification?

It is a means of describing what a system is supposed to do. When the system does what the specification says, the system *fulfils* the specification.

How do we know when a system fulfils a specification?

We prove it mathematically.

Can we just look at the code and see that it works?

No. Mistakes are almost always made.

Do mathematicians do formal proofs?

Generally, no they don't. They are concerned with proving a hard theorem. Their proofs are semi-formal.

Why should computer scientists do formal proofs?

Because, if they work with *concurrent systems*, many, many mistakes have been made in published algorithms, even in the best journals.

E.g., Lamport's *Bakery algorithm* for mutual exclusion is incorrectly explained in many operating system textbooks.

What is a concurrent system?

One that isn't sequential.

What is a sequential system?

One that isn't concurrent.

OK,OK, a sequential system is one that executes one *thread of control*. A concurrent system has many threads of control.

What's a thread of control?

Consider your favorite C program. It starts at the beginning, moves on down the code statements, goes into subroutines and returns, and finishes (maybe). Control passes from one statement to the next.

At any point, control is at a unique place. The *thread* is the list of statements executed by the program.

How can I have many threads?

Build an operating system, for example (a real one, like *Unix*).

You may have LATEX running in one *window*, while you're performing *shell commands* in another, and viewing a compiled LATEX file in another, using *xdvi*.

Or, you have a *transaction processing* machine, like the one which processes your *bank card* money withdrawals from an *Automated Teller Machine (ATM)*. It must handle many transactions at the same time. Each one forms a different thread.

Or, you have an airplane such as the Airbus A320/A330/A340 series, in which the pilot's control commands are input to a *flight control computer* which interacts with the *flight guidance management system* and also the *autopilot*. The flight control computer figures out itself how best to interpret the pilot's commands, and moves the control surfaces according to its calculations.

Isn't this risky?

Look at the RISKS-Digest
(ed. by Peter Neumann) at

`http://catless.ncl.ac.uk/Risks/`

for articles, including many by me, about the A320/A330/A340 accidents/incidents that may concern the computers.

So what are the technical problems?

A typical (classical) one: two threads want to access the same memory location at the same time: one to read, another to write.

How do I write such concurrent programs?

With a great deal of care.

What are the industrial problems?

- How to encourage industrial developers to write specifications of their systems at a high-enough level to permit logical analysis

- How to provide them the tools to do this

What sort of tools are useful?

- A formal specification language

- A proof method, that shows how you prove systems fulfil their specifications

- A automated proof-checker that helps in doing low-level proof steps

Is there just a system, and a specification?

No, normally there are many intermediate specifications. Even the system itself is described by a very-low-level specification.

People nowadays don't try to prove real code correct, they work with a very close description of the operation of the code, but written in a *specification language*.

One needs to show that each specification is fulfilled by the next more-detailed specification.

How does one do this?

There are many notions of *fulfil*. The most straightforward one is *logical implication*. The high-level specification is a logical formula and this is implied by the logical formula that is the lower-level specification

History: this *hierarchichal method* was first developed by SRI International for defining and proving the SIFT flight control system correct. SRI's verification system was known as HDM (H*ierarchical* D*evelopment* M*ethodology*), and was superceded by EHDM (E*xtended* HDM).

SIFT details at

`http://www.csl.sri.com/ft-history.html`

SRI has an ongoing *Formal Methods* program. Details at

`http://www.csl.sri.com/sri-csl-fm.html`

More about some interesting aircraft control system computer '*features*' is in John Rushby's home page (link on the Formal Methods page), under '*Images and Anomalies*'.

What are some of the discoveries about control system concurrency?

The problems of

Byzantine Agreement: How can many processors agree on a correct value when one of them is lying?

For example, two flight control computers calculate values and check with each other to make sure they calculated the same value.

But they didn't!

Hint: you can't figure out which one is wrong with only two *channels* (processors).

The Airbus A320/A330/A340 control system uses two identical channels for fault-tolerance. So they can't tolerate Byzantine faults. But do they need to?

The Boeing B777 control system has *only one* flight control computer—but the back-up is a *very simple* direct-law control system, and the switch is under pilot control.

Which is better? Who knows?

Clock Synchronisation: The AFTI-F16 aircraft had two independent channels that worked without synchronising the clocks of each channel. They got *slightly different* readings from the sensors, with interesting consequences. See Rushby.

Arbitration: Buridan's Ass. (Lamport has pointed out that the arbitration problem exists in all analogue-to-digital conversions.) Apparently, engineers don't worry about arbitration.

What about a problem with transaction processing systems?

Sequential Consistency: Suppose many transactions are happening to the same account at the same time. If they're all *reads*, no problem. If some of them are *writes*, there may be a problem. How do we ensure that the system behaves as if one transaction happened after the other (even if they really all happened at once)?

Specification Methods for Concurrent Systems

- Logic–the Assertional Method
- Algebra–process algebras
- Petri Nets
- Various industrial methods that are insufficient for verifying the system but good for organising it.

The Assertional Method

One treats the system as a *state machine*. This state machine may have infinitely many states. For each possible system action, one describes (by a logical formula) a property of the state that *must* hold for the action to take place (a *precondition*) and a property of the state that must hold after the action has happened (a *postcondition*).

For sequential systems, developed by Floyd and Hoare (late 1960's). For concurrent systems, by Ashcroft (1975), further developed by Owicki and Gries (1976).

Temporal logic methods (Pnueli 1977) are most suitable for proofs of *liveness* properties (Owicki, Lamport, Manna, Pnueli, and many others).

A good incarnation is TLA, the *Temporal Logic of Actions* and its associated specification language TLA+.

If `Q` is a logical formula expressing a system property, then $\Box$`Q` expresses that it *always* holds, and $\Diamond$`Q` that it will hold at some state in the future.

A specification has the form

$$Init \ \wedge \ \Box S \ \wedge \textit{Fair}$$

where S is the safety property (which must always hold in any state), and *Fair* says

$$\Box\Diamond\neg \textit{Enabled}(A) \ \vee \ \Box\Diamond A$$

or something similar.

Process Algebra

CCS: Robin Milner

CSP: Tony Hoare

Processes are fundamental, and are defined by algebraic equations:

```
P = a ; P
```

is a process that executes `a`, then repeats.

```
P = a ; P'
P' = b ; P
```

describes two processes.
`P` executes `a`, then `P'`, which executes `b` and returns to `P`.

The actions, however, are *joint* between two or more processes:

```
S = P || P' || P''
P = a ; P
P' = a ; b ; P'
P'' = b ; P''
```

is a system `S` which executes a joint action between `P` and `P'`, then
one between `P'` and `P''`, then ....

We will study the assertional method for specifying and verifying systems.

We will learn what a proof is, and how to do one.

We will discuss the essential classes of system properties:

safety and *liveness*

Every system property is a conjunction of a safety property with a liveness property (Alpern and Schneider 1987).

And to learn about the tools that exist, join in my seminar.

Section 2: Writing Specifications in TLA+

First, an abstract buffer with two operations

`push(something)` and `pop`

The figure shows the state of the buffer before each operation and the state after

$$\begin{array}{ccc} \text{buffer} & & \text{buffer}' \\ < \text{b,x,y,d} > & \xrightarrow{\text{pop}} & < \text{x,y,d} > \\ & & \\ < \text{b,x,y,d} > & \xrightarrow{\text{push(a)}} & < \text{b,x,y,d,a} > \end{array}$$

Here is a more concrete version of the buffer

It's an array

$\perp$ is the symbol for '**`nothing here`**'

Buffer $\qquad\qquad\qquad\qquad$ Buffer$'$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & \perp \\ \hline\end{array} \xrightarrow{\text{push(a)}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & a \\ \hline\end{array}$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & a \\ \hline\end{array} \xrightarrow{\text{move(6)}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & d & \perp & a \\ \hline\end{array}$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & d & \perp & a \\ \hline\end{array} \xrightarrow{\text{move(2)}} \begin{array}{|c|c|c|c|c|c|c|}\hline b & \perp & x & y & d & \perp & a \\ \hline\end{array}$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline b & \perp & x & y & d & \perp & a \\ \hline\end{array} \xrightarrow{\text{pop}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & \perp & x & y & d & \perp & a \\ \hline\end{array}$$

Here's how the concrete buffer is supposed to correspond to the abstract buffer as a data structure

ConcreteBuffer								AbstractBuffer
$\perp$	b	x	y	$\perp$	d	$\perp$	$\longrightarrow$	$<$ b,x,y,d $>$
$\perp$	b	x	y	$\perp$	d	a	$\longrightarrow$	$<$ b,x,y,d,a $>$
$\perp$	b	x	y	d	$\perp$	a	$\longrightarrow$	$<$ b,x,y,d,a $>$
b	$\perp$	x	y	d	$\perp$	a	$\longrightarrow$	$<$ b,x,y,d,a $>$

And here's how the operations correspond

Buffer $\qquad\qquad\qquad$ Buffer$'$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & \perp \\ \hline\end{array} \xrightarrow{\text{push(a)}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & a \\ \hline\end{array}$$

$$< \text{b,x,y,d} > \xrightarrow{\text{push(a)}} < \text{b,x,y,d,a} >$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & a \\ \hline\end{array} \xrightarrow{\text{move(6)}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & d & \perp & a \\ \hline\end{array}$$

$$< \text{b,x,y,d,a} > \xrightarrow{\texttt{no-op}} < \text{b,x,y,d,a} >$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & d & \perp & a \\ \hline\end{array} \xrightarrow{\text{move(2)}} \begin{array}{|c|c|c|c|c|c|c|}\hline b & \perp & x & y & d & \perp & a \\ \hline\end{array}$$

$$< \text{b,x,y,d,a} > \xrightarrow{\texttt{no-op}} < \text{b,x,y,d,a} >$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline b & \perp & x & y & d & \perp & a \\ \hline\end{array} \xrightarrow{\text{pop}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & \perp & x & y & d & \perp & a \\ \hline\end{array}$$

$$< \text{b,x,y,d,a} > \xrightarrow{\text{pop}} < \text{x,y,d,a} >$$

How does this fit together?

The concrete buffer *simulates* the abstract buffer

- they start in ‘equivalent’ states

- every action of the concrete buffer corresponds either to an action or to a non-action of the abstract buffer

- when the concrete buffer is sufficiently ‘live’, then the abstract buffer actually does some desired action

This method of *state machine simulation* is common to many methods, for example

- TLA of Lamport

- Input/Output machines of Tuttle, Lynch, Vaandrager

- the method of Lam and Shankar (also TL-based)

An alternative is to have actions only—then the operation of the system is an abstract machine simulation, but not a *state* machine simulation, since one doesn't have *state*

How does one specify the actions?

Here's one from the abstract buffer

module *AbstractBuffer*

actions

$$
\begin{aligned}
push(a) \triangleq\ & \land\ a \in Data \\
& \land\ Len(buffer) < N \\
& \land\ buffer' = buffer \circ \langle a \rangle
\end{aligned}
$$

Why is it written like this?

Let's compare two ways of writing this action

module *AbstractBuffer*

actions

$$push(a) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Len(buffer) < N \\ \land\ buffer' = buffer \circ \langle a \rangle \end{array}$$

badly-written actions

$$push(a) \triangleq$$
$$a \in Data \land Len(buffer) < N \land buffer' = buffer \circ \langle a \rangle$$

If that didn't persuade you, try this

module *AbstractBuffer*

actions

$$push(a|b) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Len(buffer) < N \\ \land\ \lor\ buffer' = buffer \circ \langle a \rangle \\ \quad\ \lor\ buffer' = buffer \circ \langle b \rangle \end{array}$$

badly-written actions

$$push(a|b) \triangleq$$
$$a \in Data \land Len(buffer) < N \land$$
$$(buffer' = buffer \circ \langle a \rangle \lor buffer' = buffer \circ \langle b \rangle)$$

Now we shall see how to write the two specifications. First, we define the starting states of the variables.

module *AbstractBuffer*

predicates

$$Init \triangleq buffer = \langle\,\rangle$$

module *ConcreteBuffer*

predicates

$$Init \triangleq \forall\, n \in 1..N : Buffer[n] = \bot$$

Next, we define the *push* actions

module *Buffer Actions*

abstract actions

$$push(a) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Len(buffer) < N \\ \land\ buffer' = buffer \circ \langle a \rangle \end{array}$$

concrete actions

$$push(a) \triangleq \begin{array}{l} \land\ a \in Data \\ \land\ Buffer[N] = \bot \\ \land\ Buffer'[N] = a \\ \land\ \forall\, i \in 1..(N-1) : \\ \qquad \text{unchanged } Buffer[i] \end{array}$$

And now the *pop* actions

module *Buffer Actions*

abstract actions

$$pop \triangleq \begin{array}{l} \land\ Len(buffer) > 0 \\ \land\ buffer' = tail(buffer) \end{array}$$

concrete actions

$$pop \triangleq \begin{array}{l} \land\ Buffer[1] \neq \bot \\ \land\ Buffer'[1] = \bot \\ \land\ \forall\, i \in 2..N\ : \\ \qquad \text{unchanged } Buffer[i] \end{array}$$

And there's one concrete action left

module *Buffer Actions*

abstract actions

$$no - op \triangleq \text{unchanged } buffer$$

concrete actions

$$
\begin{aligned}
move(k) \triangleq\ & \land\ k \in 2..N \\
& \land\ Buffer[k] \neq \bot \\
& \land\ Buffer[k-1] = \bot \\
& \land\ Buffer'[k] = \bot \\
& \land\ Buffer'[k-1] = Buffer[k] \\
& \land\ \forall\, i \in 1..(k-2)\ : \\
& \qquad \text{unchanged } Buffer[i] \\
& \land\ \forall\, i \in (k+1)..N\ : \\
& \qquad \text{unchanged } Buffer[i]
\end{aligned}
$$

We have the *initial conditions* and the *actions*. But we don't yet have a *specification*

A specification defines the initial conditions and the actions—

and also the sentence that says

- the system starts in the initial condition

- if variables change values, it must be because of a defined action (safety)

- it's always true that
  some desired action eventually happens
  if it can (liveness)

First some notation.

$[A]_x$ means $A \vee (x' = x)$

$[A]_{\langle x, y \rangle}$ means $A \vee (x' = x \wedge y' = y)$

Intuitively, $[A]_x$ means

Either A or x *doesn't change value*

module *AbstractBuffer*

imports

$Sequences$

parameters

$buffer$: variable
$Data, N$: constant

predicates

$$Init \triangleq buffer = \langle\,\rangle$$

module *AbstractBuffer (continued)*

actions

$$
\begin{aligned}
push(a) \;\triangleq\; & \land\; a \in Data \\
& \land\; Len(buffer) < N \\
& \land\; buffer' = buffer \circ \langle a \rangle
\end{aligned}
$$

$$
\begin{aligned}
pop \;\triangleq\; & \land\; Len(buffer) > 0 \\
& \land\; buffer' = tail(buffer)
\end{aligned}
$$

temporal

$$
\begin{aligned}
Spec \;\triangleq\; & \land\; Init \\
& \land\; \Box[pop \lor \exists\, b \,:\, push(b)]_{buffer} \\
& \land\; WF_{buffer}(pop)
\end{aligned}
$$

module *ConcreteBuffer*

parameters

$Buffer$: variable
$Data, N$: constant

assertions

$\perp \notin Data$

predicates

$Init \triangleq \land \forall n \in 1..N : Buffer[n] = \perp$

module *ConcreteBuffer (continued)*

actions

$$
\begin{aligned}
push(a) \;\triangleq\;\; & \land\ a \in Data \\
& \land\ Buffer[N] = \bot \\
& \land\ Buffer'[N] = a \\
& \land\ \forall\, i \in 1..(N-1) : \\
& \qquad \text{unchanged } Buffer[i] \\
pop \;\triangleq\;\; & \land\ Buffer[1] \neq \bot \\
& \land\ Buffer'[1] = \bot \\
& \land\ \forall\, i \in 2..N : \text{unchanged } Buffer[i]
\end{aligned}
$$

module *ConcreteBuffer (continued)*

actions

$$
move(k) \triangleq \begin{array}{l}
\land\ k \in 2..N \\
\land\ Buffer[k] \neq \bot \\
\land\ Buffer[k-1] = \bot \\
\land\ Buffer'[k] = \bot \\
\land\ Buffer'[k-1] = Buffer[k] \\
\land\ \forall\, i \in 1..(k-2) : \\
\qquad \text{unchanged } Buffer[i] \\
\land\ \forall\, i \in (k+1)..N : \\
\qquad \text{unchanged } Buffer[i]
\end{array}
$$

module *ConcreteBuffer (will this never end?)*

temporal

$$
\begin{array}{lcl}
Spec & \triangleq & \wedge\ Init \\
 & & \wedge\ \Box[\vee\ pop \\
 & & \quad\ \ \vee\ \exists\, b\ :\ push(b) \\
 & & \quad\ \ \vee\ \exists\, k\ :\ move(k)]_{Buffer} \\
 & & \wedge\ WF_{Buffer}(pop) \\
 & & \wedge\ WF_{Buffer}(\exists\, k\ :\ move(k))
\end{array}
$$

Section 3: Verification in TLA

Verification means that one has

- A description of an implementation

- A specification

One must prove that the description of the implementation fulfils the specification.

For logical methods, '*fulfils*' = '*implies*'

For real systems, one must use *hierarchical* methods in order to control the complexity.

Hierarchical methods: one describes an implementation I, then a more abstract view A_1 and proves that

$$I \Rightarrow A_1$$

One describes then an even more abstract view A_2 and proves that

$$A_1 \Rightarrow A_2$$

But if that's true, of course, then simply

$$I \Rightarrow A_2$$

and there's little need for the intermediate step.

It's when there are many separate parts that need to be brought together that hierarchical decomposition pays off.

A simple hierarchical decomposition:

Level 1 a database distributed over 5 different sites with query points over many more

Level 2
- a specification of a serial database
- a specification of a database split into 5 pieces
- a specification of a reliable protocol for queries

Level 3
- a specification of an implementation of a serial database
- a specification of an implementation of a database split into 5 pieces
- a specification of an implementation of a reliable protocol for queries

We start at the beginning.
We have two specifications.

module *Specifications*

temporal

$$
\begin{array}{ll}
Conc - Buffer - Spec \;\triangleq & \\
& \land\ Init \\
& \land\ \Box[\lor\ pop \\
& \qquad \lor\ \exists\, b\ :\ push(b) \\
& \qquad \lor\ \exists\, k\ :\ move(k)]_{Buffer} \\
& \land\ WF_{Buffer}(pop) \\
& \land\ WF_{Buffer}(\exists\, k\ :\ move(k)) \\
Abs - Buffer - Spec \;\triangleq & \\
& \land\ Init \\
& \land\ \Box[\lor\ pop \\
& \qquad \lor\ \exists\, b\ :\ push(b)]_{buffer} \\
& \land\ WF_{buffer}(pop)
\end{array}
$$

We want to show that

Conc-Buffer-Spec $\Rightarrow$ *Abs-Buffer-Spec*

But it doesn't!

Conc-Buffer-Spec has a variable *Buffer* which doesn't occur in *Abs-Buffer-Spec*

Abs-Buffer-Spec has a variable *buffer* which doesn't occur in *Conc-Buffer-Spec*

But are these variables essential? We merely want to *specify* a buffer, without really caring what specific object is a buffer.

An implementation builds a data structure *buffer* from 'concrete' data structures that it contains.

In 'real' programming languages, a set is 'implemented' as a bit array. The set (abstract data structure) is a statically-defined function of the basic data structure (bit array = sequence of adjacent words).

We define such a function: call it $\overline{buffer}$. We want to know that this **always** behaves as a buffer, no matter how the underlying 'concrete' data structures behave.

Where predicate A contains an occurrence of the variable *buffer*, let $\overline{A}$ be $A[buffer/\overline{buffer}]$, that is, A with all occurrences of $buffer$ replaced by $\overline{buffer}$.

Thus we prove

$\forall$*Buffer* :
Conc–Buffer–Spec $\Rightarrow$ $\overline{\textit{Abs–Buffer–Spec}}$

Or, more formally, ...

module *Theorems*

include *ConcreteBuffer* **as** *CB(N)*
include *AbstractBuffer* **as** *AB(N)*

$\overline{buffer} \triangleq \ldots\ldots\ldots\ldots$

theorems

$$\forall Buffer \, : \, CB(N).Spec \Rightarrow \overline{AB(N).Spec}$$

What does **include** mean?

It means that the variables, definitions, operations, from $\mathit{ConcreteBuffer}$ are visible in this buffer, with the same names, except that every operation or temporal definition is prefixed with the given name of the **include**-ed module.

Similarly for $\mathit{AbstractBuffer}$.

One can **include** variables and constants with names other than the original ones – see *Michael Blume*'s Einführung or the *TLA+ Manual*.

Section 4: Proofs in TLA

Now to the proof.
The proof is formal, formally laid out in a hierarchical style, in which substeps constitute a logically correct proof of each step.

Technically, it is an *inverse natural deduction* proof style.

There are two numbering schemes in *pf.sty*

One is absolute—

the.full.path.number

One is relative—

$\langle$TheLevelNumber$\rangle$TheStepNumber

Here are examples of the same proof with both schemes.

1. $A \wedge (P \wedge Q \wedge R)$
   Proof:
   1.1. A
      Proof: I guess A just is true. $\square$
   1.2. $P \wedge Q \wedge R$
      Proof:
      1.2.1. $P \wedge Q$
         Proof:
         1.2.1.1. P
            Proof: I guess P just is true. $\square$
         1.2.1.2. Q
            Proof: I guess Q just is true. $\square$
         1.2.1.3. Q.E.D.
            Proof: Conjoin 1.2.1.1 and 1.2.1.2. $\square$
      1.2.2. R
         Proof: I guess R just is true. $\square$
      1.2.3. Q.E.D.
         Proof: Conjoin 1.2.1 and 1.2.2. $\square$
   1.3. Q.E.D.
      Proof: Conjoin 1.1 and 1.2. $\square$

$\langle 1 \rangle 1.\ A \wedge (P \wedge Q \wedge R)$

Proof:

$\langle 2 \rangle 1.\ A$

Proof: I guess A just is true. $\square$

$\langle 2 \rangle 2.\ P \wedge Q \wedge R$

Proof:

$\langle 3 \rangle 1.\ P \wedge Q$

Proof:

$\langle 4 \rangle 1.\ P$

Proof: I guess P just is true. $\square$

$\langle 4 \rangle 2.\ Q$

Proof: I guess Q just is true. $\square$

$\langle 4 \rangle 3.$ Q.E.D.

Proof: Conjoin $\langle 4 \rangle 1$ and $\langle 4 \rangle 2$. $\square$

$\langle 3 \rangle 2.\ R$

Proof: I guess R just is true. $\square$

$\langle 3 \rangle 3.$ Q.E.D.

Proof: Conjoin $\langle 3 \rangle 1$ and $\langle 3 \rangle 2$. $\square$

$\langle 2 \rangle 3.$ Q.E.D.

Proof: Conjoin $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$. $\square$

Proof step numbers in the buffer example are relative.

See if you can assign *path-numbers* to the proof steps as we go

So, how do we prove

$\langle 0 \rangle 1.\ \forall Buffer : CB(N).Spec \Rightarrow \overline{AB(N).Spec}$??

We can prove it if we treat $Buffer$ like a variable (which it is):

$\langle 1 \rangle 1.\ CB(N).Spec \Rightarrow \overline{AB(N).Spec}$

Remember, we have a *state function*
$\overline{buffer} \ \triangleq\ f(Buffer)$
that interprets $buffer$.

By convention, we let

$$\overline{buffer} \;\triangleq\; f(Buffer)$$

(this is the *Refinement Mapping*) and

$\overline{AB(N).Spec}$ is $AB(N).Spec$, with every occurrence of $buffer$ replaced by $\overline{buffer}$.

Then, we prove

$\langle 2 \rangle 1.$ $CB(N).Spec \Rightarrow \overline{AB(N).Spec}$

Predicate logic allows us to conclude what we want from this.

Each specification is a conjunction

$$Init \wedge \Box[action \vee action \vee \ldots]_{variables} \wedge Liveness$$

It seems most reasonable to prove the conjunction bit by bit:

$\langle 3 \rangle 1.$ $CB(N).Spec \Rightarrow \overline{AB(N).Init}$

$\langle 3 \rangle 2.$ $CB(N).Spec \Rightarrow \Box[\vee\ \overline{AB(N).pop}$
$\qquad\qquad\qquad\qquad \vee\ \overline{\exists\, a : AB(N)push(a)}]_{\overline{buffer}}$

$\langle 3 \rangle 3.$ $CB(N).Spec \Rightarrow \overline{WF_{buffer}(AB(N).pop)}$

$\langle 3 \rangle 4.$ Q.E.D.

Proof: Follows by propositional logic from the conjunction of steps $\langle 3 \rangle 1$, $\langle 3 \rangle 2$ and $\langle 3 \rangle 3$. $\Box$

In fact, it turns out that we may prove

$\langle 3 \rangle 1.$ $CB(N).Spec \Rightarrow \overline{AB(N).Init}$

Proof:

$\langle 4 \rangle 1.$ $CB(N).Init \Rightarrow \overline{AB(N).Init}$

$\langle 4 \rangle 2.$ Q.E.D.

Proof: Follows directly by propositional logic. $\square$

$\langle 3 \rangle 2.$ $CB(N).Spec \Rightarrow \Box[\vee\ \overline{AB(N).pop}$
$\qquad \vee\ \overline{\exists\, a\ :\ AB(N)push(a)}]_{\overline{buffer}}$

Proof:

$\langle 4 \rangle 1.$ $\wedge\ CB(N).Init$
$\wedge\ \Box[\vee\ CB(N).pop$
$\qquad \vee\ \exists\, a\ :\ CB(N).push(a)$
$\qquad \vee\ \exists\, k\ :\ CB(N).move(k)]_{Buffer}$
$\Rightarrow \Box[\vee\ \overline{AB(N).pop}$
$\qquad \vee\ \overline{\exists\, a\ :\ AB(N)push(a)}]_{\overline{buffer}}$

$\langle 4 \rangle 2.$ Q.E.D.

Proof: Follows directly by propositional logic. $\Box$

Consider now the step

$\langle 4 \rangle 1.$ $CB(N).Init \Rightarrow \overline{AB(N).Init}$

Proof: .... is simply math. $\square$

We may prove

$$\begin{array}{l}
\langle 4 \rangle 1.\ \Box[\vee\ CB(N).pop \\
\qquad\quad \vee\ \exists\, a\ :\ CB(N).push(a) \\
\qquad\quad \vee\ \exists\, k\ :\ CB(N).move(k)]_{Buffer} \\
\qquad\qquad\qquad \Rightarrow \Box[\vee\ \overline{AB(N).pop} \\
\qquad\qquad\qquad\qquad \vee\ \overline{\exists\, a\ :\ AB(N)push(a)}]_{\overline{buffer}}
\end{array}$$

Proof:

$$\begin{array}{l}
\langle 5 \rangle 1.\ (\vee\ CB(N).pop \\
\qquad\quad \vee\ \exists\, a\ :\ CB(N).push(a) \\
\qquad\quad \vee\ \exists\, k\ :\ CB(N).move(k) \\
\qquad\quad \vee\ Buffer' = Buffer) \\
\qquad\qquad\qquad \Rightarrow (\vee\ \overline{AB(N).pop} \\
\qquad\qquad\qquad\qquad \vee\ \overline{\exists\, a\ :\ AB(N)push(a)} \\
\qquad\qquad\qquad\qquad \vee\ \overline{buffer}' = \overline{buffer})
\end{array}$$

⟨5⟩2. Q.E.D.

Proof: Follows directly by the TLA rule STL.4 $\dfrac{F \Rightarrow G}{\Box F\ \Rightarrow\ \Box G}$ □

Let's do it......

$\langle 5 \rangle 1.$ $(\vee\ CB(N).pop$
$\quad \vee\ \exists\, a\ :\ CB(N).push(a)$
$\quad \vee\ \exists\, k\ :\ CB(N).move(k)$
$\quad \vee\ Buffer' = Buffer)$
$\qquad \Rightarrow (\vee\ \overline{AB(N).pop}$
$\qquad\quad \vee\ \overline{\exists\, a\ :\ AB(N)push(a)}$
$\qquad\quad \vee\ \overline{buffer}' = \overline{buffer})$

Proof:

$\langle 6 \rangle 1.$ $CB(N).pop \Rightarrow \overline{AB(N).pop}$

$\langle 6 \rangle 2.$ $\exists\, a\ :\ CB(N).push(a) \Rightarrow$
$\qquad \exists\, a\ :\ \overline{AB(N).push(a)}$

$\langle 6 \rangle 3.$ $\exists\, k\ :\ CB(N).move(k) \Rightarrow \overline{buffer}' = \overline{buffer}$

$\langle 6 \rangle 4.$ $Buffer' = Buffer \Rightarrow \overline{buffer}' = \overline{buffer}$

$\langle 6 \rangle 5.$ Q.E.D.

Proof: Follows by propositional logic from $\langle 6 \rangle 1$, $\langle 6 \rangle 2$, $\langle 6 \rangle 3$, and $\langle 6 \rangle 4$:

$$(A \Rightarrow X) \wedge (B \Rightarrow Y) \wedge (C \Rightarrow Z) \wedge (D \Rightarrow Z)$$
$$\Rightarrow (A \vee B \vee C \vee D \Rightarrow X \vee Y \vee Z)\ \square$$

Now, part of this says:
a concrete $push$ is an abstract $push$.

$\langle 6 \rangle 2.$ $\exists\, a : CB.push(a)$

$$\Rightarrow$$

$$\overline{\exists\, a : AB.push(a)}$$

Proof:

Let: a : constant

$\langle 7 \rangle 1.$ $CB.push(a)$

$$\Rightarrow$$

$$\overline{AB.push}(a)$$

$\langle 7 \rangle 2.$ Q.E.D.

Follows from $\langle 7 \rangle 1$ by predicate logic. $\square$

$\langle 7 \rangle 1.$ $CB.push(a)$

$$\Rightarrow$$

$$\overline{AB.push}(a)$$

Proof:

$\langle 8 \rangle 1.$ $CB.push(a) \Rightarrow a \in Data$

Proof: Immediate from the definition of $CB.push$. $\square$

$\langle 8 \rangle 2.$ $CB.push(a) \Rightarrow Len(\overline{buffer}) < N$

$\langle 8 \rangle 3.$ $CB.push(a) \Rightarrow \overline{buffer}' = \overline{buffer} \circ \langle a \rangle$

$\langle 8 \rangle 4.$ Q.E.D.

Proof: Follows immediately from $\langle 8 \rangle 1$, $\langle 8 \rangle 2$ and $\langle 8 \rangle 3$ using propositional logic. $\square$

$\langle 8 \rangle 2.$ $CB.push(a) \Rightarrow Len(\overline{buffer}) < N$

Proof:

$\langle 9 \rangle 1.$ $CB.push(a) \Rightarrow Buffer[N] = \bot$

Proof: Immediate from the definition of $CB.push(a)$. $\square$

$\langle 9 \rangle 2.$ $Buffer[N] = \bot \Rightarrow$

$$Len(SelectSeq(Buffer, NonVoid)) < N$$

Proof: Follows from the definition of $SelectSeq$ and Len, along with a certain amount of data structure manipulation, which is omitted. $\square$

$\langle 9 \rangle 3.$ Q.E.D.

Proof: Follows from $\langle 9 \rangle 1$, $\langle 9 \rangle 2$ and the definition of $\overline{buffer}$ by propositional logic. $\square$

What do all these symbols mean?

Let: $$\begin{aligned} NonVoid(k) &\triangleq k \neq \bot \\ \overline{buffer} &\triangleq SelectSeq(Buffer, NonVoid) \\ FirstFull &\triangleq Buffer[1] \neq \bot \\ NotEmpty &\triangleq \exists\, i \in 1 \,..\, N : Buffer[i] \neq \bot \end{aligned}$$

module *Sequences*

import *Naturals*

$$m \,..\, n \triangleq \{i \in Nat : (m \leq i) \land (i \leq n)\}$$

$$Len(s) \triangleq$$
$$\quad \text{choose } n : (n \in Nat) \land ((\text{Domain } s) = (1 \,..\, n))$$

$$Head(s) \triangleq s[1]$$

$$Tail(s) \triangleq [i \in 1 \,..(Len(s) - 1) \mapsto s[i + 1]]$$

$$s \circ t \triangleq$$
$$\quad [i \in 1 \,..(Len(s) + Len(t)) \mapsto$$
$$\qquad \textbf{if } i \leq Len(s) \textbf{ then } s[i]$$
$$\qquad\qquad \textbf{else } t[i - Len(s)]]$$

$$Seq(S) \triangleq \text{union } \{[(1 \,..\, n) \rightarrow S] : n \in Nat\}$$

$$SubSeq(s, m, n) \triangleq$$
$$\quad [i \in (1 \,..(1 + n - m)) \mapsto s[i + m - 1]]$$

module *Sequences (continued)*

$$
\begin{array}{l}
\mathit{SelectSeq}(s, \mathit{test}(\_)) \;\triangleq \\
\textbf{let}\;\; F[\,t \,:\, \mathit{Seq}(\{s[i] \,:\, i \in (1 \,..\, \mathit{Len}(s))\})\,] \;\triangleq \\
\qquad\qquad \textbf{if}\;\; t = \langle\,\rangle \;\;\textbf{then}\;\; \langle\,\rangle \\
\qquad\qquad\qquad\qquad\quad \textbf{else}\;\; \textbf{if}\;\; \mathit{test}(\mathit{Head}(t)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle\,\mathit{Head}(t)\,\rangle \circ \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad F[\,\mathit{Tail}(t)] \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\;\; F[\,\mathit{Tail}(t)] \\
\textbf{in}\;\;\; F[s]
\end{array}
$$

$\langle 8 \rangle 3.$ $CB.push(a) \Rightarrow \overline{buffer'} = \overline{buffer} \circ \langle\, a \,\rangle$

Proof:

$\langle 9 \rangle 1.$ $CB.push(a) \Rightarrow$
$$SelectSeq(Buffer, NonVoid) = SelectSeq([i \in 1\,..(N-1) \mapsto Buffer[i]], NonVoid)$$

Proof:

$\langle 10 \rangle 1.$ $CB.push(a) \Rightarrow Buffer[N] = \bot$

Proof: Immediate from the definition of $CB.push(a)$. $\square$

$\langle 10 \rangle 2.$ $Buffer[N] = \bot$
$$\Rightarrow$$
$$SelectSeq(Buffer, NonVoid) = SelectSeq([i \in 1\,..(N-1) \mapsto Buffer[i]], NonVoid)$$

Proof: Follows immediately from the definition of $SelectSeq$ using manipulations of the data structure. $\square$

$\langle 10 \rangle 3.$ Q.E.D.

Proof: Follows immediately from $\langle 10 \rangle 1$ and $\langle 10 \rangle 2$ by propositional logic. $\square$

$\langle 9 \rangle 1.$ (was on last slide)

$\langle 9 \rangle 2.$ $CB.push(a)$

$$\Rightarrow$$
$$Buffer' =$$
$$[i \in 1\,..(N-1) \mapsto Buffer[i]] \circ \langle\, a \,\rangle$$

Proof: Follows from the definition of $CB.push$ and the sequence operations. $\square$

$\langle 9 \rangle 3.$ $SelectSeq([i \in 1\,..(N-1) \mapsto Buffer[i]]$
$$\circ \langle\, a \,\rangle, NonVoid) =$$
$$SelectSeq([i \in 1\,..(N-1) \mapsto Buffer[i]], NonVoid)$$
$$\circ \langle\, a \,\rangle$$

Proof: Follows from the definition of $SelectSeq$ and $NonVoid$. $\square$

$\langle 9 \rangle 4.$ Q.E.D.

Proof: Follows immediately from $\langle 9 \rangle 1$, $\langle 9 \rangle 2$, $\langle 9 \rangle 3$ by propositional logic, substitution, and the definition of $\overline{buffer}$. $\square$

This completes the proof.
All the substeps have been proved.

Now we'll take a look at the
TLA special inference rules, and the
proof scheme for TLA verifications

Section 5: More About Proofs in TLA+

We return to the *safety step*

$$
\begin{array}{l}
\langle 4\rangle 1.\ \Box[\vee\ CB(N).pop \\
\qquad\quad \vee\ \exists\, a\ :\ CB(N).push(a) \\
\qquad\quad \vee\ \exists\, k\ :\ CB(N).move(k)]_{Buffer} \\
\qquad\qquad\quad \Rightarrow \Box[\vee\ \overline{AB(N).pop} \\
\qquad\qquad\qquad\quad \vee\ \overline{\exists\, a\ :\ AB(N)push(a)}]_{\overline{buffer}}
\end{array}
$$

$$
\begin{array}{l}
\langle 5\rangle 1.\ (\vee\ CB(N).pop \\
\qquad\quad \vee\ \exists\, a\ :\ CB(N).push(a) \\
\qquad\quad \vee\ \exists\, k\ :\ CB(N).move(k) \\
\qquad\quad \vee\ Buffer' = Buffer) \\
\qquad\qquad\quad \Rightarrow (\vee\ \overline{AB(N).pop} \\
\qquad\qquad\qquad\quad \vee\ \overline{\exists\, a\ :\ AB(N)push(a)} \\
\qquad\qquad\qquad\quad \vee\ \overline{buffer}' = \overline{buffer})
\end{array}
$$

$\langle 5\rangle$2. Q.E.D.

Proof: directly by the TLA rule

STL.4 $\dfrac{F \Rightarrow G}{\Box F\ \Rightarrow\ \Box G}$

What TLA proof rules are there?

First, **Temporal Logic rules**

These are valid for linear-time temporal logic

STL1 : $\dfrac{F \text{ a tautology}}{\Box F}$

STL2. $\vdash \Box F \;\Rightarrow\; F$

STL3. $\vdash \Box\Box F \;\equiv\; \Box F$

STL4. $\dfrac{F \Rightarrow G}{\Box F \;\Rightarrow\; \Box G}$

STL5. $\vdash \Box(F \wedge G) \;\equiv\; (\Box F) \wedge (\Box G)$

STL6. $\vdash (\Diamond\Box F) \wedge (\Diamond\Box G) \;\equiv\; \Diamond\Box(F \wedge G)$

STL7. $\vdash \Box\Diamond\Box F \equiv \Diamond\Box F$

LATTICE.

$\succ$ a well–founded partial order on a set S

c not free in F or G

$$\frac{F \wedge (\mathsf{c} \in \mathsf{S}) \;\Rightarrow\; (H_{\mathsf{c}} \leadsto (G \vee \exists \mathsf{d} \in \mathsf{S} : (\mathsf{c} \succ \mathsf{d}) \wedge H_{\mathsf{d}}))}{F \;\Rightarrow\; ((\exists \mathsf{c} \in \mathsf{S} : H_{\mathsf{c}}) \leadsto G)}$$

What's a *well-founded partial order*?

order: the numbers are ordered by $<$
but it's *total*:
all numbers are *comparable*

Sets are ordered by $\subset$
and it's *partial*:
not every pair of sets is comparable

Sequences are ordered by *prefix*
it's also partial

well-founded: if you follow a
chain of $x_1 > x_2 > x_3 >$ .....
it always stops

Try it with numbers, say $x_1 = 510$

Try it with sequences, say
$x_1 = \langle 5, 7, 24, 610, 7, 5, 4, 2, 3, 2 \rangle$

Lattice is used for *fairness* proofs.
It's a way of ensuring that *recursion stops*
(always provided you've written the recursion
properly)

Think of G as the *ground clause* and
H_c as the recursive call with parameter c

Use of temporal logic in *safety* proofs
is *pro forma*, as we shall see

Second, **Basic TLA Rules**
These specifically involve the TLA 'prime' operator.

TLA1. $$\frac{P \wedge (f' = f) \;\Rightarrow\; P'}{\Box P \;\equiv\; P \wedge \Box[\mathcal{P} \Rightarrow \mathcal{P}']_f}$$

TLA2. $$\frac{P \wedge [\mathcal{A}]_f \;\Rightarrow\; Q \wedge [\mathcal{B}]_g}{\Box P \wedge \Box[\mathcal{A}]_f \;\Rightarrow\; \Box Q \wedge \Box[\mathcal{B}]_g}$$

Third, **Invariance Rules**

INV1. $$\frac{I \wedge [\mathcal{N}]_f \Rightarrow I'}{I \wedge \Box[\mathcal{N}]_f \Rightarrow \Box I}$$

INV2. $\vdash \Box I \;\Rightarrow\; (\Box[\mathcal{N}]_f \;\equiv\; \Box[\mathcal{N} \wedge \mathcal{I} \wedge \mathcal{I}']_f)$

Fourth, **Weak Fairness Rules**

WF1.

$$\frac{\begin{array}{l} P \wedge [\mathcal{N}]_f \;\Rightarrow\; (P' \vee Q') \\ P \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \;\Rightarrow\; Q' \\ P \;\Rightarrow\; \mathit{Enabled}\, \langle \mathcal{A} \rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \;\Rightarrow\; (P \leadsto Q)}$$

WF2.

$$\frac{\begin{array}{l} \langle \mathcal{N} \wedge \mathcal{B} \rangle_f \;\Rightarrow\; \langle \overline{\mathcal{M}} \rangle_{\overline{g}} \\ P \wedge P' \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \wedge \overline{\mathit{Enabled}\, \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathcal{B} \\ P \wedge \overline{\mathit{Enabled}\, \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathit{Enabled}\, \langle \mathcal{A} \rangle_f \\ \Box[\mathcal{N} \wedge \neg \mathcal{B}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \wedge \Box F \\ \qquad\qquad \wedge \Diamond\Box\overline{\mathit{Enabled}\, \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \Diamond\Box P \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \wedge \Box F \;\Rightarrow\; \overline{\mathrm{WF}_g(\mathcal{M})}}$$

where

F, G, H_{c} are TLA formulas
P, Q, I are predicates
$\mathcal{A}$, $\mathcal{B}$, $\mathcal{N}$, $\mathcal{M}$ are actions
f, g are state functions

Now to the proof scheme.

In the proof scheme, the steps that are new for each verification are indicated by *******Novel.******* in the proof

The *proof scheme* helps to control the proof:

$\langle 0 \rangle 1.$ $(\forall y : Init_1 \wedge \Box[N_1]_v \wedge L_1)$
$\quad\quad \Rightarrow (\overline{Init_2} \wedge \Box[\overline{N_2}]_{\overline{w}} \wedge \overline{L_2})$

Proof:

$\langle 1 \rangle 1.$ $(Init_1 \wedge \Box[N_1]_v \wedge L_1)$
$\quad\quad \Rightarrow (\overline{Init_2} \wedge \Box[\overline{N_2}]_{\overline{w}} \wedge \overline{L_2})$

Proof:

$\langle 2 \rangle 1.$ $Init_1 \Rightarrow \overline{Init_2}$

Proof: ******Novel.******. $\Box$

$\langle 2 \rangle 2.$ $Init_1 \wedge \Box[N_1]_v \Rightarrow \Box\overline{[N_2]_w}$

Proof: See next slides. $\Box$

$\langle 2 \rangle 3.$ $Init_1 \wedge \Box[N_1]_v \wedge L_1 \Rightarrow \overline{L_2}$

Proof: ******Novel.******. $\Box$

$\langle 2 \rangle 4.$ Q.E.D.

Proof: By propositional logic from $\langle 2 \rangle 1$, $\langle 2 \rangle 2$, $\langle 2 \rangle 3$. $\Box$

$\langle 1 \rangle 2.$ Q.E.D.

Proof: By predicate logic from $\langle 1 \rangle 1$. $\Box$

Let's look at *safety*:

$\langle 2 \rangle 2.$ $Init_1 \wedge \Box[N_1]_v \Rightarrow \Box\overline{[N_2]_w}$

Proof: First we must find an invariant:

$\langle 3 \rangle 1.$ $Init_1 \wedge \Box[N_1]_v \Rightarrow \Box Inv$

$\langle 3 \rangle 2.$ $\Box Inv \wedge \Box[N_1]_v \Rightarrow \Box[\overline{N_2}]_{\overline{w}}$

$\langle 3 \rangle 3.$ Q.E.D.

Proof: Immediate from $\langle 3 \rangle 1$ and $\langle 3 \rangle 2$ by propositional logic. $\Box$

Now the substeps for the invariant:

$\langle 3 \rangle 1.\ Init_1 \land \Box[N_1]_v \Rightarrow \Box Inv$

$\langle 4 \rangle 1.\ Init_1 \Rightarrow Inv$

Proof: ******Novel.****** $\Box$

$\langle 4 \rangle 2.\ Inv \land \Box[N_1]_v \Rightarrow \Box Inv$

Proof:

$\langle 5 \rangle 1.\ Inv \land [N_1]_v \Rightarrow Inv'$

Proof: ******Novel.****** $\Box$

$\langle 5 \rangle 2.$ Q.E.D.

Proof: Immediate from $\langle 5 \rangle 1$ by Rule INV1. $\Box$

$\langle 4 \rangle 3.$ Q.E.D.

Proof: Immediate from $\langle 4 \rangle 1$ and $\langle 4 \rangle 2$ by propositional logic. $\Box$

Now the rest:

$\langle 3 \rangle 2.$ $\Box Inv \wedge \Box[N_1]_v \Rightarrow \Box[\overline{N_2}]_{\overline{w}}$

Proof:

$\langle 4 \rangle 1.$ $\Box Inv \wedge \Box[N_1]_v \Rightarrow \Box[N_1 \wedge Inv \wedge Inv']_v$

Proof: Immediate from INV2 using STL5. $\Box$

$\langle 4 \rangle 2.$ $\Box[N_1 \wedge Inv \wedge Inv']_v \Rightarrow \Box[\overline{N_2}]_{\overline{w}}$

Proof:

$\langle 5 \rangle 1.$ $[N_1 \wedge Inv \wedge Inv']_v \Rightarrow [\overline{N_2}]_{\overline{w}}$

Proof: ******Novel.******. $\Box$

$\langle 5 \rangle 2.$ Q.E.D.

Proof: Immediate from $\langle 5 \rangle 1$ by STL4. $\Box$

$\langle 4 \rangle 3.$ Q.E.D.

Proof: Immediate from $\langle 4 \rangle 1$ and $\langle 4 \rangle 2$ by propositional logic. $\Box$

Section 5: Get **Live**!
(If it's *safe*)

$\langle 5 \rangle 1.$ $Inv \wedge [N_1]_v \Rightarrow Inv'$

$\langle 6 \rangle 1.$ $Inv \wedge (N_1 \vee v' = v) \Rightarrow Inv'$

Proof: Suppose $N_1 \triangleq A_1 \vee A_2 \vee A_3$, where

A_1, A_2, A_3 are the actions of the implementing module.

$\langle 7 \rangle 1.$ $Inv \wedge A_1 \Rightarrow Inv'$

Proof: *******************

$\langle 7 \rangle 2.$ $Inv \wedge A_2 \Rightarrow Inv'$

Proof: *******************

$\langle 7 \rangle 3.$ $Inv \wedge A_3 \Rightarrow Inv'$

Proof: *******************

$\langle 7 \rangle 4.$ $Inv \wedge v' = v \Rightarrow Inv'$

Proof: Since the only variables in Inv are v, this is immediate. $\square$

$\langle 7 \rangle 5.$ Q.E.D.

Proof: Propositional Logic from $\langle 7 \rangle 1$, $\langle 7 \rangle 2$, $\langle 7 \rangle 3$, $\langle 7 \rangle 4$. $\square$

$\langle 5 \rangle 1.$ $[N_1 \wedge Inv \wedge Inv']_v \Rightarrow [\overline{N_2}]_{\overline{w}}$

Proof:

$\langle 6 \rangle 1.$ $A_1 \wedge Inv \wedge Inv' \Rightarrow [\overline{N_2}]_{\overline{w}}$

Proof: *****************

$\langle 6 \rangle 2.$ $A_2 \wedge Inv \wedge Inv' \Rightarrow [\overline{N_2}]_{\overline{w}}$

Proof: ****************

$\langle 6 \rangle 3.$ $A_3 \wedge Inv \wedge Inv' \Rightarrow [\overline{N_2}]_{\overline{w}}$

Proof: ***************

$\langle 6 \rangle 4.$ $(v' = v) \Rightarrow [\overline{N_2}]_{\overline{w}}$

Proof:

$\langle 7 \rangle 1.$ $v' = v \Rightarrow \overline{w}' = \overline{w}$

Proof: Since the only variables in $\overline{w}$ are v, this is immediate. $\square$

$\langle 7 \rangle 2.$ Q.E.D.

Proof: Immediate by propositional logic from $\langle 7 \rangle 1$. $\square$

$\langle 6 \rangle 5.$ Q.E.D.

Proof: Propositional Logic from $\langle 6 \rangle 1$, $\langle 6 \rangle 2$, $\langle 6 \rangle 3$, $\langle 6 \rangle 4$. $\square$

(Meta-Theorem!)
Thus, we can see that a *safety proof* exists providing only that the *proof obligations*

$Init_1 \Rightarrow \overline{Init_2}$

For every action A
in the antecedent Spec
$Inv \wedge A \Rightarrow Inv'$

For every action A
in the antecedent Spec
$A \wedge Inv \wedge Inv' \Rightarrow (\overline{N_2} \vee \overline{w}' = \overline{w})$

are proved.

In fact, we can go further, in principle.
Write every action A in *disjunctive normal form*

$$A \Leftrightarrow B^{A}{}_{1} \vee B^{A}{}_{2} \vee \ldots.$$

Then, by propositional logic, we only need

$$Init_1 \Rightarrow \overline{Init_2}$$

and

$$Inv \wedge B \Rightarrow Inv'$$

$$B \wedge Inv \wedge Inv' \Rightarrow (\overline{N_2} \vee \overline{w}' = \overline{w})$$

for every disjunct B of any action
in the antecedent spec.

But it's less clean to write and remember.

What kind of thing is an invariant?
Remember, it must remain true through every action.

Type information, for example

What else??

The buffer example is so straightforward that the invariant is just $(N_1 \vee v' = v)$ itself
and I didn't go down to the step level.

But in general one needs to go down to the step level

What about those awful *liveness properties*?

We need to prove

$$\overline{WF_{\mathit{buffer}}(pop)}$$

given

$$WF_{\mathit{Buffer}}(pop) \wedge WF_{\mathit{Buffer}}(\exists\, k \,:\, move(k))$$

and the initial condition and safety property of *ConcreteBuffer*

Intuitively, $\overline{WF_{\mathit{buffer}}(pop)}$ means

When pop *can* go, it must–sometime.
I.e., when pop is continually enabled,
it must go sometime

Is $\overline{AB.pop}$ enabled? Is $CB.pop$ enabled?

ConcreteBuffer AbstractBuffer

$$\boxed{\perp}\boxed{\perp}\boxed{b}\boxed{\perp}\boxed{d}\boxed{\perp}\boxed{\perp} \longrightarrow < b,d >$$

Intuitively, $AB.pop$ is enabled when the $buffer$ is nonempty. So $Buffer$ is also nonempty, but there
may not be anything in $Buffer[1]$, so $CB.pop$
is not enabled.

So $AB.pop$ may be enabled, even though
$CB.pop$ is not

What's the formal definition of $WF_v(A)$?

$$\Box\Diamond\neg \mathit{Enabled}\langle A\rangle_v \lor \Box\Diamond\langle A\rangle_v$$

that is

$$\Box\Diamond\neg \mathit{Enabled}(A \land v' \neq v) \lor \Box\Diamond(A \land v' \neq v)$$

Remember $[A]_v \;\triangleq\; (A \lor v' = v)$

Likewise, we can write $\langle A\rangle_v \;\triangleq\; (A \land v' \neq v)$

Then $\neg[\neg A]_v \Leftrightarrow \langle A\rangle_v$
and $\neg\langle\neg A\rangle_v \Leftrightarrow [A]_v$
and $\neg[A]_v \Leftrightarrow \langle\neg A\rangle_v$
and $\neg\langle A\rangle_v \Leftrightarrow [\neg A]_v$
etc.

What's the formal definition of $Enabled$?

Suppose I have an action such as

$$pop \triangleq \land Len(buffer) > 0 \\ \land buffer' = tail(buffer)$$

It has variable $buffer$, and next value $buffer'$.

If I say *there is* a next value, that means I *can* do pop, in principle

That's exactly what $Enabled$ says:

$$Enabled(pop) \triangleq \\ \exists\, newvalue : \land Len(buffer) > 0 \\ \land newvalue = tail(buffer)$$

In general, let action A have flexible variables y, z

$$Enabled(A) \;\triangleq\; \exists\, newy, newz : A(y'/newy, z'/newz)$$

where

- $newy$ and $newz$ are new symbols not otherwise occurring in A;

- $A(y'/newy, z'/newz)$ is A, but with $newy$ substituted for every occurrence of y', similarly for $newz$

So, $Enabled(pop) \triangleq$

$$\begin{array}{l} \exists\, newvalue : \land\ Len(buffer) > 0 \\ \qquad\qquad\qquad\quad \land\ newvalue = tail(buffer) \\ \qquad\qquad\qquad\quad \Leftrightarrow \\ \land\ Len(buffer) > 0 \\ \land\ \exists\, newvalue : newvalue = tail(buffer) \end{array}$$

But $\exists\, newvalue : newvalue = tail(buffer)$ is true!

Why? Let $\phi(x) \triangleq (x = tail(buffer))$.
Then $\phi(tail(buffer))$ is true
(it's an axiom for equality!)
So, we may invoke the rule $\dfrac{\phi(t)}{\exists x : \phi(x)}$

to infer that $\exists\, newvalue : \phi(newvalue)$ is true

So

$$\begin{array}{l}\land\ Len(buffer) > 0 \\ \land\ \exists\, newvalue : newvalue = tail(buffer)\end{array}$$

$$\Leftrightarrow$$

$$\begin{array}{l}\land\ Len(buffer) > 0 \\ \land\ \text{true}\end{array}$$

$$\Leftrightarrow$$

$$Len(buffer) > 0$$

Similarly,

$$Enabled(CB.pop \land Buffer' \neq Buffer)$$

$$\Leftrightarrow$$

$$\exists\, Newbuf : \begin{array}{l} \land\ Buffer[1] \neq \bot \\ \land\ Newbuf[1] = \bot \\ \land\ \forall\, i \in 2..N : Newbuf[i] = Buffer[i] \end{array}$$

$$\Leftrightarrow$$

$$\begin{array}{l} \land\ Buffer[1] \neq \bot \\ \land\ \exists\, Newbuf : \begin{array}{l} \land\ Newbuf[1] = \bot \\ \land\ \forall\, i \in 2..N : Newbuf[i] = Buffer[i] \end{array} \end{array}$$

We'll show

$$\exists \, Newbuf \; : \; \land \; Newbuf[1] = \bot$$
$$\qquad\qquad\qquad \land \; \forall \, i \in 2..N \; : \; Newbuf[i] = Buffer[i]$$

is true. Define

$$TheRealNewBuf \;\triangleq\; [Buffer \text{ except } !1 = \bot]$$

Then

$$\land \; TheRealNewBuf[1] = \bot$$
$$\land \; \forall \, i \in 2..N \; : \; TheRealNewBuf[i] = Buffer[i]$$

follows immediately from the definition

So, $Enabled\langle \, CB.pop \, \rangle_{Buffer} \Leftrightarrow (Buffer[1] \neq \bot)$

Finally, you try it now with $\exists\, k\ :\ CB.move(k)$.

it's very similar to $CB.pop$

Next time:

Using the **WF** rules

Remember the **Weak Fairness Rules**

WF1.

$$\frac{\begin{array}{l} P \wedge [\mathcal{N}]_f \;\Rightarrow\; (P' \vee Q') \\ P \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \;\Rightarrow\; Q' \\ P \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge \mathsf{WF}_f(\mathcal{A}) \;\Rightarrow\; (P \leadsto Q)}$$

WF2.

$$\frac{\begin{array}{l} \langle \mathcal{N} \wedge \mathcal{B} \rangle_f \;\Rightarrow\; \langle \overline{\mathcal{M}} \rangle_{\overline{g}} \\ P \wedge P' \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathcal{B} \\ P \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \\ \Box[\mathcal{N} \wedge \neg\mathcal{B}]_f \wedge \mathsf{WF}_f(\mathcal{A}) \wedge \Box F \\ \qquad\qquad \wedge \Diamond\Box\overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \Diamond\Box P \end{array}}{\Box[\mathcal{N}]_f \wedge \mathsf{WF}_f(\mathcal{A}) \wedge \Box F \;\Rightarrow\; \overline{\mathsf{WF}_g(\mathcal{M})}}$$

where

F, G, H_{c} are TLA formulas
P, Q, I are predicates
$\mathcal{A}$, $\mathcal{B}$, $\mathcal{N}$, $\mathcal{M}$ are actions
f, g are state functions

$\langle 3 \rangle$1. $\land\ \Box[\lor\ \exists\, a\ :\ CB.push(a)$
$\lor\ CB.pop$
$\lor\ \exists\, k\ :\ CB.move(k)]_{Buffer}]$
$\land\ WF_{Buffer}(CB.pop)$
$\land\ WF_{Buffer}(\exists\, k\ :\ CB.move(k))$
$\Rightarrow$
$\overline{WF_{buffer}(AB.pop)}$

Proof:

$\langle 4 \rangle$1. $\langle \lor\ \exists\, a\ :\ CB.push(a)$
$\lor\ CB.pop$
$\lor\ \exists\, k\ :\ CB.move(k)) \land CB.pop \rangle_{Buffer}$
$\Rightarrow$
$\langle \overline{AB.pop} \rangle_{\overline{buffer}}$

Proof: *****************

$\langle 4 \rangle$2. $\land\ FirstFull$
$\land\ FirstFull'$
$\land\ \langle\, CB.pop \,\rangle_{Buffer}$
$\land\ \overline{Enabled \langle\, AB.pop \,\rangle_{buffer}}$
$\Rightarrow$
$CB.pop$

Proof: *******************

$\langle 4 \rangle 3.$ $\land$ $\mathit{FirstFull}$

$\land$ $\overline{\mathit{Enabled}\langle AB.pop \rangle_{buffer}}$

$\Rightarrow$

$\mathit{Enabled}\langle CB.pop \rangle_{Buffer}$

Proof: ****************

$\langle 4 \rangle 4.$ $\land$ $\Box[\land \lor \exists\, a : CB.push(a)$

$\lor\ CB.pop$

$\lor\ \exists\, k : CB.move(k)$

$\land\ \neg CB.pop]_{Buffer}$

$\land$ $WF_{Buffer}(CB.pop)$

$\land$ $WF_{Buffer}(\exists\, k : CB.move(k))$

$\land$ $\Diamond\Box\overline{\mathit{Enabled}\langle AB.pop \rangle_{buffer}}$

$\Rightarrow$

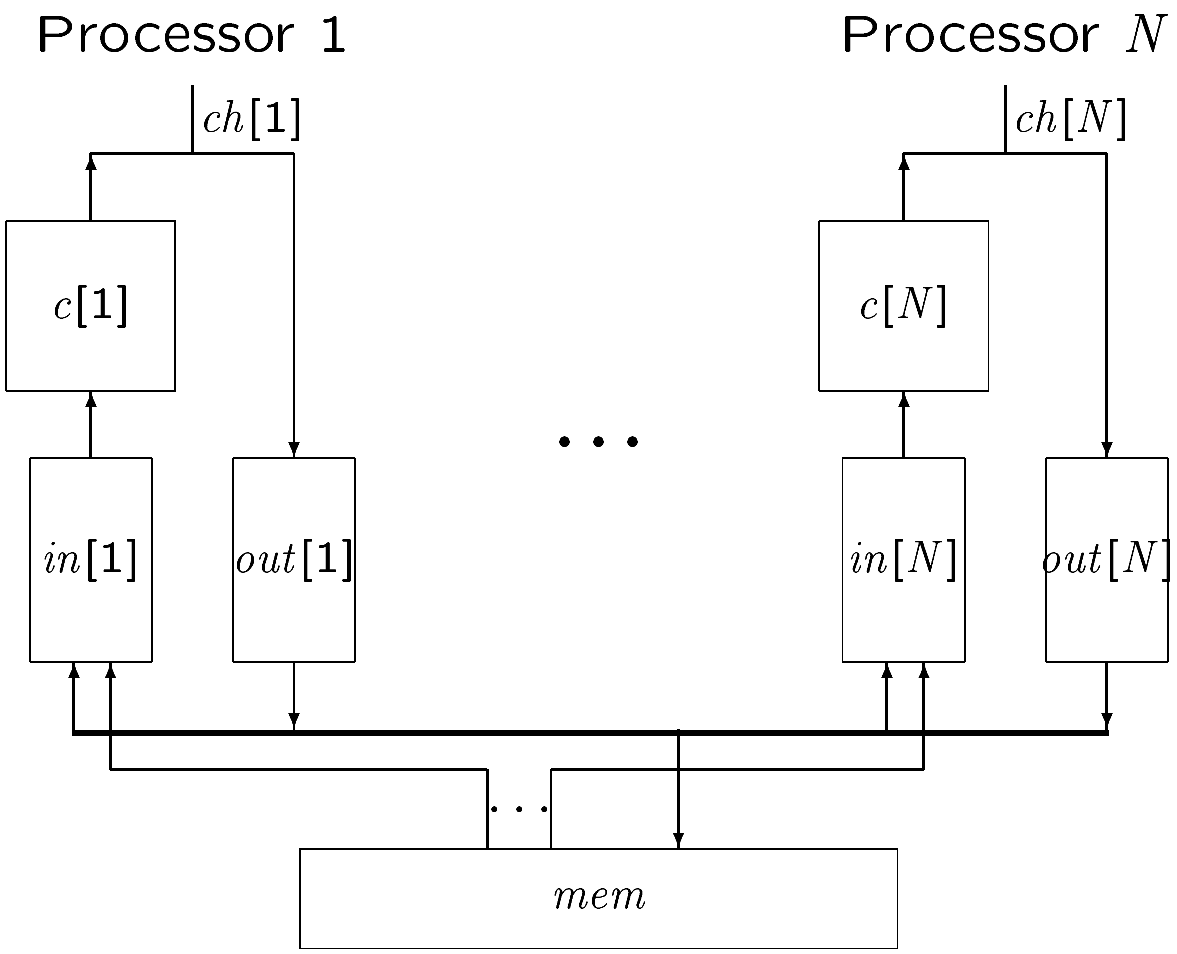
$\Diamond\Box \mathit{FirstFull}$

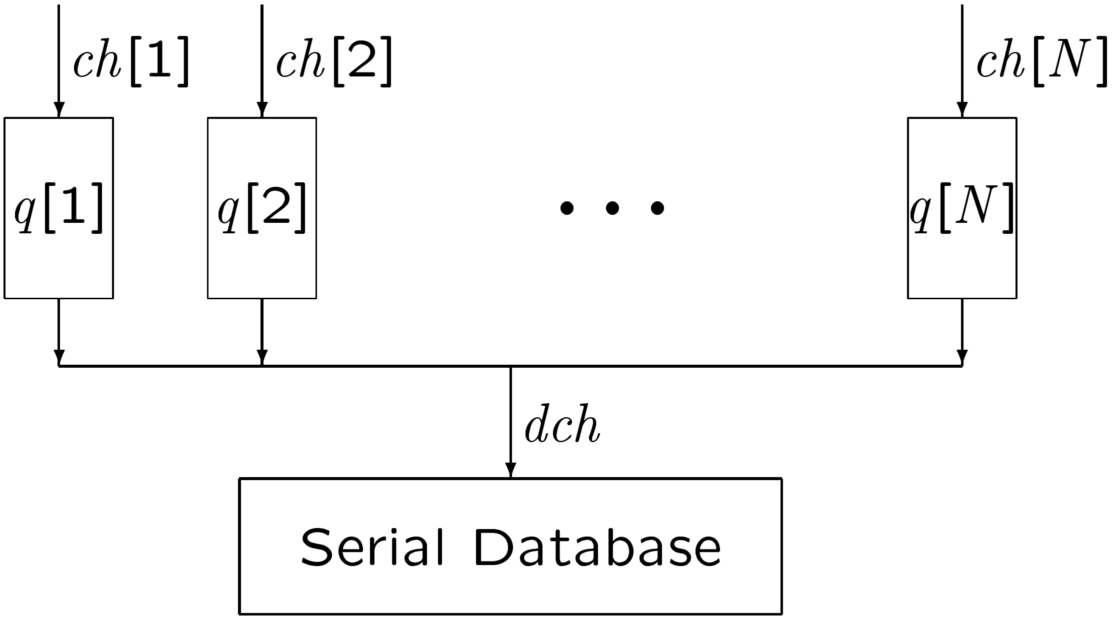
Proof: ***************

$\langle 4 \rangle 5.$ Q.E.D.

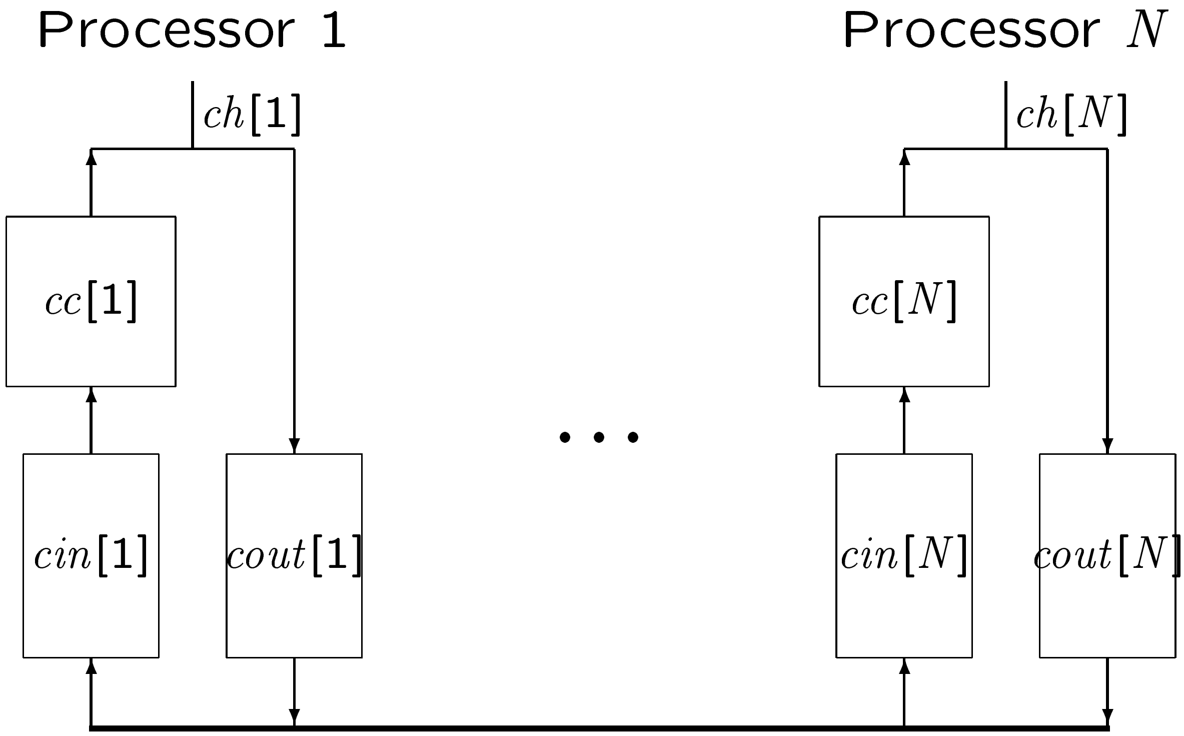
Proof: Follows from $\langle 4 \rangle 1$, $\langle 4 \rangle 2$, $\langle 4 \rangle 3$ and $\langle 4 \rangle 4$ using TLA Proof Rule WF2 with $\mathcal{N} \triangleq (\exists\, a : CB.push(a) \lor CB.pop \lor \exists\, k : CB.move(k))$, $f = \mathit{Buffer}$, $\mathcal{A} \triangleq \mathcal{B} \triangleq CB.pop$, $\mathcal{M} \triangleq AB.pop$, $g = \mathit{buffer}$, $P \triangleq \mathit{FirstFull}$, $\Box F \triangleq WF_{Buffer}(\exists\, k : CB.move(k))$. $\Box$

Section 7: The Lazy Caching Algorithm and Invariants


Processor 1
Processor N
$ch[1]$
$ch[N]$
$c[1]$
$c[N]$
$in[1]$
$out[1]$
$in[N]$
$out[N]$
mem



ch[1]
ch[2]
ch[N]
q[1]
q[2]
q[N]
dch
Serial Database



Processor 1
Processor N
$ch[1]$
$ch[N]$
$cc[1]$
$cc[N]$
$cin[1]$
$cout[1]$
$cin[N]$
$cout[N]$

module *LazyCache*

import *MemParams, ChannelInterface, Sequences, Naturals*

$\mathbf{parameters}\quad c, in, out, mem\ :\ \text{variable}$

$\perp\ \triangleq\ \text{choose}\ i\ :\ i \notin Data$

$$Restrict(f)\ \triangleq$$
$$\quad \{g \in [Addr \rightarrow Data \cup \{\perp\}]\ :$$
$$\qquad \forall\, a \in Addr\ :\ g[a] \in \{f[a], \perp\}\}$$

module *LazyCache: the Initial condition*

$$Init \triangleq \begin{array}{l} \land\ mem \in [Addr \rightarrow InitData] \\ \land\ c \in [Proc \rightarrow Restrict(mem)] \\ \land\ in = [i \in Proc \mapsto \langle\, \rangle] \\ \land\ out = [i \in Proc \mapsto \langle\, \rangle] \end{array}$$

module *LazyCache: Actions*

$$
\begin{aligned}
Read(i, d, a) \;\triangleq\;\; & \land\; out[i] = \langle\,\rangle \\
& \land\; \forall j \in 1 \,..\, Len(in[i]) : \\
& \qquad Len(in[i][j]) = 2 \\
& \land\; c[i][a] = d \\
& \land\; ChanOp(ch[i], \langle i, \text{“Rd”}, d, a\rangle) \\
& \land\; \forall j \in Proc \setminus \{i\} : \\
& \qquad \text{UNCHANGED } ch[j] \\
& \land\; \text{UNCHANGED } \langle c, in, out, mem\rangle
\end{aligned}
$$

$$
\begin{aligned}
Write(i, d, a) \;\triangleq\;\; & \land\; ChanOp(ch[i], \langle i, \text{“Wr”}, d, a\rangle) \\
& \land\; out' = [out \text{ EXCEPT} \\
& \qquad\qquad !\,[i] = out[i] \circ \langle\langle d, a\rangle\rangle] \\
& \land\; \forall j \in Proc \setminus \{i\} : \\
& \qquad \text{UNCHANGED } ch[j] \\
& \land\; \text{UNCHANGED } \langle c, in, mem\rangle
\end{aligned}
$$

module *LazyCache: Actions*

$$
\begin{aligned}
&MemWrite(i) \triangleq \\
&\quad \land\ out[i] \neq \langle\,\rangle \\
&\quad \land\ out' = [out \text{ except } ![i] = Tail(out[i])] \\
&\quad \land\ in' = [j \in Proc \mapsto \\
&\qquad\quad in[j] \circ \textbf{if } j = i \ \textbf{ then } \langle Head(out[i]) \circ \langle \text{"}*\text{"} \rangle\rangle \\
&\qquad\qquad\qquad\qquad\quad \textbf{ else } \ \langle Head(out[i])\rangle \qquad] \\
&\quad \land\ mem' = [mem \text{ except} \\
&\qquad\qquad ![Head(out[i])[2]] = Head(out[i])[1]\,] \\
&\quad \land\ \text{unchanged } \langle c, ch\rangle
\end{aligned}
$$

$$
\begin{aligned}
&CacheUpdate(i) \triangleq \\
&\qquad \land\ in[i] \neq \langle\,\rangle \\
&\qquad \land\ in' = [in \text{ except } ![i] = Tail(in[i])] \\
&\qquad \land\ c' = [c \text{ except} \\
&\qquad\qquad\qquad ![i][Head(in[i])[2]] = Head(in[i])[1]\,] \\
&\qquad \land\ \text{unchanged } \langle out, ch, mem\rangle
\end{aligned}
$$

module *LazyCache: Actions*

$$MemRead(i) \triangleq$$
$$\land\ \exists\, a \in Addr :$$
$$in' = [in \text{ except } ![i] = in[i] \circ \langle \langle mem[a], a \rangle \rangle]$$
$$\land\ \text{unchanged } \langle out, c, mem, ch \rangle$$

$$CacheInval(i) \triangleq$$
$$\land\ c' \in \{[c \text{ except } ![i] \in Restrict(c[i])]\}$$
$$\land\ \text{unchanged } \langle in, out, mem, ch \rangle$$

module *LazyCache: Actions*

$$
\begin{aligned}
&Next(i) \triangleq \\
&\qquad \lor\ \exists\, d \in Data, a \in Addr\ : \\
&\qquad\qquad \lor\ Read(i, d, a) \\
&\qquad\qquad \lor\ Write(i, d, a) \\
&\qquad \lor\ MemWrite(i) \\
&\qquad \lor\ CacheUpdate(i) \\
&\qquad \lor\ MemRead(i) \\
&\qquad \lor\ CacheInval(i)
\end{aligned}
$$

module *LazyCache: Specification*

$$vars \triangleq \langle c, in, out, mem, ch \rangle$$

$$\begin{aligned} Spec \triangleq\ & \land\ Init \\ & \land\ \Box[\exists\, i \in Proc : Next(i)]_{vars} \\ & \land\ \forall\, i \in Proc : \\ & \qquad \lor\ \mathsf{WF}_{vars}(CacheUpdate(i)) \\ & \qquad \lor\ \mathsf{WF}_{vars}(MemWrite(i)) \end{aligned}$$

module *CCache: Definitions*

import *MemParams, ChannelInterface, Sequences, Naturals*

parameters *cc, cin, cout :* variable

module *CCache: Init*

$$
\begin{aligned}
Init \;\triangleq\; & \land\; cc \in [Proc \rightarrow [Addr \rightarrow InitData]] \\
& \land\; \forall\, i, j \in Proc \;:\; cc[i] = cc[j] \\
& \land\; cin = [i \in Proc \mapsto \langle\,\rangle] \\
& \land\; cout = [i \in Proc \mapsto \langle\,\rangle]
\end{aligned}
$$

module *CCache: Actions*

$$Read(i, d, a) \triangleq$$
$$\begin{aligned}
&\land\ cout[i] = \langle\,\rangle \\
&\land\ \forall j \in (1 \,..\, Len(cin[i]))\ :\ cin[i][j][1] \neq i \\
&\land\ cc[i][a] = d \\
&\land\ ChanOp(ch[i], \langle i, \text{“Rd”}, d, a\rangle) \\
&\land\ \forall j \in Proc \setminus \{i\}\ :\ \textsf{unchanged}\ ch[j] \\
&\land\ \textsf{unchanged}\ \langle cc, cin, cout\rangle
\end{aligned}$$

$$Write(i, d, a) \triangleq$$
$$\begin{aligned}
&\land\ ChanOp(ch[i], \langle i, \text{“Wr”}, d, a\rangle) \\
&\land\ cout' = [cout\ \textsf{except} \\
&\qquad\qquad\quad !\,[i] = cout[i] \circ \langle\langle i, \text{“Wr”}, d, a\rangle\rangle] \\
&\land\ \forall j \in Proc \setminus \{i\}\ : \\
&\qquad \textsf{unchanged}\ ch[j] \\
&\land\ \textsf{unchanged}\ \langle cc, cin\rangle
\end{aligned}$$

module *CCache: Actions*

$$
\begin{aligned}
&MemWrite(i) \triangleq \\
&\quad \land\ cout[i] \neq \langle\,\rangle \\
&\quad \land\ cout' = [cout \text{ except } ![i] = Tail(cout[i])] \\
&\quad \land\ cin' = [j \in Proc \mapsto cin[j] \circ \langle Head(cout[i])\rangle] \\
&\quad \land\ \text{unchanged } \langle cc, ch\rangle
\end{aligned}
$$

$$
\begin{aligned}
&CacheUpdate(i) \triangleq \\
&\qquad \land\ cin[i] \neq \langle\,\rangle \\
&\qquad \land\ cin' = [cin \text{ except } ![i] = Tail(cin[i])] \\
&\qquad \land\ cc' = [cc \text{ except} \\
&\qquad\qquad ![i][Head(cin[i])[4]] = Head(cin[i])[3]] \\
&\qquad \land\ \text{unchanged } \langle cout, ch\rangle
\end{aligned}
$$

module *CCache: Actions*

$$
\begin{array}{l}
Next(i) \;\triangleq \\
\quad \lor\ \exists\, d \in Data, a \in Addr\ : \\
\qquad\quad \lor\ Read(i, d, a) \\
\qquad\quad \lor\ Write(i, d, a) \\
\quad \lor\ MemWrite(i) \\
\quad \lor\ CacheUpdate(i)
\end{array}
$$

module *CCache: Specification*

$$vars \triangleq \langle cc, cin, cout, ch \rangle$$

$$Spec \triangleq$$

$$\begin{array}{l} \land\ Init \\ \land\ \Box[\exists\, i \in Proc\ :\ Next(i)]_{vars} \\ \land\ \forall\, i \in Proc\ : \\ \qquad \land\ \mathsf{WF}_{vars}(CacheUpdate(i)) \\ \qquad \land\ \mathsf{WF}_{vars}(MemWrite(i)) \end{array}$$

A predicate needed to state the invariant $CCInv.3(p)$

Let: $\forall\, i \in Proc \;:\; NotRefresh_i(j) \;\triangleq$

$$\begin{aligned} &\wedge\; j \in 1 \,..\, Len(lcin[i]) \\ &\wedge\; lcin[i][j][2] \neq \text{“Rd”} \end{aligned}$$

The first two components of the invariant

Let: $CCInv.1(p) \;\triangleq\; Len(lcin[p]) = Len(in[p])$

$CCInv.2(p) \;\triangleq\; c[p] \in Restrict(vc[p])$

Either there's no $MemWrite$ in the queue..

Let: $NoMemWrite(a, p) \;\triangleq$

$$\begin{array}{l}
\land\ p \in Proc \\
\land\ vc[p][a] = mem[a] \\
\land\ \forall\, k \in 1 \,..\, Len(in[p]) : \\
\qquad \lor\ in[p][k][2] \neq a \\
\qquad \lor\ in[p][k][1] = mem[a]
\end{array}$$

.. or the last $MemWrite$ is the value of mem

Let: $LastIsMem(a, p) \triangleq$

$$
\begin{array}{l}
\land\ p \in Proc \\
\land\ \exists\, k \in 1 \,..\, Len(in[p]) : \\
\qquad \land\ NotRefresh_p(k) \\
\qquad \land\ in[p][k][1] = mem[a] \\
\qquad \land\ in[p][k][2] = a \\
\qquad \land\ \forall\, j \in k+1 \,..\, Len(in[p]) : \\
\qquad\qquad \lor\ in[p][j][2] \neq a \\
\qquad\qquad \lor\ in[p][j][1] = mem[a]
\end{array}
$$

.. and the first $MemWrite$ is the value of vc

Let: $FirstIsVC(a, p) \triangleq$

$$
\begin{aligned}
&\land\ p \in Proc \\
&\land\ \exists\, k \in 1 \,..\, Len(in[p]) : \\
&\qquad \land\ NotRefresh_p(k) \\
&\qquad \land\ in[p][k][2] = a \\
&\qquad \land\ \forall\, j \in 1 \,..\, k-1 : \\
&\qquad\qquad \lor\ in[p][j][2] \neq a \\
&\qquad\qquad \lor\ in[p][j][1] = vc[p][a]
\end{aligned}
$$

.. and between $MemWrites$ are just repeats of the last one

Let: $MemWritesFromTo(m, n, p) \triangleq$

$$
\begin{aligned}
&\land\ p \in Proc \\
&\land\ m \in 1 \,..\, Len(in[p]) \\
&\land\ n \in 1 \,..\, Len(in[p]) \\
&\land\ m < n \\
&\land\ NotRefresh_p(m) \\
&\land\ NotRefresh_p(n) \\
&\land\ \forall j \in m + 1 \,..\, n - 1 \ : \ \neg NotRefresh_p(j) \\
&\quad \Rightarrow \\
&\qquad \forall j \in m + 1 \,..\, n - 1 \ : \\
&\qquad\quad \lor\ in[p][j][2] \neq in[p][m][2] \\
&\qquad\quad \lor\ in[p][j][1] = in[p][m][1]
\end{aligned}
$$

...so here's the third component of the invariant

$$\text{Let: } BetweenMemWrites(p) \triangleq$$
$$\quad \forall m, n \in 1 \,..\, Len(in[p]) : MemWritesFromTo(m, n, p)$$

$$CCInv.3(p) \triangleq$$
$$\quad \land\ BetweenMemWrites(p)$$
$$\quad \land\ \forall a \in Addr : \lor\ NoMemWrite(a, p)$$
$$\qquad\qquad \lor\ \land\ LastIsMem(a, p) \quad \land\ FirstIsVC(a, p)$$

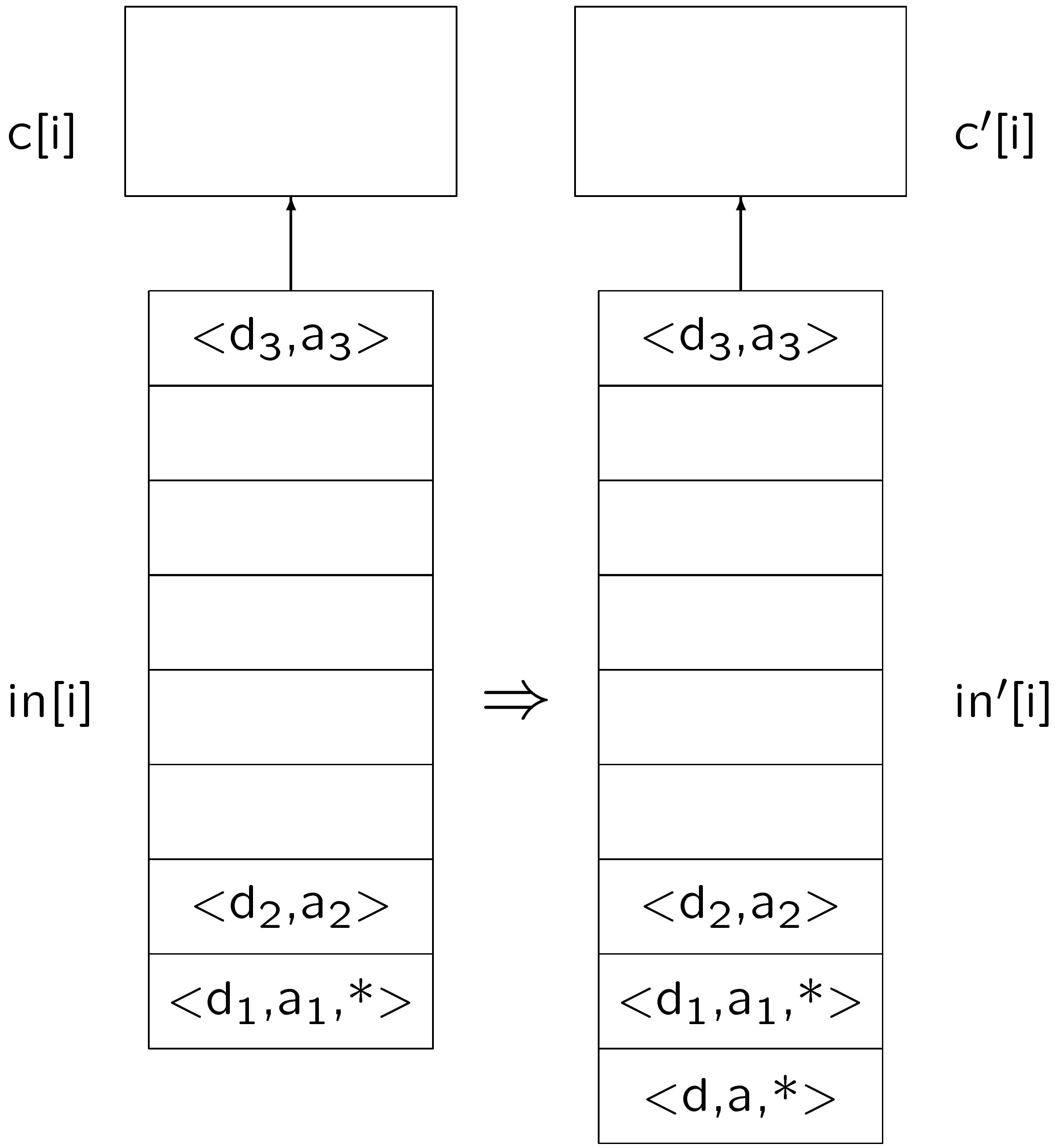
...and the invariant itself

Let: $$CCInv(p) \triangleq \begin{array}{l} \land\ CCInv.1(p) \\ \land\ CCInv.2(p) \\ \land\ CCInv.3(p) \end{array}$$

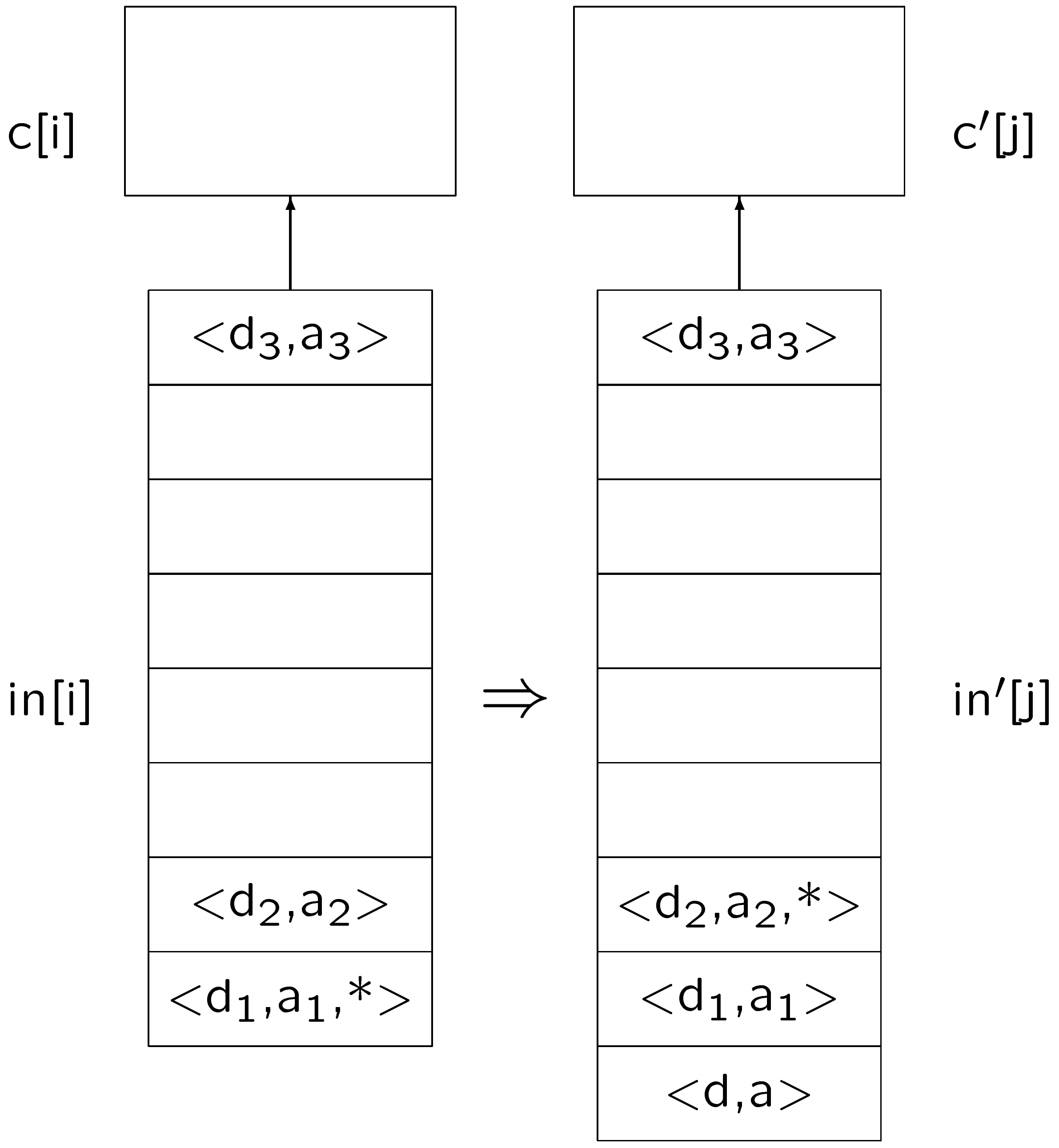
$$\begin{array}{lcl} CCInv.1 & \triangleq & \forall\, p \in Proc\ :\ CCInv.1(p) \\ CCInv.2 & \triangleq & \forall\, p \in Proc\ :\ CCInv.2(p) \\ CCInv.3 & \triangleq & \forall\, p \in Proc\ :\ CCInv.3(p) \end{array}$$

$$CCInv \triangleq CCInv.1 \land CCInv.2 \land CCInv.3$$

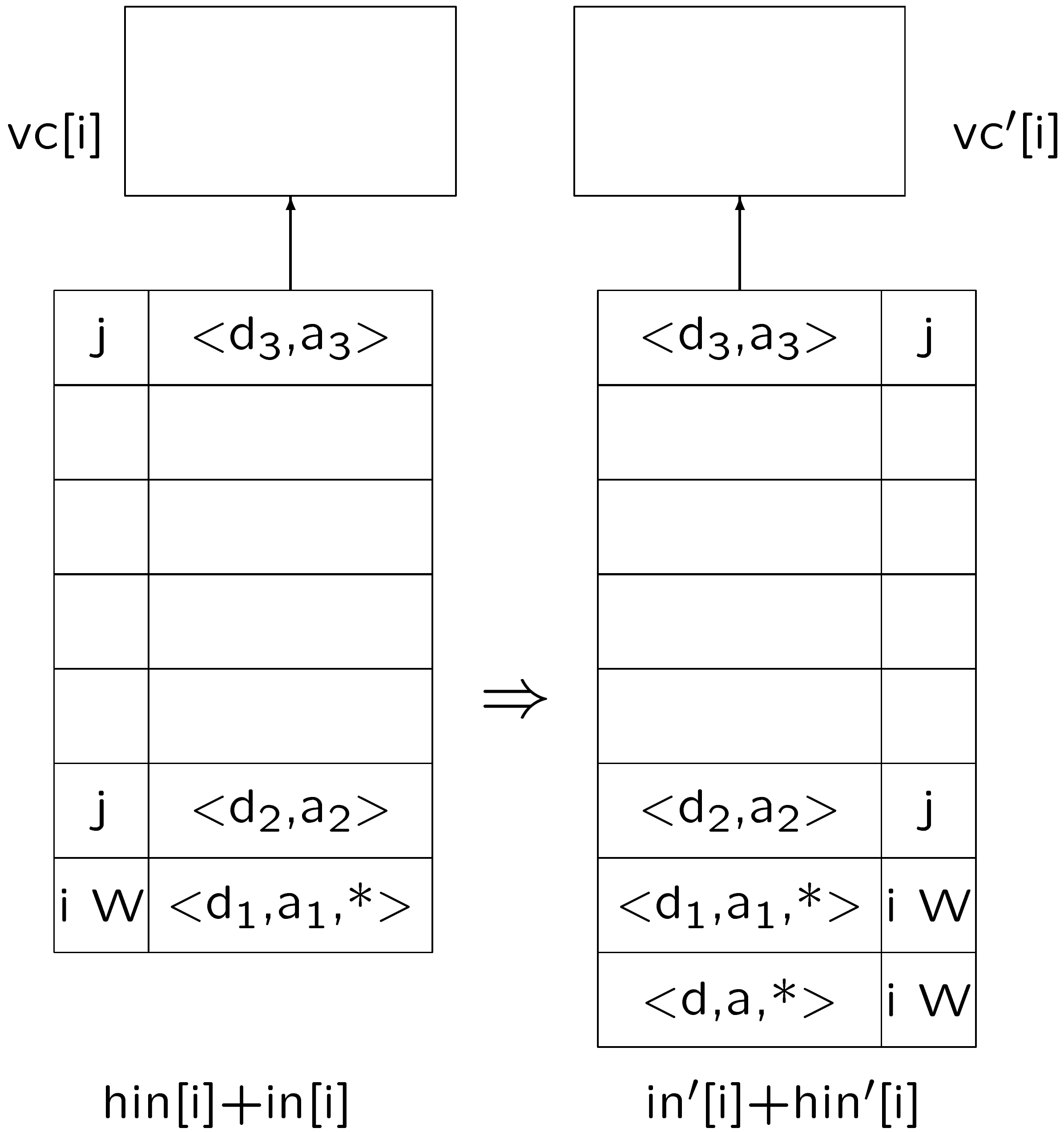
Section 8: The Invariants for Lazy Caching



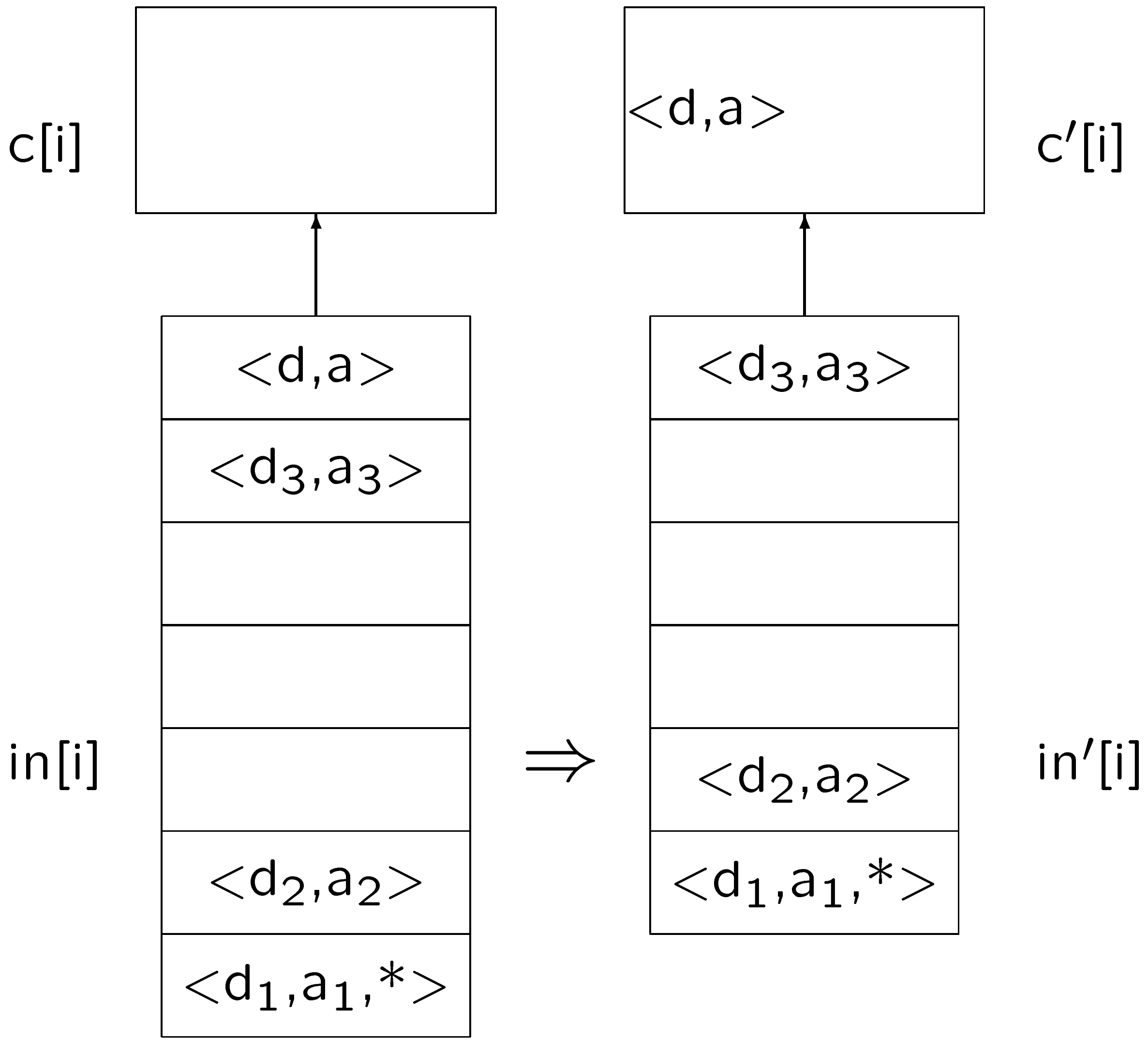
MemWrite(i) (for in[i])



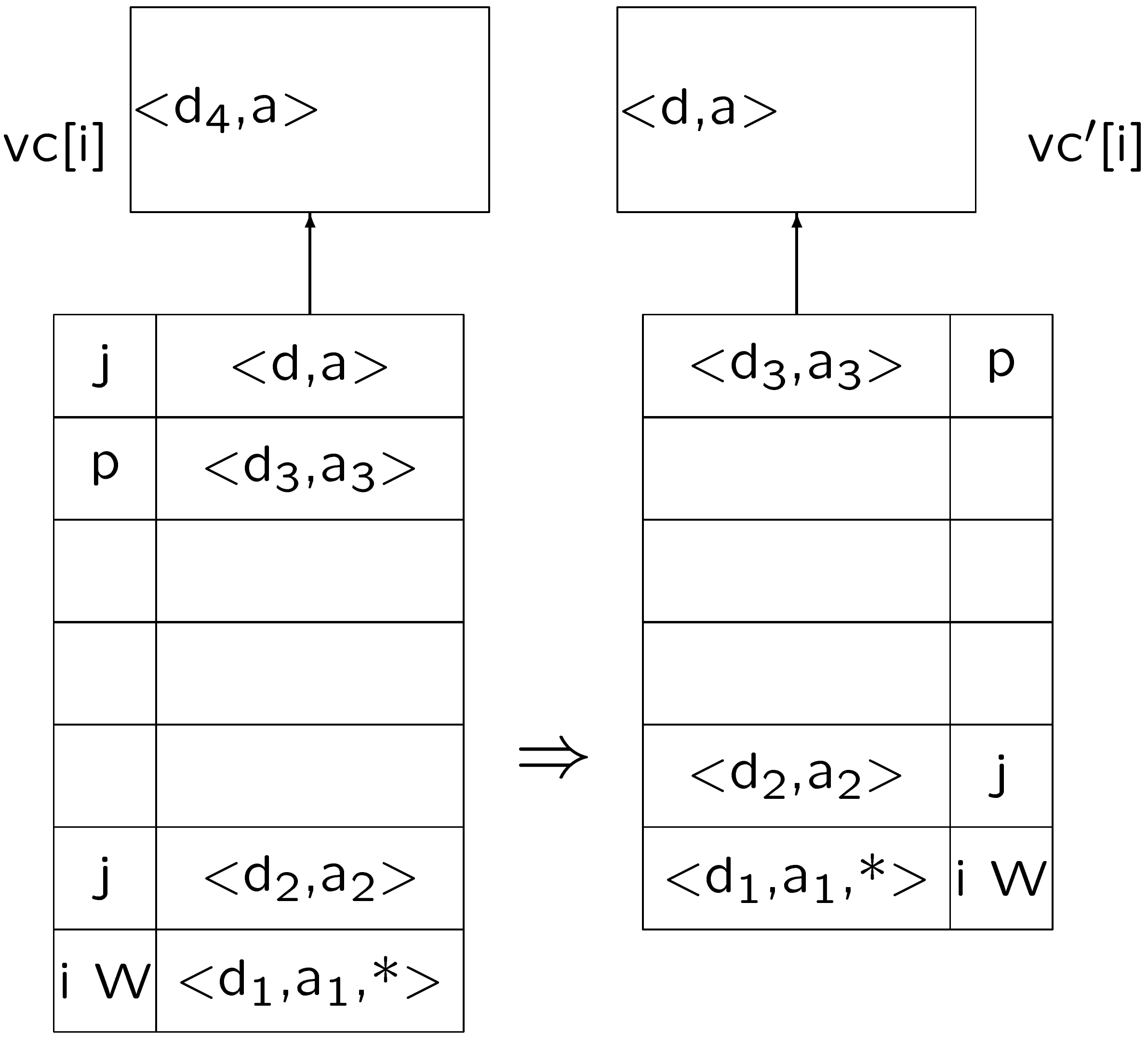
MemWrite(i) (for in[j])


vc[i]
vc′[i]
j
<d3,a3>
j
<d2,a2>
i W
<d1,a1,*>
<d3,a3>
j
<d2,a2>
j
<d1,a1,*>
i W
<d,a,*>
i W
hin[i]+in[i]
in′[i]+hin′[i]

HMemWrite(i)

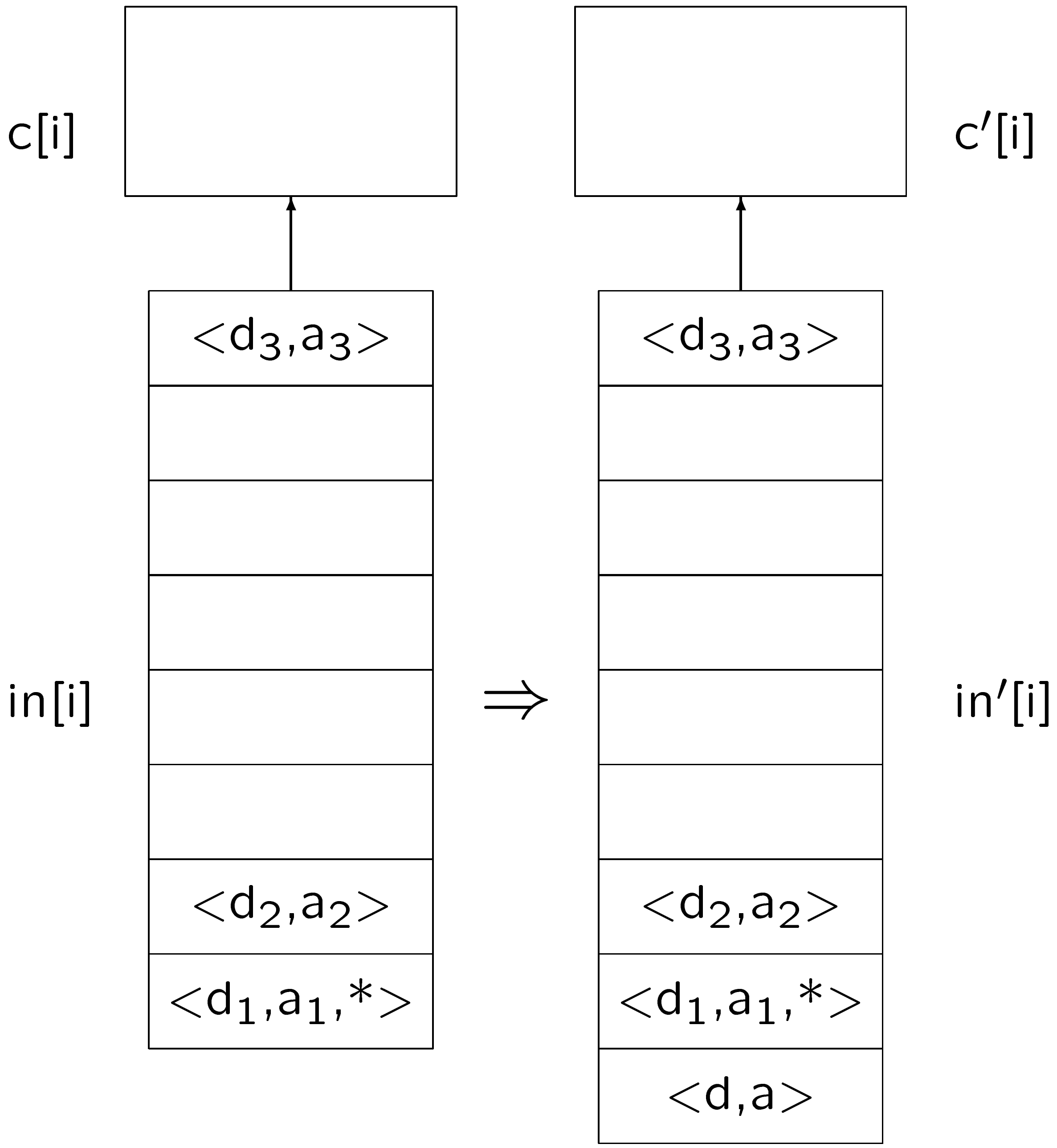


CacheUpdate(i)

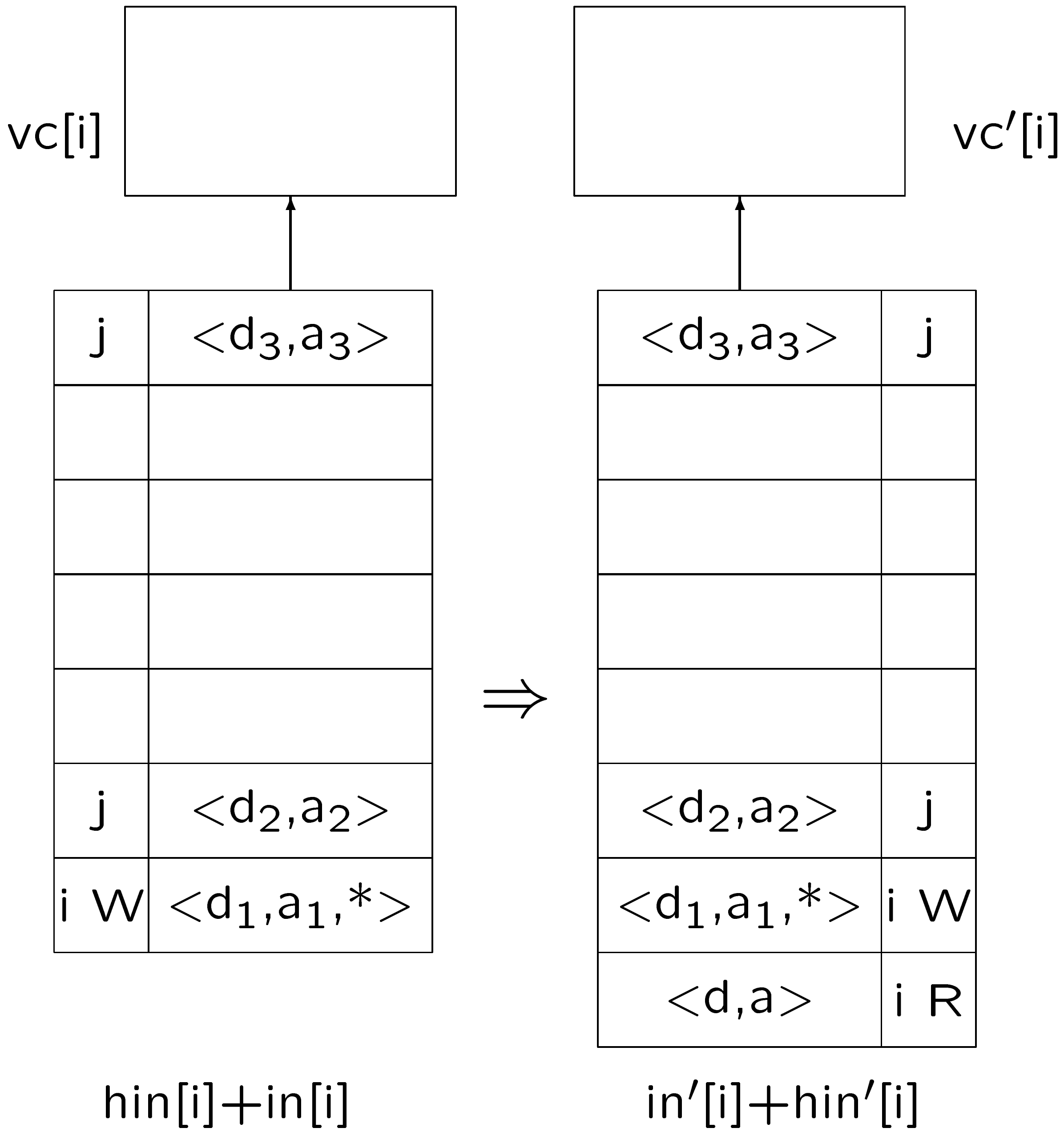


hin[i]+in[i] in′[i]+hin′[i]

HCacheUpdate(i)

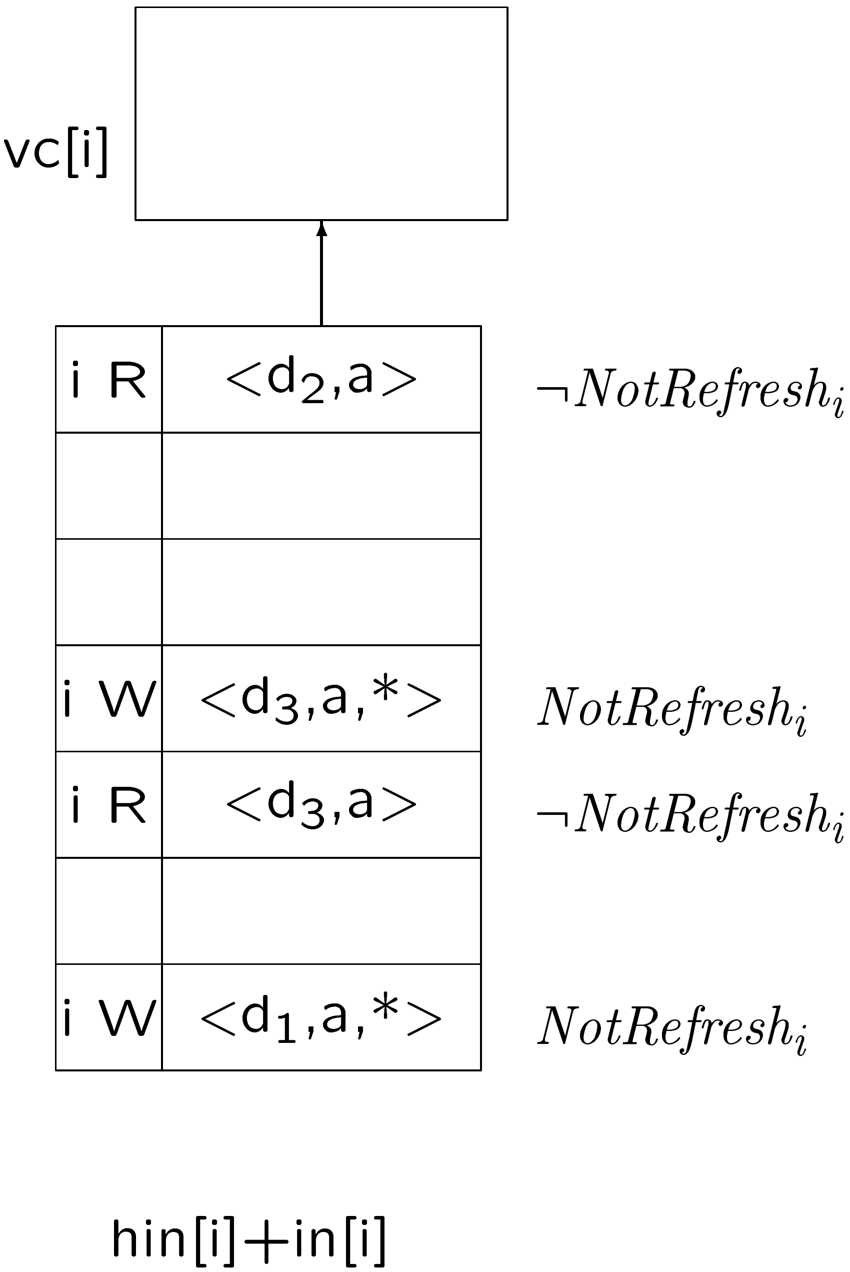


MemRead(i)

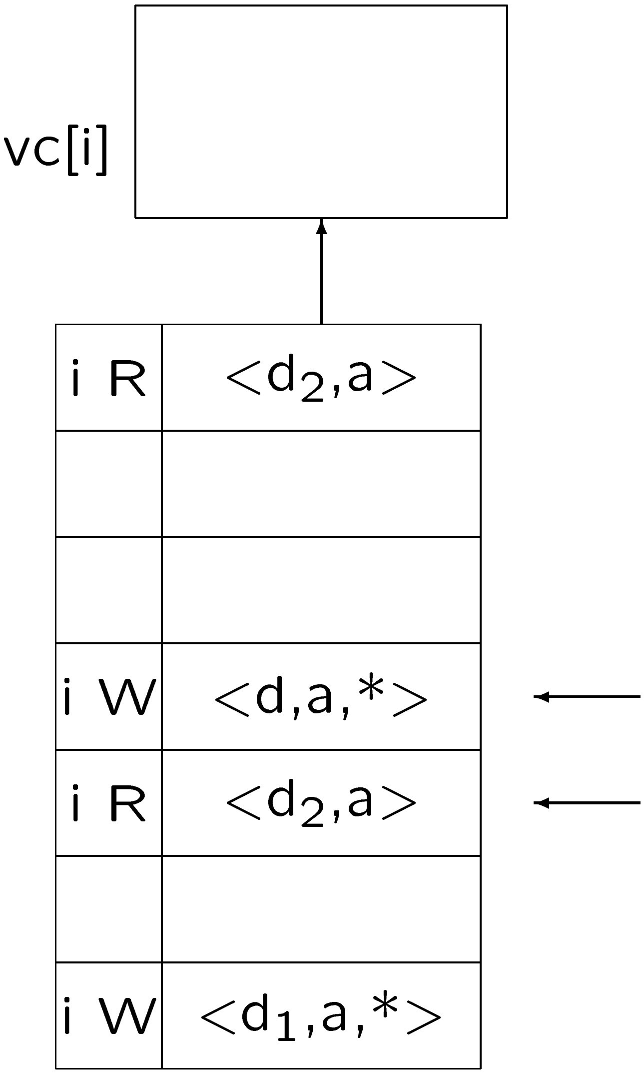


vc[i]
vc′[i]
j
$<d_3,a_3>$
j
$<d_2,a_2>$
i W
$<d_1,a_1,*>$
$\Rightarrow$
$<d_3,a_3>$
j
$<d_2,a_2>$
j
$<d_1,a_1,*>$
i W
$<d,a>$
i R
hin[i]+in[i]
in′[i]+hin′[i]

HMemRead(i)

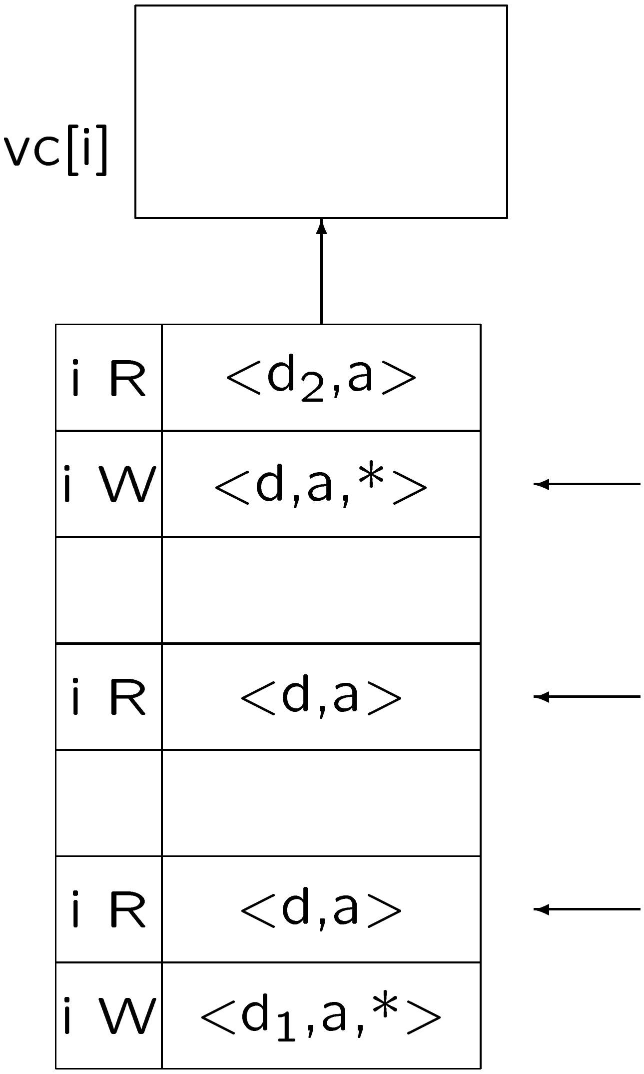


Illustrating $NotRefresh_i(k)$



hin[i]+in[i]

Can this happen?



hin[i]+in[i]

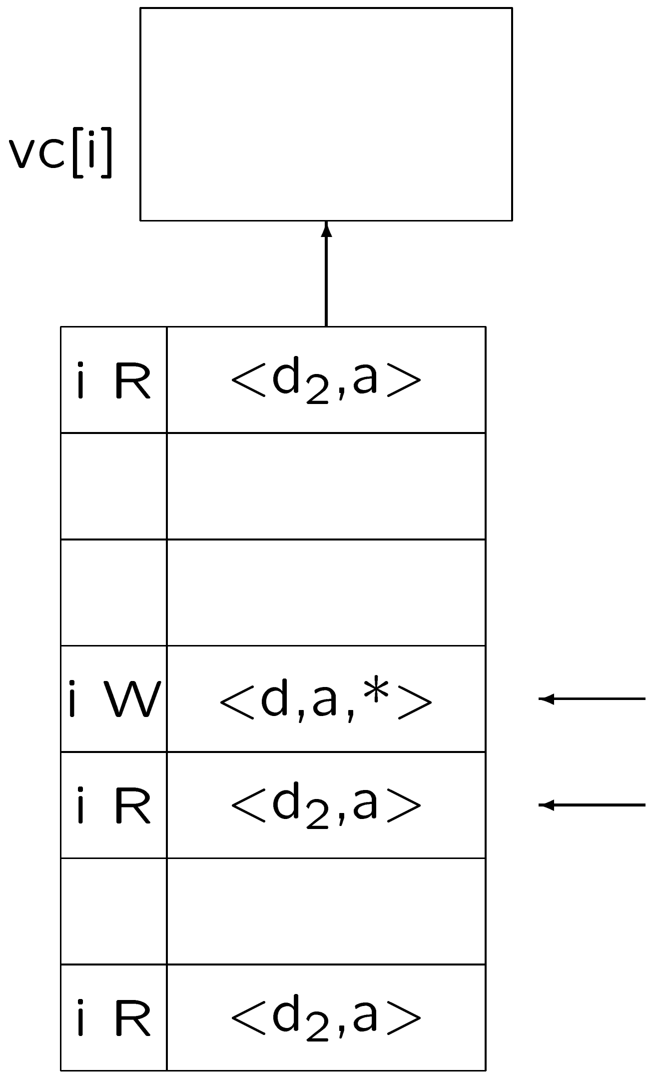
$$MemWritesFromTo(2, 7, i)$$

Hence the invariant

$$BetweenMemWrites(i) \triangleq$$
$$\forall m, n \in 1 \,..\, Len(in[i]) : MemWritesFromTo(m, n, i)$$

where

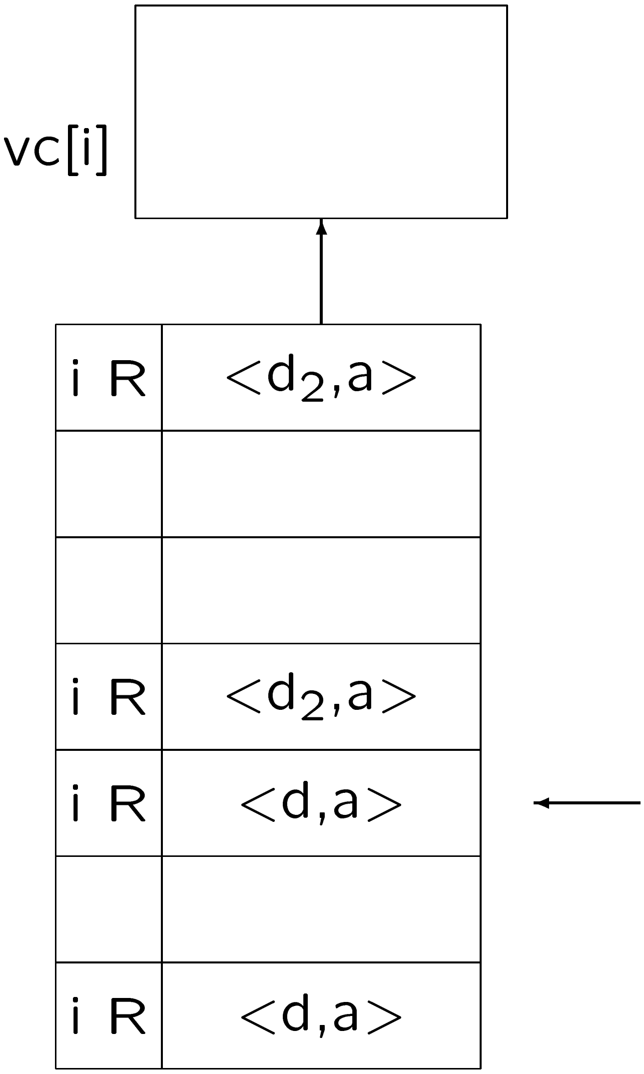
$$
\begin{array}{l}
MemWritesFromTo(m, n, i) \triangleq \\
\quad \land\ i \in Proc \\
\quad \land\ m \in 1 \,..\, Len(in[i]) \\
\quad \land\ n \in 1 \,..\, Len(in[i]) \\
\quad \land\ m < n \\
\quad \land\ NotRefresh_i(m) \\
\quad \land\ NotRefresh_i(m) \\
\quad \land\ \forall j \in m+1 \,..\, n-1 : \neg NotRefresh_i(j) \\
\quad \Rightarrow \forall j \in m+1 \,..\, n-1 : \\
\qquad\qquad \lor\ in[i][j][2] \neq in[i][m][2] \\
\qquad\qquad \lor\ in[i][j][1] = in[i][m][1]
\end{array}
$$



hin[i]+in[i]

But what about this??

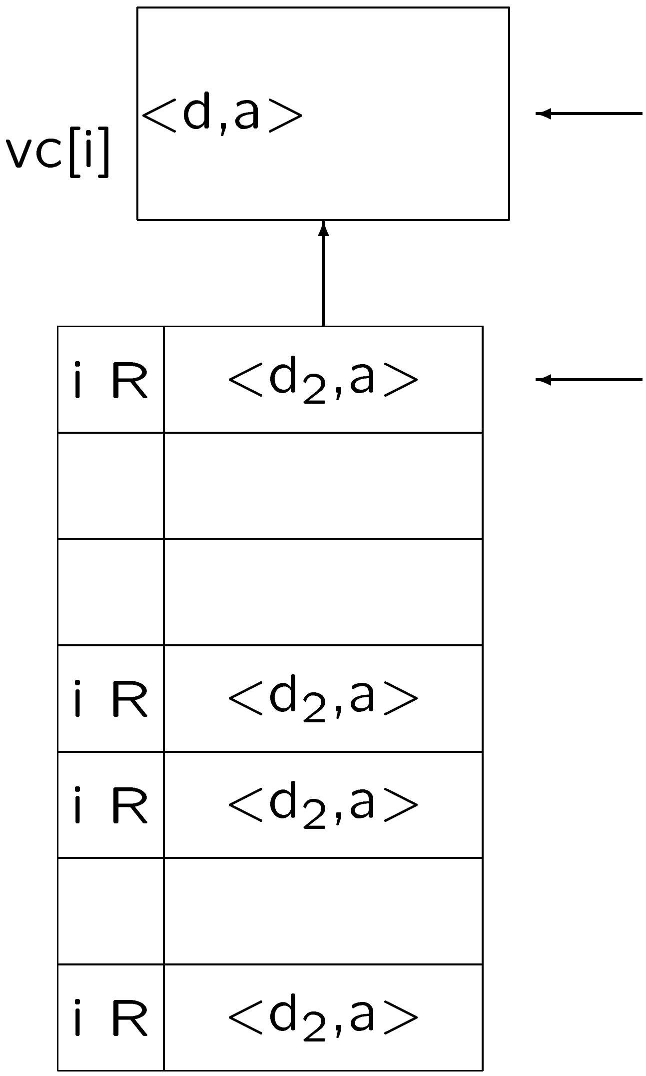
So every refresh of “a” that follows the last write of “a” had better contain the same data value as that write.



hin[i]+in[i]

Or this??

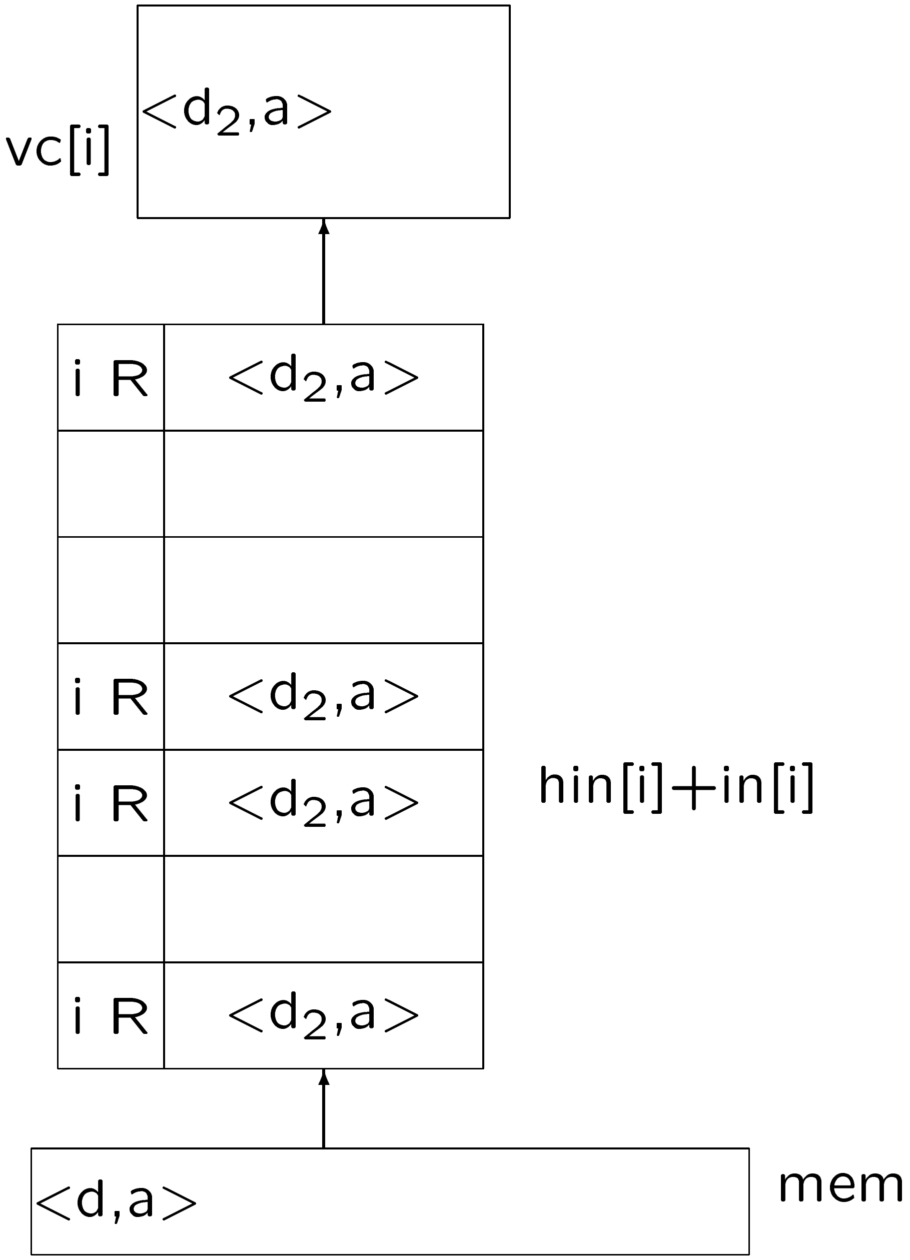
So if there are only refreshes, they had better all have the same value. What value?



hin[i]+in[i]

Ummmm, any old value?

The same value that has already been written in vc[i][a].



But aren't we forgetting something?

Final invariant:

- Refreshes between adjacent writes must be the same as the first one;
- When there are only refreshes in in[i], they must be the same as in vc and in mem;
- Otherwise, it's vc until the first Write, and mem beyond the last one

$$
\begin{aligned}
&NoMemWrite(a, p) \;\triangleq \\
&\qquad \land\; p \in Proc \\
&\qquad \land\; vc[p][a] = mem[a] \\
&\qquad \land\; \forall\, k \in 1 \,..\, Len(in[p]) \;: \\
&\qquad\qquad \lor\; in[p][k][2] \neq a \\
&\qquad\qquad \lor\; in[p][k][1] = mem[a]
\end{aligned}
$$

$$
\begin{array}{l}
LastIsMem(a, p) \;\triangleq \\
\quad \land\; p \in Proc \\
\quad \land\; \exists\, k \in 1 \,..\, Len(in[p]) : \\
\qquad\quad \land\; NotRefresh_p(k) \\
\qquad\quad \land\; in[p][k][1] = mem[a] \\
\qquad\quad \land\; in[p][k][2] = a \\
\qquad\quad \land\; \forall\, j \in k + 1 \,..\, Len(in[p]) : \\
\qquad\qquad\quad \lor\; in[p][j][2] \neq a \\
\qquad\qquad\quad \lor\; in[p][j][1] = mem[a]
\end{array}
$$

$$
\begin{array}{l}
FirstIsVC(a, p) \ \triangleq \\
\quad \land\ p \in Proc \\
\quad \land\ \exists\, k \in 1 \,..\, Len(in[p]) : \\
\qquad\quad \land\ NotRefresh_p(k) \\
\qquad\quad \land\ in[p][k][2] = a \\
\qquad\quad \land\ \forall\, j \in 1 \,..\, k-1 : \\
\qquad\qquad\quad \lor\ in[p][j][2] \neq a \\
\qquad\qquad\quad \lor\ in[p][j][1] = vc[p][a]
\end{array}
$$

$$
\begin{array}{l}
CCInv.3(p) \;\triangleq \\
\quad \land\; BetweenMemWrites(p) \\
\quad \land\; \forall\, a \in Addr\; : \\
\qquad\qquad \lor\; NoMemWrite(a, p) \\
\qquad\qquad \lor\; \land\; LastIsMem(a, p) \\
\qquad\qquad\quad\;\; \land\; FirstIsVC(a, p)
\end{array}
$$

That's not all:

$$CCInv.1(p) \triangleq Len(hin[p]) = Len(in[p])$$

$$CCInv.2(p) \triangleq c[p] \in Restrict(vc[p])$$

Finally,

$$CCInv \triangleq \forall p \in Proc : \begin{array}{l} \land\ CCInv.1(p) \\ \land\ CCInv.2(p) \\ \land\ CCInv.3(p) \end{array}$$

Here's $BetweenMemWrites$

$$
\begin{array}{l}
\land\ i \in Proc \land HMemWrite(i) \\
\land\ p \in Proc \\
\land\ CCInv.1(p) \\
\land\ BetweenMemWrites(p) \\
\land\ \forall\, a \in Addr\ :\ \lor\ LastIsMem(a, p) \\
\qquad\qquad\qquad\quad\ \lor\ NoMemWrite(a, p) \\
\quad \Rightarrow BetweenMemWrites'(p)
\end{array}
$$

Here's $FirstIsVC$

$$
\begin{array}{l}
\land\ i \in Proc \\
\land\ HMemWrite(i) \\
\land\ p \in Proc \\
\land\ CCInv.1(p) \\
\land\ a \in Addr \\
\land\ FirstIsVC(a, p) \\
\qquad \Rightarrow FirstIsVC'(a, p)
\end{array}
$$

With $NoMemWrite(Head(out[i])[2], p)$, things are less simple:

$$\begin{array}{l}
\land\ i \in Proc \land HMemWrite(i) \\
\land\ p \in Proc \\
\land\ CCInv.1(p) \\
\land\ NoMemWrite(Head(out[i])[2], p) \\
\Rightarrow \lor\ NoMemWrite'(Head(out[i])[2], p) \\
\quad\ \ \lor\ FirstIsVC'(Head(out[i])[2], p)
\end{array}$$

But $\mathit{LastIsMem}$ is easier:

$$
\begin{array}{l}
\land \ i \in \mathit{Proc} \land \mathit{HMemWrite}(i) \\
\land \ p \in \mathit{Proc} \\
\Rightarrow \quad \mathit{LastIsMem}'(\mathit{Head}(\mathit{out}[i])[2], p)
\end{array}
$$

so we can conclude

$$
\begin{array}{l}
\land \ i \in \mathit{Proc} \land \mathit{HMemWrite}(i) \\
\land \ p \in \mathit{Proc} \\
\land \ \mathit{CCInv}.1(p) \\
\land \ \mathit{NoMemWrite}(\mathit{Head}(\mathit{out}[i])[2], p) \\
\Rightarrow \lor \ \mathit{NoMemWrite}'(\mathit{Head}(\mathit{out}[i])[2], p) \\
\quad\ \ \lor \land \ \mathit{LastIsMem}'(\mathit{Head}(\mathit{out}[i])[2], p) \\
\quad\quad\ \ \ \land \ \mathit{FirstIsVC}'(\mathit{Head}(\mathit{out}[i])[2], p)
\end{array}
$$

The proof for the other values of a for $NoMemWrite$ is straightforward.

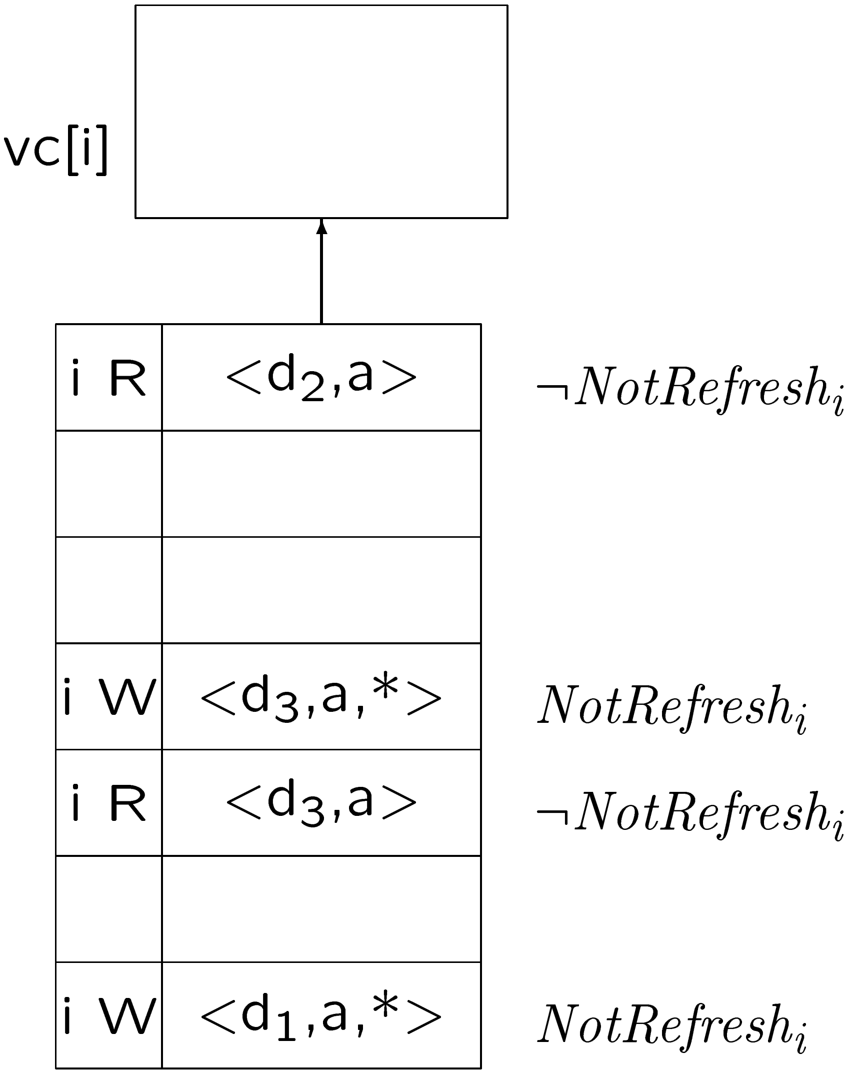
There's a symmetric problem with $FirstIsVC$ and $HCacheUpdate$.

In the buffer example, no invariant was needed, but liveness is non-trivial (we haven't seen it yet)

LazyCache is interesting because the safety proof is non-trivial.... even though liveness is trivial.

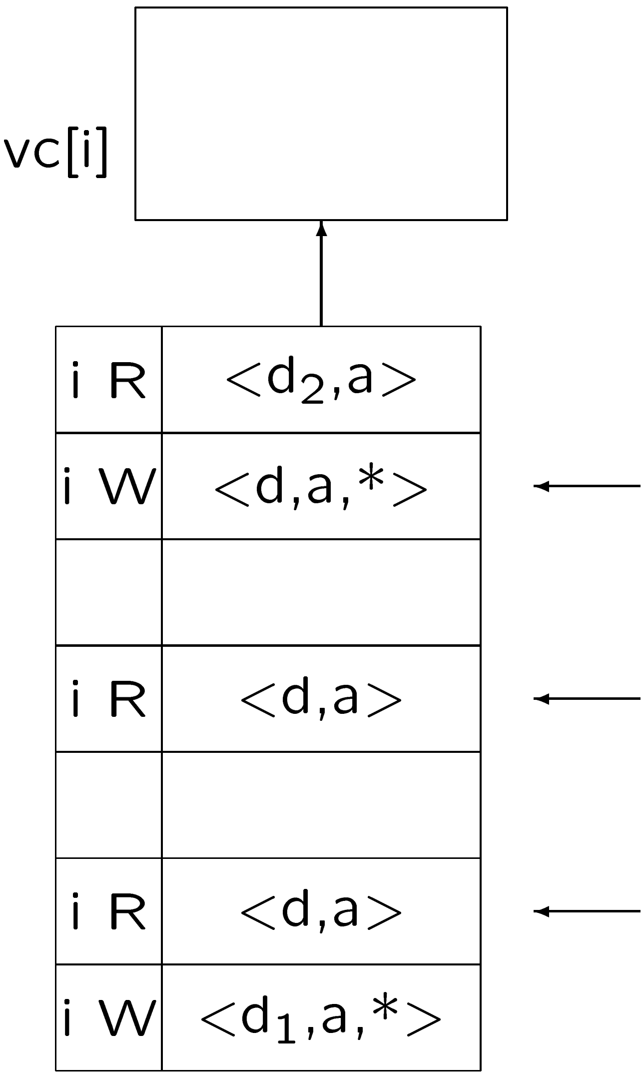
How does it all work?

Section 9: How the Lazy Caching Invariants are Proved



hin[i]+in[i]

Illustrating $NotRefresh_i(k)$



hin[i]+in[i]

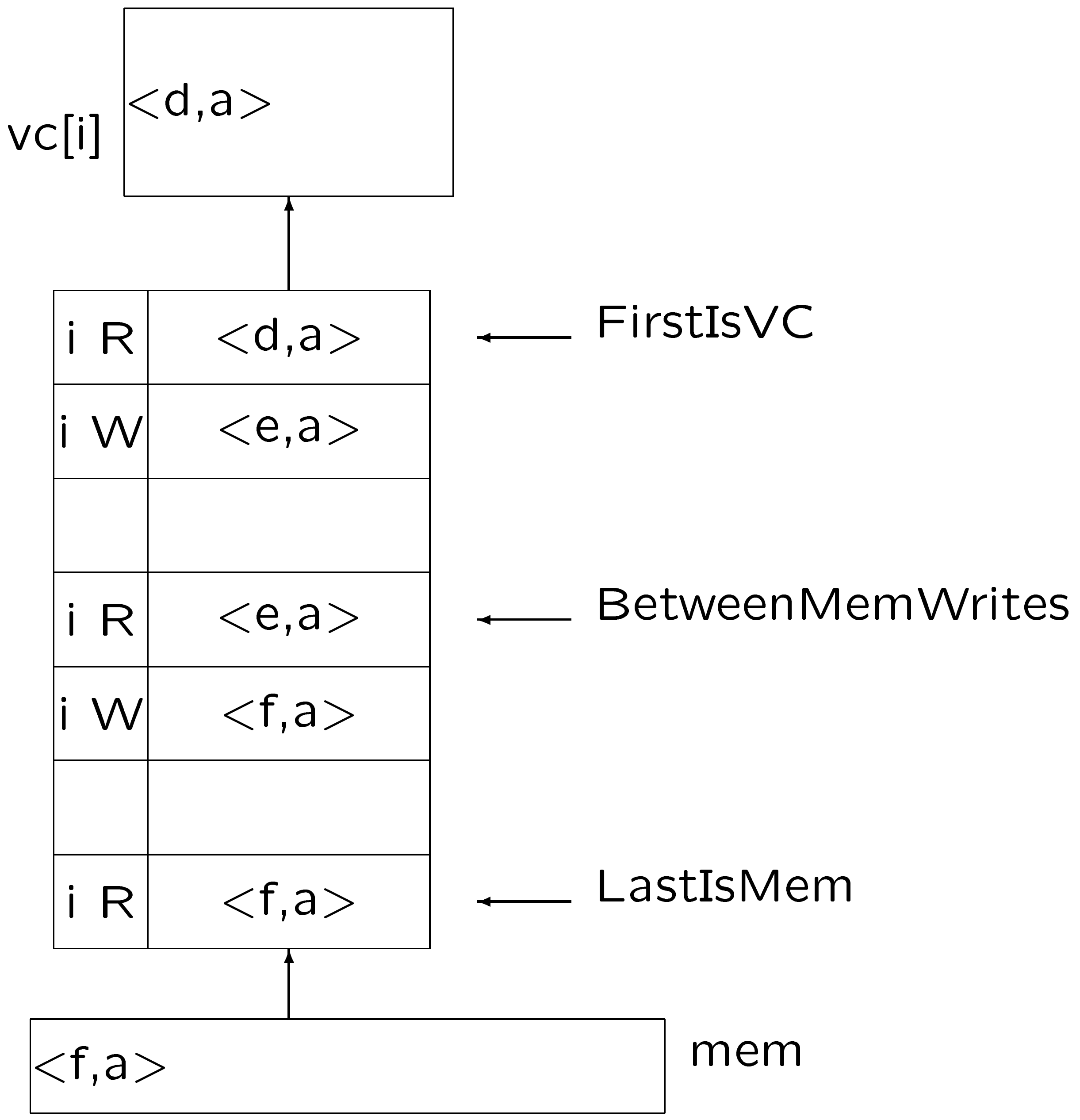
$$MemWritesFromTo(2, 7, i)$$

Hence the invariant

$$BetweenMemWrites(i) \triangleq$$
$$\forall m, n \in 1 \,..\, Len(in[i]) : MemWritesFromTo(m, n, i)$$

where

$$
\begin{array}{l}
MemWritesFromTo(m, n, i) \triangleq \\
\quad \land\ i \in Proc \\
\quad \land\ m \in 1 \,..\, Len(in[i]) \\
\quad \land\ n \in 1 \,..\, Len(in[i]) \\
\quad \land\ m < n \\
\quad \land\ NotRefresh_i(m) \\
\quad \land\ NotRefresh_i(n) \\
\quad \land\ \forall j \in m+1 \,..\, n-1 : \neg NotRefresh_i(j) \\
\quad \Rightarrow \forall j \in m+1 \,..\, n-1 : \\
\qquad\qquad \lor\ in[i][j][2] \neq in[i][m][2] \\
\qquad\qquad \lor\ in[i][j][1] = in[i][m][1]
\end{array}
$$



Illustrating the entire invariant

Final invariant:

- Refreshes between adjacent writes must be the same as the first one (BetweenMemWrites);

- When there are only refreshes in in[i], they must be the same as in vc and in mem (NoMemWrite);

- Otherwise, it's vc until the first Write, and mem beyond the last one (FirstIsVC, LastIsMem)

$$
\begin{array}{l}
NoMemWrite(a, p) \;\triangleq \\
\qquad \land\; p \in Proc \\
\qquad \land\; vc[p][a] = mem[a] \\
\qquad \land\; \forall\, k \in 1 \,..\, Len(in[p]) \;: \\
\qquad\qquad \lor\; in[p][k][2] \neq a \\
\qquad\qquad \lor\; in[p][k][1] = mem[a]
\end{array}
$$

$$
\begin{array}{l}
LastIsMem(a, p) \;\triangleq \\
\quad \land\; p \in Proc \\
\quad \land\; \exists\, k \in 1 \,..\, Len(in[p]) \;: \\
\qquad\quad \land\; NotRefresh_p(k) \\
\qquad\quad \land\; in[p][k][1] = mem[a] \\
\qquad\quad \land\; in[p][k][2] = a \\
\qquad\quad \land\; \forall\, j \in k + 1 \,..\, Len(in[p]) \;: \\
\qquad\qquad\quad \lor\; in[p][j][2] \neq a \\
\qquad\qquad\quad \lor\; in[p][j][1] = mem[a]
\end{array}
$$

$$
\begin{array}{l}
FirstIsVC(a, p) \triangleq \\
\quad \land\ p \in Proc \\
\quad \land\ \exists\, k \in 1 \,..\, Len(in[p]) : \\
\qquad \land\ NotRefresh_p(k) \\
\qquad \land\ in[p][k][2] = a \\
\qquad \land\ \forall\, j \in 1 \,..\, k-1 : \\
\qquad\quad \lor\ in[p][j][2] \neq a \\
\qquad\quad \lor\ in[p][j][1] = vc[p][a]
\end{array}
$$

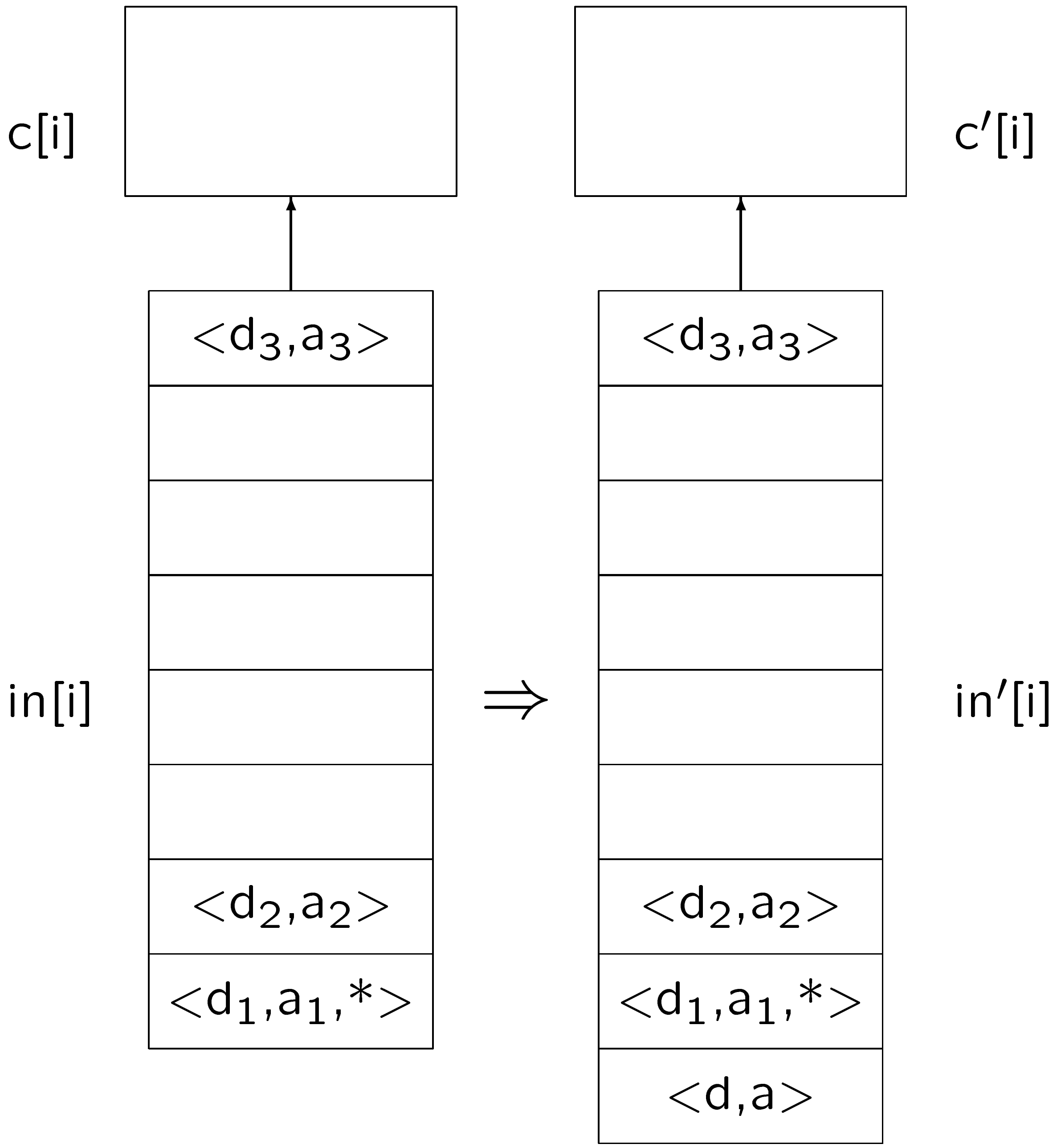
$$
\begin{array}{l}
\mathit{CCInv}.3(p) \;\triangleq \\
\quad \land\ \mathit{BetweenMemWrites}(p) \\
\quad \land\ \forall\, a \in \mathit{Addr} : \\
\qquad\qquad \lor\ \mathit{NoMemWrite}(a, p) \\
\qquad\qquad \lor\ \land\ \mathit{LastIsMem}(a, p) \\
\qquad\qquad \ \land\ \mathit{FirstIsVC}(a, p)
\end{array}
$$

That's not all:

$$
\begin{array}{lll}
CCInv.1(p) & \triangleq & \land\ Len(hin[p]) = Len(in[p]) \\
 & & \land\ Len(hout[p]) = Len(out[p]) \\
 & & \land\ \forall\, a \in Addr\ : \\
 & & \quad \forall\, d \in Data\ : \\
 & & \quad \forall\, j \in 1 \,..\, Len(in[i])\ : \\
 & & \qquad Len(in[i][j]) = 3 \\
 & & \qquad\quad \equiv \land\ hin[i][j][1] = i \\
 & & \qquad\qquad \land\ hin[i][j][2] = \text{"Wr"} \\
CCInv.2(p) & \triangleq & c[p] \in Restrict(vc[p])
\end{array}
$$

Finally,

$$
\begin{array}{lll}
CCInv & \triangleq & \forall\, p \in Proc\ : \land\ CCInv.1(p) \\
 & & \qquad\qquad\quad \land\ CCInv.2(p) \\
 & & \qquad\qquad\quad \land\ CCInv.3(p)
\end{array}
$$



MemWrite(i)

Here's $BetweenMemWrites$ for $HMemWrite$

$$
\begin{array}{l}
\land\ i \in Proc \land HMemWrite(i) \\
\land\ p \in Proc \\
\land\ CCInv.1(p) \\
\land\ BetweenMemWrites(p) \\
\land\ \forall\, a \in Addr\ :\ \begin{array}[t]{l} \lor\ LastIsMem(a, p) \\ \lor\ NoMemWrite(a, p) \end{array} \\
\quad \Rightarrow BetweenMemWrites'(p)
\end{array}
$$

Here's $FirstIsVC$

$$
\begin{array}{l}
\land\ i \in Proc \\
\land\ HMemWrite(i) \\
\land\ p \in Proc \\
\land\ CCInv.1(p) \\
\land\ a \in Addr \\
\land\ FirstIsVC(a, p) \\
\qquad \Rightarrow FirstIsVC'(a, p)
\end{array}
$$

With $NoMemWrite(Head(out[i])[2], p)$, things are less simple:

$$
\begin{array}{l}
\land\ i \in Proc \land HMemWrite(i) \\
\land\ p \in Proc \\
\land\ CCInv.1(p) \\
\land\ NoMemWrite(Head(out[i])[2], p) \\
\Rightarrow \lor\ NoMemWrite'(Head(out[i])[2], p) \\
\quad\ \ \lor\ FirstIsVC'(Head(out[i])[2], p)
\end{array}
$$

But $LastIsMem$ is easier:

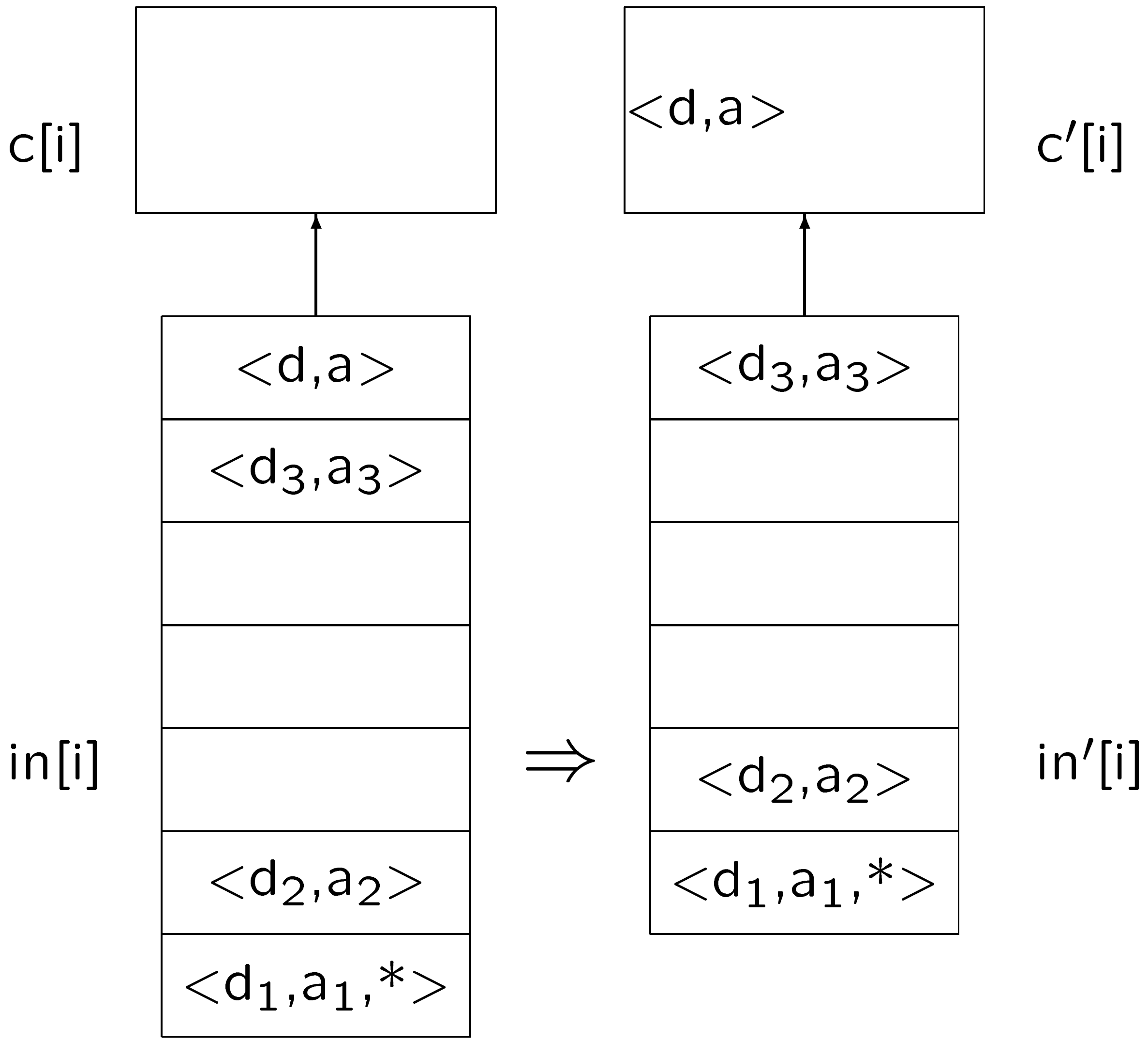
$$
\begin{array}{l}
\land\ i \in Proc \land HMemWrite(i) \\
\land\ p \in Proc \\
\Rightarrow \quad LastIsMem'(Head(out[i])[2], p)
\end{array}
$$

so we can conclude

$$
\begin{array}{l}
\land\ i \in Proc \land HMemWrite(i) \\
\land\ p \in Proc \\
\land\ CCInv.1(p) \\
\land\ NoMemWrite(Head(out[i])[2], p) \\
\Rightarrow \lor\ NoMemWrite'(Head(out[i])[2], p) \\
\quad\ \ \lor \land\ LastIsMem'(Head(out[i])[2], p) \\
\quad\ \ \land\ FirstIsVC'(Head(out[i])[2], p)
\end{array}
$$

The proof for the other values of a for $NoMemWrite$ is straightforward.

There's a symmetric problem with $FirstIsVC$ and $HCacheUpdate$.



CacheUpdate(i)

The proof of the simulation is straightforward, given the invariant.

HRead simulates LC.Read

HWrite simulates LC.Write

HMemWrite simulates LC.MemWrite

HCacheUpdate simulates either CacheUpdate (NotRefresh(Head(in[i]))) or a no-op

HMemRead and HCacheInval both simulate noops

In the buffer example, no invariant was needed, but liveness is non-trivial (we haven't seen it yet)

LazyCache is interesting because the safety proof is non-trivial.... even though liveness is trivial.

Section 10: Liveness Proofs: The Intuition

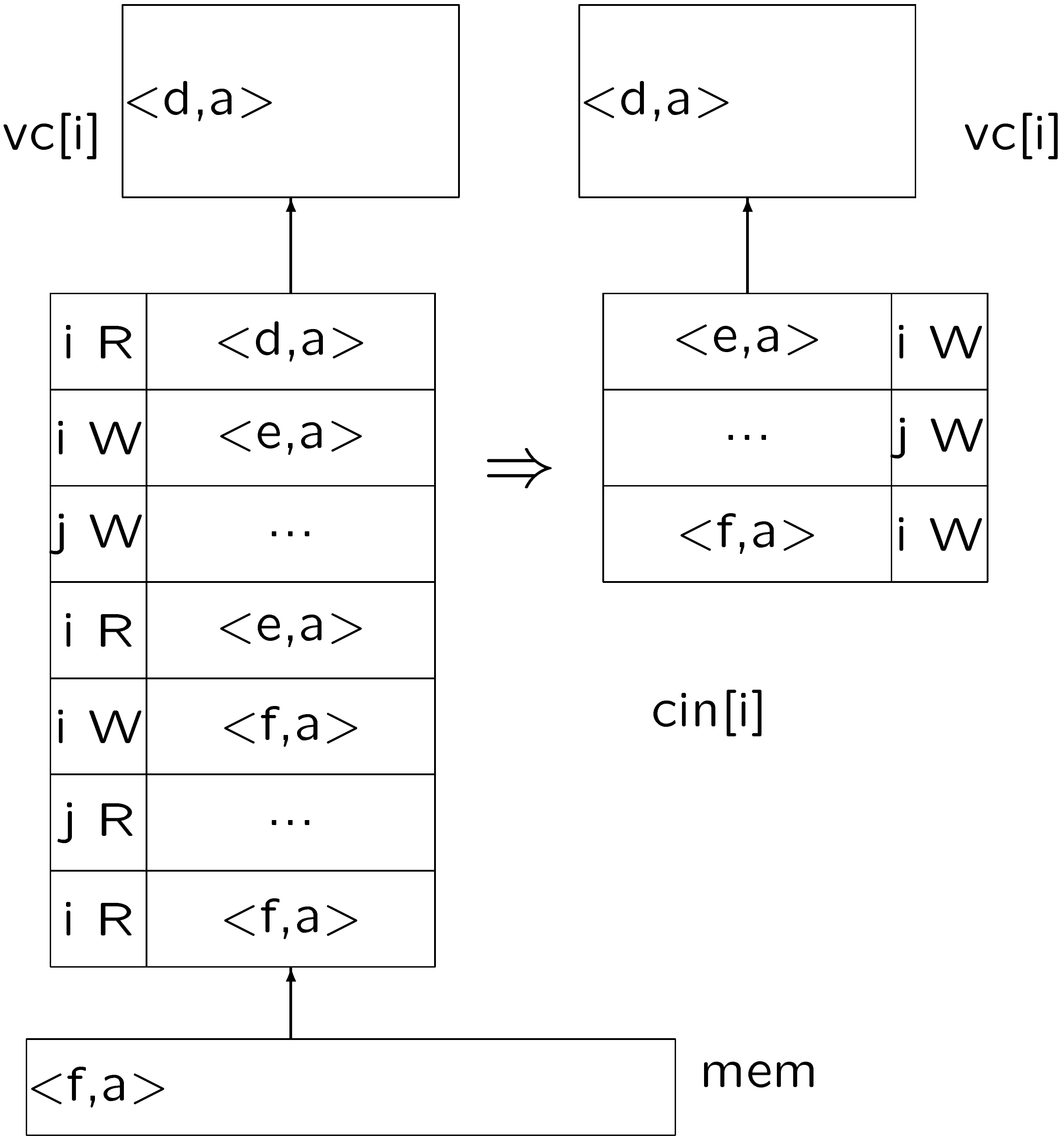
We consider Lazy Caching

Weak Fairness is asserted of both LC.MemWrite and LC.CacheUpdate, or CC.MemWrite and CC.CacheUpdate

The refinement mapping to Complete Cache is

$$\overline{cc} \leftarrow vc$$
$$\overline{cin} \leftarrow [i \in Proc \mapsto SelectSeq(hin[i], P[2] \neq \text{“Rd”})]$$
$$\overline{cout} \leftarrow hout$$
$$\overline{ch} \leftarrow ch$$

So, only $NotRefresh$ items are selected from hin. $\neg NotRefresh$ elements are invisible.



The refinement mapping to CCache

We have a similar situation as with the abstract buffer.

When there is a $NotRefresh$ in $hin[i]$, but $\neg NotRefresh(Head(in[i]))$, then $CC.CacheUpdate$ is $Enabled$ (check it) since some $NotRefresh$ is at the $Head$ of $\overline{cin}$, by definition. But $LC.CacheUpdate$ will be a CC.no-op.

$$\mathsf{WF}_{hvars}(HCacheUpdate) \equiv$$
$$\quad \lor\ \Box\Diamond\neg Enabled\langle HCacheUpdate(i)\rangle_{hvars}$$
$$\quad \lor\ \Box\Diamond\langle HCacheUpdate(i)\rangle_{hvars}$$

Fix a state, now.

If there's something in in[i] now, there's must be something in in[i][1]. And it stays there until a $HCacheUpdate(i)$.

So, either $\Diamond HCacheUpdate(i)$
or $\Box Enabled(HCacheUpdate(i))$.

$$\Box Enabled(HCacheUpdate(i)) \equiv$$
$$\quad \neg\Diamond\neg Enabled(HCacheUpdate(i))$$

so it contradicts $\Diamond\neg Enabled(HCacheUpdate(i))$ and thus $\Box\Diamond\neg Enabled(HCacheUpdate(i))$.

Conclusion: $\Diamond HCacheUpdate(i)$.

So, for any state,

$$Enabled(HCacheUpdate(i)) \Rightarrow \Diamond HCacheUpdate(i)$$

Thus

$$\Box(Enabled(HCacheUpdate(i)) \Rightarrow \Diamond HCacheUpdate(i))$$

This is a special condition called ‘leads to’:

$$Enabled(HCacheUpdate(i)) \leadsto HCacheUpdate(i)$$

Intuition: because

$$\overline{Enabled(CC.CacheUpdate(i))}$$
$$\quad \Rightarrow Enabled(HCacheUpdate(i))$$

and

$$Enabled(HCacheUpdate(i)) \leadsto HCacheUpdate(i))$$

we can prove

$$\overline{Enabled(CC.CacheUpdate(i))} \leadsto HCacheUpdate(i)$$

($A \Rightarrow B \leadsto C$ implies $A \leadsto C$)

So, since

$$\overline{Enabled(CC.CacheUpdate(i))} \leadsto HCacheUpdate(i)$$

and since $\overline{Enabled(CC.CacheUpdate(i))}$ stays true until the $NotRefresh$ is popped (and maybe afterwards), $HCacheUpdate(i)$ will repeat until the $NotRefresh$ is popped, hence intuitively

$$\overline{Enabled(CC.CacheUpdate(i))} \leadsto \overline{CC.CacheUpdate(i)}$$

Section 11: Lively Buffers, formally

Again, the concrete-abstract buffer correspondence as data structure

ConcreteBuffer		AbstractBuffer
$\begin{array}{\|c\|c\|c\|c\|c\|c\|c\|}\hline \perp & b & x & y & \perp & d & \perp \\ \hline\end{array}$	$\longrightarrow$	$<$ b,x,y,d $>$
$\begin{array}{\|c\|c\|c\|c\|c\|c\|c\|}\hline \perp & b & x & y & \perp & d & a \\ \hline\end{array}$	$\longrightarrow$	$<$ b,x,y,d,a $>$
$\begin{array}{\|c\|c\|c\|c\|c\|c\|c\|}\hline \perp & b & x & y & d & \perp & a \\ \hline\end{array}$	$\longrightarrow$	$<$ b,x,y,d,a $>$
$\begin{array}{\|c\|c\|c\|c\|c\|c\|c\|}\hline b & \perp & x & y & d & \perp & a \\ \hline\end{array}$	$\longrightarrow$	$<$ b,x,y,d,a $>$

Here's again how the operations correspond

Buffer $\qquad\qquad\qquad\qquad$ Buffer′

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & \perp \\ \hline\end{array} \xrightarrow{\text{push(a)}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & a \\ \hline\end{array}$$

$$< b,x,y,d > \xrightarrow{\text{push(a)}} < b,x,y,d,a >$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & \perp & d & a \\ \hline\end{array} \xrightarrow{\text{move(6)}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & d & \perp & a \\ \hline\end{array}$$

$$< b,x,y,d,a > \xrightarrow{\texttt{no-op}} < b,x,y,d,a >$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline \perp & b & x & y & d & \perp & a \\ \hline\end{array} \xrightarrow{\text{move(2)}} \begin{array}{|c|c|c|c|c|c|c|}\hline b & \perp & x & y & d & \perp & a \\ \hline\end{array}$$

$$< b,x,y,d,a > \xrightarrow{\texttt{no-op}} < b,x,y,d,a >$$

$$\begin{array}{|c|c|c|c|c|c|c|}\hline b & \perp & x & y & d & \perp & a \\ \hline\end{array} \xrightarrow{\text{pop}} \begin{array}{|c|c|c|c|c|c|c|}\hline \perp & \perp & x & y & d & \perp & a \\ \hline\end{array}$$

$$< b,x,y,d,a > \xrightarrow{\text{pop}} < x,y,d,a >$$

The refinement mapping

$$
\begin{aligned}
NonVoid(k) &\triangleq k \neq \perp \\
\overline{buffer} &\triangleq SelectSeq(Buffer, NonVoid)
\end{aligned}
$$

and auxiliary definitions

$$
\begin{aligned}
&FirstFull \triangleq Buffer[1] \neq \perp \\
&NotEmpty \triangleq \exists\, i \in 1 \,..\, N \,:\, Buffer[i] \neq \perp \\
&NotStuffed \triangleq \exists\, i \in 1 \,..(N-1) \,:\, Buffer[i] = \perp \\
&Descending(Buffer) \triangleq \\
&\qquad \textstyle\sum_{i=1}^{N} (Buffer[i] = \perp).2^{N-i} \\
&MaxDescending \triangleq \textstyle\sum_{i=1}^{N} 2^{N-i}
\end{aligned}
$$

The **Weak Fairness Rules**

WF1.

$$\frac{\begin{array}{l} P \wedge [\mathcal{N}]_f \;\Rightarrow\; (P' \vee Q') \\ P \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \;\Rightarrow\; Q' \\ P \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \;\Rightarrow\; (P \leadsto Q)}$$

WF2.

$$\frac{\begin{array}{l} \langle \mathcal{N} \wedge \mathcal{B} \rangle_f \;\Rightarrow\; \langle \overline{\mathcal{M}} \rangle_{\overline{g}} \\ P \wedge P' \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathcal{B} \\ P \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \\ \Box[\mathcal{N} \wedge \neg \mathcal{B}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \wedge \Box F \\ \qquad\qquad \wedge \Diamond\Box\overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \Diamond\Box P \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \wedge \Box F \;\Rightarrow\; \overline{\mathrm{WF}_g(\mathcal{M})}}$$

where

F, G, H_{c} are TLA formulas
P, Q, I are predicates
$\mathcal{A}$, $\mathcal{B}$, $\mathcal{N}$, $\mathcal{M}$ are actions
f, g are state functions

The Lattice Rule

$$\frac{\begin{array}{c}\succ \text{ a well-founded partial order on a set } \mathsf{S} \\ \mathsf{c} \text{ not free in } F \text{ or } G \\ F \land (\mathsf{c} \in \mathsf{S}) \;\Rightarrow\; (H_{\mathsf{c}} \leadsto (G \lor \exists \mathsf{d} \in \mathsf{S} : (\mathsf{c} \succ \mathsf{d}) \land H_{\mathsf{d}}))\end{array}}{F \;\Rightarrow\; ((\exists \mathsf{c} \in \mathsf{S} : H_{\mathsf{c}}) \leadsto G)}$$

For the buffer liveness proof, we use TLA Proof Rule WF2 with

$$\mathcal{N} \triangleq \begin{array}{l} \vee \; \exists\, a \,:\, CB.push(a) \\ \vee \; CB.pop \\ \vee \; \exists\, k \,:\, CB.move(k) \end{array}$$

$$\mathcal{A} \triangleq \mathcal{B} \triangleq CB.pop$$

$$\mathcal{M} \triangleq AB.pop$$

$$f = Buffer$$

$$g = buffer$$

$$P \triangleq FirstFull$$

$$\Box F \triangleq WF_{Buffer}(\exists\, k \,:\, CB.move(k))$$

The conclusion of the WF2 rule gives us the liveness property we want

$$
\begin{array}{ll}
\langle 2 \rangle 3. & \land\ \Box[\lor\ \exists\, a : CB.push(a) \\
& \qquad \lor\ CB.pop \\
& \qquad \lor\ \exists\, k : CB.move(k)]_{Buffer} \\
& \land\ WF_{Buffer}(CB.pop) \\
& \land\ WF_{Buffer}(\exists\, k : CB.move(k)) \\
& \Rightarrow \overline{WF_{buffer}(AB.pop)}
\end{array}
$$

It is justified by giving the four premises. The fourth premise is the most difficult to prove. The others are straightforward.

Proof:

$$\langle 3\rangle 1.\ \langle \wedge \vee \exists\, a : CB.push(a) \\ \quad\quad \vee\ CB.pop \\ \quad\quad \vee\ \exists\, k : CB.move(k) \\ \quad \wedge\ CB.pop\rangle_{Buffer}$$

$$\Rightarrow \langle \overline{AB.pop}\rangle_{\overline{buffer}}$$

$$\langle 3\rangle 2.\ \wedge\ FirstFull \\ \wedge\ FirstFull' \\ \wedge\ \langle CB.pop\rangle_{Buffer} \\ \wedge\ \overline{Enabled\langle AB.pop\rangle_{buffer}}$$

$$\Rightarrow CB.pop$$

$$\langle 3\rangle 3.\ \wedge\ FirstFull \\ \wedge\ \overline{Enabled\langle AB.pop\rangle_{buffer}}$$

$$\Rightarrow Enabled\langle CB.pop\rangle_{Buffer}$$

The hard one—-

$$
\begin{array}{l}
\langle 3 \rangle 4.\ \wedge\ \Box[\wedge\ \vee\ \exists\, a : CB.push(a) \\
\qquad\qquad\qquad \vee\ CB.pop \\
\qquad\qquad\qquad \vee\ \exists\, k : CB.move(k) \\
\qquad\qquad\quad \wedge\ \neg CB.pop]_{Buffer} \\
\qquad \wedge\ WF_{Buffer}(CB.pop) \\
\qquad \wedge\ WF_{Buffer}(\exists\, k : CB.move(k)) \\
\qquad \wedge\ \Diamond\Box\overline{Enabled\langle AB.pop\rangle_{buffer}} \\
\qquad\quad \Rightarrow \\
\qquad\quad \Diamond\Box FirstFull
\end{array}
$$

A reminder of how the proof goes

$$\langle 3\rangle 1.\ \langle \land \lor \exists\, a : CB.push(a) \\ \lor\ CB.pop \\ \lor \exists\, k : CB.move(k) \\ \land\ CB.pop\,\rangle_{Buffer} \\ \Rightarrow \langle \overline{AB.pop} \rangle_{\overline{buffer}}$$

Proof:

$$\langle 4\rangle 1.\ \land \lor \exists\, a : CB.push(a) \\ \lor\ CB.pop \\ \lor \exists\, k : CB.move(k)) \\ \land\ CB.pop \\ \equiv \\ CB.pop$$

$$\langle 4\rangle 2.\ \langle\, CB.pop\,\rangle_{Buffer} \\ \Rightarrow \\ \langle \overline{AB.pop} \rangle_{\overline{buffer}}$$

$$
\langle 3 \rangle 2.\quad \begin{aligned}
&\land\ FirstFull \\
&\land\ FirstFull' \\
&\land\ \langle\, CB.pop \,\rangle_{Buffer} \\
&\land\ \overline{Enabled\langle\, AB.pop \,\rangle_{buffer}} \\
&\Rightarrow\ CB.pop
\end{aligned}
$$

Just look at it

$\langle 3 \rangle 3.\ \wedge\ FirstFull$

$\wedge\ \overline{Enabled\langle\, AB.pop\,\rangle_{buffer}}$

$\Rightarrow\ Enabled\langle\, CB.pop\,\rangle_{Buffer}$

$\langle 4 \rangle 1.\ FirstFull$

$\Rightarrow$

$Enabled\langle\, CB.pop\,\rangle_{Buffer}$

$$
\begin{array}{ll}
\langle 3 \rangle 4. & \land\ \Box[\land\ \lor\ \exists\, a : CB.push(a) \\
& \qquad\qquad \lor\ CB.pop \\
& \qquad\qquad \lor\ \exists\, k : CB.move(k) \\
& \qquad\quad \land\ \neg CB.pop]_{Buffer} \\
& \land\ WF_{Buffer}(CB.pop) \\
& \land\ WF_{Buffer}(\exists\, k : CB.move(k)) \\
& \land\ \Diamond\Box\overline{Enabled\langle AB.pop\rangle_{buffer}} \\
& \Rightarrow \Diamond\Box FirstFull
\end{array}
$$

Proof:

First, a simplified version

$$
\begin{array}{ll}
\langle 4 \rangle 1. & \land\ \Box[\lor\ \exists\, a : CB.push(a) \\
& \qquad\quad \lor\ \exists\, k : CB.move(k)]_{Buffer} \\
& \land\ WF_{Buffer}(CB.pop) \\
& \land\ WF_{Buffer}(\exists\, k : CB.move(k)) \\
& \land\ \Diamond\Box NotEmpty \\
& \Rightarrow \Diamond\Box FirstFull
\end{array}
$$

Lecture 12: The Bridge Rule: Towards a Schema for Liveness

The Bridge Rule

Let X, Y, Z be propositional variables (that is, symbols standing for any sentences). Then

$$\frac{\begin{array}{l}\Box X \wedge Y \Rightarrow \Box Y \\ \Box X \Rightarrow \Box(Z \Rightarrow \Diamond Y)\end{array}}{\Box X \wedge \Diamond\Box Z \Rightarrow \Diamond\Box Y}$$

Now for an interpretation.

Suppose X is an action, which is a disjunction of actions. Then $\Box X$ is a safety property. Let Y be a condition. We show

$$\Box X \wedge Y \Rightarrow \Box Y$$

X will be $\mathcal{N} \wedge \neg\mathcal{B}$, and Y will be something like the extra enabling condition for the concrete action.

We thus show that Y is an invariant if $\mathcal{B}$ never happens.

The formula

$$\Box X \Rightarrow \Box(Z \Rightarrow \Diamond Y)$$

is the same as (defined to be)

$$\Box X \Rightarrow (Z \rightsquigarrow Y)$$

which is the conclusion of the Lattice Rule.

The conclusion of the Bridge Rule is

$$\Box X \wedge \Diamond\Box Z \Rightarrow \Diamond\Box Y$$

which is the form of the fourth hypothesis of WF2. Thus the Bridge Rule bridges the logical gap between the Lattice Rule and the fourth WF2 premise.

How is $\Box X \wedge Y \Rightarrow \Box Y$ proved?

Using INV1:

$$\frac{\Box X \wedge Y \Rightarrow Y'}{\Box X \wedge Y \Rightarrow \Box Y}$$

How is $\Box X \Rightarrow \Box(Z \Rightarrow \Diamond Y)$ proved?
Using the Lattice Rule.

And we draw the conclusion

$$\Box X \wedge \Diamond\Box Z \Rightarrow \Diamond\Box Y$$

which is the temporal premise of WF2.

In the proof of the Bridge Rule, we use the TL rules

$$
\begin{array}{ll}
\text{STL7} & \vdash \Box\Diamond\Box F \equiv \Diamond\Box F \\
\text{STL2}^{\Diamond} & \vdash F \Rightarrow \Diamond F \\
\text{STL4}^{\Diamond} & \dfrac{F \Rightarrow G}{\Diamond F \Rightarrow \Diamond G} \\
\text{STL6}^{\Diamond} & \vdash (\Box\Diamond F) \lor (\Box\Diamond G) \equiv \Box\Diamond(F \lor G)
\end{array}
$$

The latter three are easily shown from their normal forms by using $\Box\neg \equiv \neg\Diamond$ and $\Diamond\neg \equiv \neg\Box$.

e.g. STL2$^{\Diamond}$ by using the contrapositive:
$A \Rightarrow B$ if and only if $\neg B \Rightarrow \neg A$

STL6$^{\Diamond}$ by using De Morgan's laws

⟨5⟩1. $$\frac{\Box X \wedge FF \Rightarrow \Box FF \qquad \Box X \Rightarrow \Box(NE \Rightarrow \Diamond FF)}{\Box X \wedge \Diamond\Box NE \Rightarrow \Diamond\Box FF}$$

Proof:

⟨6⟩1. $$\frac{Z \wedge FF \Rightarrow \Box FF}{\Box Z \Rightarrow \Box(FF \Rightarrow \Box FF)}$$

⟨6⟩2. $$\frac{\Box X \Rightarrow \Box(NE \Rightarrow \Diamond FF) \qquad \Box X \Rightarrow \Box(FF \Rightarrow \Box FF)}{\Box X \Rightarrow \Box(NE \Rightarrow \Box\Diamond FF)}$$

⟨6⟩3. $$\frac{\Box X \Rightarrow \Box(NE \Rightarrow \Box Y)}{\Box X \wedge \Box NE \Rightarrow \Box Y}$$

⟨6⟩4. $$\frac{\Box X \wedge \Box NE \Rightarrow \Box Y}{\Box(X \wedge NE) \Rightarrow \Box Y}$$

⟨6⟩5. $$\frac{\Box(X \wedge NE) \Rightarrow \Box Y)}{\Diamond\Box(X \wedge NE) \Rightarrow \Diamond\Box Y}$$

⟨6⟩6. $$\frac{\Diamond\Box(X \wedge NE) \Rightarrow \Diamond\Box Y)}{\Diamond\Box X \wedge \Diamond\Box NE \Rightarrow \Diamond\Box Y}$$

⟨6⟩7. $$\frac{\Diamond\Box X \wedge \Diamond\Box NE \Rightarrow \Diamond\Box Y}{\Box X \wedge \Diamond\Box NE \Rightarrow \Diamond\Box Y}$$

⟨6⟩8. $$\frac{\Box X \wedge \Diamond\Box NE \Rightarrow \Diamond\Box\Diamond FF}{\Box X \wedge \Diamond\Box NE \Rightarrow \Diamond\Box FF}$$

$\langle 6 \rangle 9.$ Q.E.D.

⟨6⟩1. $$\frac{Z \land FF \Rightarrow \Box FF}{\Box Z \Rightarrow \Box(FF \Rightarrow \Box FF)}$$

Proof: Follows immediately by trivial propositional logic and then STL4. $\Box$

⟨6⟩2. $$\frac{\begin{array}{c}\Box X \Rightarrow \Box(NE \Rightarrow \Diamond FF)\\ \Box X \Rightarrow \Box(FF \Rightarrow \Box FF)\end{array}}{\Box X \Rightarrow \Box(NE \Rightarrow \Box\Diamond FF)}$$

Proof: ****************

⟨6⟩3. $$\frac{\Box X \Rightarrow \Box(NE \Rightarrow \Box Y)}{\Box X \land \Box NE \Rightarrow \Box Y}$$

Proof: This rule follows from PTL. $\Box$

⟨6⟩4. $$\frac{\Box X \land \Box NE \Rightarrow \Box Y}{\Box(X \land NE) \Rightarrow \Box Y}$$

Proof: Immediate from STL5 and substitution of equivalent subformulas. $\Box$

$\langle 6 \rangle$5. $\dfrac{\Box(X \land NE) \Rightarrow \Box Y)}{\Diamond\Box(X \land NE) \Rightarrow \Diamond\Box Y}$

Proof: Immediate from STL4$^{\Diamond}$. $\Box$

$\langle 6 \rangle$6. $\dfrac{\Diamond\Box(X \land NE) \Rightarrow \Diamond\Box Y)}{\Diamond\Box X \land \Diamond\Box NE \Rightarrow \Diamond\Box Y}$

Proof: STL6 and substituting equivalent subformulas. $\Box$

$\langle 6 \rangle$7. $\dfrac{\Diamond\Box X \land \Diamond\Box NE \Rightarrow \Diamond\Box Y}{\Box X \land \Diamond\Box NE \Rightarrow \Diamond\Box Y}$

Proof: Strengthening a hypothesis, using STL2$^{\Diamond}$. $\Box$

$\langle 6 \rangle$8. $\dfrac{\Box X \land \Diamond\Box NE \Rightarrow \Diamond\Box\Diamond FF}{\Box X \land \Diamond\Box NE \Rightarrow \Diamond\Box FF}$

Proof: STL7, and substituting equivalent subformulas. $\Box$

$\langle 6 \rangle$9. Q.E.D.

Proof: Note $Y \triangleq \Diamond FF$.

$\langle 6 \rangle 2.$
$$\frac{\Box X \Rightarrow \Box(NE \Rightarrow \Diamond FF) \qquad \Box X \Rightarrow \Box(FF \Rightarrow \Box FF)}{\Box X \Rightarrow \Box(NE \Rightarrow \Box\Diamond FF)}$$

Proof:

$\langle 7 \rangle 1.$
$$\begin{array}{l} \land\ \Box(NE \Rightarrow \Diamond FF) \\ \land\ \Box(FF \Rightarrow \Box FF) \\ \quad \Rightarrow \\ \quad\quad \Box(NE \Rightarrow \Box\Diamond FF) \end{array}$$

$\langle 7 \rangle 2.$ Q.E.D.

Proof: The conclusion of the rule follows from the premises using $\langle 7 \rangle 1$, Modus Ponens, and propositional logic. □

$\langle 7 \rangle 1.$ $\wedge\ \Box(NE \Rightarrow \Diamond FF)$
$\wedge\ \Box(FF \Rightarrow \Box FF)$
$\Rightarrow$
$\Box(NE \Rightarrow \Box\Diamond FF)$

Proof:

$\langle 8 \rangle 1.$ $\wedge\ \Box(NE \Rightarrow \Diamond FF)$
$\wedge\ \Box(FF \Rightarrow \Box FF)$
$\Rightarrow$
$\wedge\ \Box(NE \Rightarrow \Diamond FF)$
$\wedge\ \Box(FF \Rightarrow \Diamond\Box FF)$

$\langle 8 \rangle 2.$ $\wedge\ \Box(NE \Rightarrow \Diamond FF)$
$\wedge\ \Box(FF \Rightarrow \Diamond\Box FF)$
$\equiv$
$\wedge\ (NE \leadsto FF)$
$\wedge\ (FF \leadsto \Box FF)$

$\langle 8 \rangle 3.$ $\wedge\ (NE \leadsto FF)$
$\wedge\ (FF \leadsto \Box FF)$
$\Rightarrow$
$NE \leadsto \Box FF$

$\langle 8 \rangle 4.$ $NE \leadsto \Box FF \equiv \Box(NE \Rightarrow \Diamond\Box FF)$

$\langle 8 \rangle 5.$ $\Box(NE \Rightarrow \Diamond\Box FF) \Rightarrow \Box(NE \Rightarrow \Box\Diamond\Box FF)$

$\langle 8 \rangle 6.$ $\Box(NE \Rightarrow \Box\Diamond\Box FF) \Rightarrow \Box(NE \Rightarrow \Box\Diamond FF)$

$\langle 8 \rangle 1.$ $\land\ \Box(NE \Rightarrow \Diamond FF)$
$\land\ \Box(FF \Rightarrow \Box FF)$
$\Rightarrow$
$\land\ \Box(NE \Rightarrow \Diamond FF)$
$\land\ \Box(FF \Rightarrow \Diamond\Box FF)$

Proof sketch: Uses STL4, STL2$^{\Diamond}$ and propositional logic. $\Box$

$\langle 8 \rangle 2.$ $\land\ \Box(NE \Rightarrow \Diamond FF)$
$\land\ \Box(FF \Rightarrow \Diamond\Box FF)$
$\equiv$
$\land\ (NE \leadsto FF)$
$\land\ (FF \leadsto \Box FF)$

Proof: Follows immediately from the definition of $\leadsto$. $\Box$

$\langle 8 \rangle 3.$ $\wedge\ (NE \leadsto FF)$
$\wedge\ (FF \leadsto \Box FF)$
$\Rightarrow$
$NE \leadsto \Box FF$

Proof sketch: Follows by transitivity of $\leadsto$. $\Box$

$\langle 8 \rangle 4.$ $NE \leadsto \Box FF \equiv \Box(NE \Rightarrow \Diamond\Box FF)$

Proof: Follows immediately by the definition of $\leadsto$. $\Box$

$\langle 8 \rangle 5.$ $\Box(NE \Rightarrow \Diamond\Box FF) \Rightarrow \Box(NE \Rightarrow \Box\Diamond\Box FF)$

Proof sketch: Follows by STL4 and STL7. $\Box$

$\langle 8 \rangle 6.$ $\Box(NE \Rightarrow \Box\Diamond\Box FF) \Rightarrow \Box(NE \Rightarrow \Box\Diamond FF)$

Proof sketch: Follows by STL4 and STL2, using substitution of equivalent subformulas. $\Box$

Done

I haven't yet finished analysing the proof obligations, so I won't be able to give them here.

Let's see how the proof finishes up.

The first Bridge Rule hypothesis

$\langle 5 \rangle 2.$ $\land\ \Box[\lor\ \exists\, a : CB.push(a)$
$\quad\quad \lor\ \exists\, k : CB.move(k)]_{Buffer}$
$\land\ WF_{Buffer}(CB.pop)$
$\land\ WF_{Buffer}(\exists\, k : CB.move(k))$
$\land\ FirstFull$
$\Rightarrow$
$\Box FirstFull$

Proof:

$\langle 6 \rangle 1.$ $\land\ \Box[\lor\ \exists\, a : CB.push(a)$
$\quad\quad \lor\ \exists\, k : CB.move(k)]_{Buffer}$
$\land\ FirstFull$
$\Rightarrow$
$\Box FirstFull$

$\langle 6 \rangle 2.$ Q.E.D.

Proof: Immediate from $\langle 6 \rangle 1$ by adding a hypothesis. $\Box$

⟨6⟩1. $\land\ \Box[\lor\ \exists\, a : CB.push(a)$
$\quad\quad\lor\ \exists\, k : CB.move(k)]_{Buffer}$
$\land\ FirstFull$
$\Rightarrow$
$\Box FirstFull$

Proof:

⟨7⟩1. $\land\ [\lor\ \exists\, a : CB.push(a)$
$\quad\quad\lor\ \exists\, k : CB.move(k)]_{Buffer}$
$\land\ FirstFull$
$\Rightarrow$
$FirstFull'$

Proof: Follows from the definitions of $push$ and $move$. □

⟨7⟩2. Q.E.D.

Proof: Immediate by TLA Proof Rule INV1 from ⟨7⟩1, with the antecedent conjuncts in the other order. □

Section 13: The Final Steps:
use of the Lattice Rule and WF1

The Lattice Rule

$$\frac{\begin{array}{c} \succ \text{ a well-founded partial order on a set S} \\ \text{c not free in } F \text{ or } G \\ F \wedge (\mathsf{c} \in \mathsf{S}) \;\Rightarrow\; (H_{\mathsf{c}} \leadsto (G \vee \exists \mathsf{d} \in \mathsf{S} : (\mathsf{c} \succ \mathsf{d}) \wedge H_{\mathsf{d}})) \end{array}}{F \;\Rightarrow\; ((\exists \mathsf{c} \in \mathsf{S} : H_{\mathsf{c}}) \leadsto G)}$$

The Weak Fairness Rule WF1.

WF1.

$$\frac{\begin{array}{l} P \wedge [\mathcal{N}]_f \;\Rightarrow\; (P' \vee Q') \\ P \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \;\Rightarrow\; Q' \\ P \;\Rightarrow\; \mathit{Enabled}\, \langle \mathcal{A} \rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{WF}_f(\mathcal{A}) \;\Rightarrow\; (P \leadsto Q)}$$

We have seen how to get,
using the Bridge Rule,

$$
\begin{array}{ll}
\langle 4 \rangle 4. & \wedge\ \Box[\vee\ \exists\, a\ :\ CB.push(a) \\
 & \qquad \vee\ \exists\, k\ :\ CB.move(k)]_{Buffer} \\
 & \wedge\ WF_{Buffer}(CB.pop) \\
 & \wedge\ WF_{Buffer}(\exists\, k\ :\ CB.move(k)) \\
 & \wedge\ \Diamond\Box NotEmpty \\
 & \Rightarrow\ \Diamond\Box FirstFull
\end{array}
$$

from

$$
\begin{array}{ll}
\langle 5 \rangle 3. & \wedge\ \Box[\exists\, a\ :\ CB.push(a).....]_{Buffer} \\
 & \wedge\ WF_{Buffer}(CB.pop) \\
 & \wedge\ WF_{Buffer}(\exists\, k\ :\ CB.move(k)) \\
 & \Rightarrow\ \Box(NotEmpty \Rightarrow \Diamond FirstFull)
\end{array}
$$

Now we prove the latter.

First a temporal logic equivalence.
For $X \triangleq$ ......:

$\langle 6 \rangle$1. $\Box X \Rightarrow$
$(NotEmpty \land NotStuffed \Rightarrow \Diamond FirstFull)$
$\equiv$
$(NotEmpty \Rightarrow \Diamond FirstFull)$

Proof:

$\langle 7 \rangle$1. $\Box X \Rightarrow$
$\neg NotStuffed \Rightarrow \Diamond FirstFull$

Proof:

$\langle 8 \rangle$1. $\Box X \Rightarrow \neg NotStuffed \Rightarrow FirstFull$

Proof: Immediate from definitions. $\Box$

$\langle 8 \rangle$2. $FirstFull \Rightarrow \Diamond FirstFull$

Proof: Immediate from STL2. $\Box$

$\langle 8 \rangle$3. Q.E.D.

Proof: Immediately from $\langle 8 \rangle$1 and $\langle 8 \rangle$2. $\Box$

$\langle 7 \rangle$2. Q.E.D.

Proof: Propositional logic from $\langle 7 \rangle$1. $\Box$

Substituting in the consequent, using this new equivalence, we prove

$$\begin{array}{ll} \langle 6\rangle 2. & \land\ \Box[\exists\, a : CB.push(a)......]_{Buffer} \\ & \land\ WF_{Buffer}(CB.pop) \\ & \land\ WF_{Buffer}(\exists\, k : CB.move(k)) \\ & \Rightarrow \Box(\land\ NotEmpty \\ & \qquad\ \land\ NotStuffed \\ & \qquad\ \Rightarrow \Diamond FirstFull) \end{array}$$

which is shown using the Lattice Rule

First, we fiddle with the consequent

$$
\begin{array}{l}
\langle 7 \rangle 2.\ \land\ \Box[\exists\, a : CB.push(a)\ldots\ldots\ldots \\
\qquad\ \land\ WF_{\mathit{Buffer}}(CB.pop) \\
\qquad\ \land\ WF_{\mathit{Buffer}}(\exists\, k : CB.move(k)) \\
\qquad\quad \Rightarrow \\
\qquad\quad \Box(\land\ \exists\, i \in 0 \mathrel{..} MaxDescending : \\
\qquad\qquad\qquad Descending(Buffer) = i \\
\qquad\qquad \land\ NotEmpty \\
\qquad\qquad \land\ NotStuffed \\
\qquad\qquad\quad \Rightarrow \\
\qquad\qquad\quad \Diamond FirstFull)
\end{array}
$$

We can do this because

$$\exists\, i \in 0 \,..\, MaxDescending \,:\, Descending(Buffer) = i$$

is provable from arithmetic, thus

$$\langle 7 \rangle 2.\ \ \square(\exists\, i \in 0 \,..\, MaxDescending \,: \\ Descending(Buffer) = i)$$

by STL4.

Thus $A \Rightarrow \square B$ is equivalent to

$$A \Rightarrow \square B \land \square(\exists\, i \in 0 \,..\, MaxDescending \,:\, ...)$$

and then one uses STL5 to compactify the conclusion

Thus the instantiations for the Lattice Rule are:

$$\exists\, i : H_i \;\triangleq\; \begin{array}{l} \land\ \exists\, i \in 0 \mathrel{..} \mathit{MaxDescending} : \\ \qquad \mathit{Descending}(\mathit{Buffer}) = i \\ \land\ \mathit{NotEmpty} \\ \land\ \mathit{NotStuffed} \end{array}$$

which suggests that

$$H_i \;\triangleq\; \begin{array}{l} \land\ i \in 0 \mathrel{..} \mathit{MaxDescending} : \\ \land\ \mathit{Descending}(\mathit{Buffer}) = i \\ \land\ \mathit{NotEmpty} \\ \land\ \mathit{NotStuffed} \end{array}$$

and $G \;\triangleq\; \mathit{FirstFull}$

Then we show this using the Lattice Rule.
The consequent of the premise, remember, is

$$(H_{\mathsf{c}} \leadsto (G \lor \exists \mathsf{d} \in \mathsf{S} : (\mathsf{c} \succ \mathsf{d}) \land H_{\mathsf{d}}))$$

Here's the premise

$$
\begin{array}{l}
\langle 8 \rangle 1.\ \land\ \Box[\lor\ \exists\, a : \mathit{CB}.\mathit{push}(a) \\
\qquad\qquad\quad \lor\ \exists\, k : \mathit{CB}.\mathit{move}(k)]_{\mathit{Buffer}} \\
\qquad\quad \land\ \mathit{WF}_{\mathit{Buffer}}(\mathit{CB}.\mathit{pop}) \\
\qquad\quad \land\ \mathit{WF}_{\mathit{Buffer}}(\exists\, k : \mathit{CB}.\mathit{move}(k)) \\
\qquad\quad \land\ i \in 0 \,..\, \mathit{MaxDescending} \\
\qquad\qquad \Rightarrow \\
\qquad\qquad \Box(\land\ \mathit{NotEmpty} \\
\qquad\qquad\quad\ \land\ \mathit{NotStuffed} \\
\qquad\qquad\quad\ \land\ \mathit{Descending}(\mathit{Buffer}) = i) \\
\qquad\qquad\qquad \Rightarrow \\
\qquad\qquad\qquad \Diamond(\lor\ \mathit{FirstFull} \\
\qquad\qquad\qquad\quad\ \lor\ \exists\, j \in 0 \,..\, \mathit{MaxDescending} : \\
\qquad\qquad\qquad\qquad\quad \land\ j < i \\
\qquad\qquad\qquad\qquad\quad \land\ \mathit{NotEmpty} \\
\qquad\qquad\qquad\qquad\quad \land\ \mathit{NotStuffed} \\
\qquad\qquad\qquad\qquad\quad \land\ \mathit{Descending}(\mathit{Buffer}) = j)
\end{array}
$$

This step is justified by an application of WF1 (and simple propositional logic). The substeps consist of the premises of WF1, with the instantiations

$$
\begin{array}{lcl}
\mathcal{N} & \triangleq & \exists\, a : CB.push(a) \lor \exists\, k : CB.move(k), \\
\mathcal{A} & \triangleq & \exists\, k : CB.move(k), \\
P & \triangleq & \land\ NotEmpty \\
 & & \land\ NotStuffed \\
 & & \land\ Descending(Buffer) = i
\end{array}
$$

and

$$
\begin{array}{lcl}
Q & \triangleq & \lor\ FirstFull \\
 & & \lor\ \exists\, j \in 0 \,..\, MaxDescending : \\
 & & \qquad \land\ j < i \\
 & & \qquad \land\ NotEmpty \\
 & & \qquad \land\ NotStuffed \\
 & & \qquad \land\ Descending(Buffer) = j)
\end{array}
$$

First, some trivial simplification of the consequent,
from

$$\exists j \in 0 \mathrel{.\,.} MaxDescending : j < i \land \Theta$$

to $\exists j < i \land \Theta$, for appropriate Θ.

$$
\begin{array}{ll}
\langle 9 \rangle 1. & \land\ \exists j < i : Descending(Buffer) = j \\
& \qquad \equiv \\
& \qquad \exists j \in 0 \mathrel{.\,.} MaxDescending : \\
& \qquad\quad \land\ j < i \\
& \qquad\quad \land\ Descending(Buffer) = j \\
& \land\ \exists j < i : Descending(Buffer') = j \\
& \qquad \equiv \\
& \qquad \exists j \in 0 \mathrel{.\,.} MaxDescending : \\
& \qquad\quad \land\ j < i \\
& \qquad\quad \land\ Descending(Buffer') = j
\end{array}
$$

Proof: Pretty obvious from arithmetic and predicate logic. $\square$

The next step is the first premise of WF1,
$P \land [\mathcal{N}]_f \;\Rightarrow\; (P' \lor Q')$

$$
\begin{array}{l}
\langle 9 \rangle 2.\ \begin{array}[t]{l}
\land\ \mathit{NotEmpty} \\
\land\ \mathit{NotStuffed} \\
\land\ \mathit{Descending}(\mathit{Buffer}) = i \\
\land\ \begin{array}[t]{l}
\lor\ \exists\, a\ :\ \mathit{CB}.\mathit{push}(a) \\
\lor\ \exists\, k\ :\ \mathit{CB}.\mathit{move}(k) \\
\lor\ \mathit{Buffer}' = \mathit{Buffer}
\end{array} \\
\quad \Rightarrow \\
\quad \lor\ \begin{array}[t]{l}
\land\ \mathit{NotEmpty}' \\
\land\ \mathit{NotStuffed}' \\
\land\ \mathit{Descending}(\mathit{Buffer}') = i
\end{array} \\
\quad \lor\ \mathit{FirstFull}' \\
\quad \lor\ \exists\, j \in 0 \,..\, \mathit{MaxDescending}\ : \\
\qquad\qquad \begin{array}[t]{l}
\land\ j < i \\
\land\ \mathit{NotEmpty}' \\
\land\ \mathit{NotStuffed}' \\
\land\ \mathit{Descending}(\mathit{Buffer}') = j
\end{array}
\end{array}
\end{array}
$$

The second premise of WF1,
$P \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \;\Rightarrow\; Q'$

$$
\begin{array}{ll}
\langle 9 \rangle 3. & \wedge\ NotStuffed \\
& \wedge\ NotEmpty \\
& \wedge\ Descending(Buffer) = i \\
& \wedge\ \langle \wedge\ \vee\ \exists\, a : CB.push(a) \\
& \qquad\quad \vee\ \exists\, k : CB.move(k) \\
& \quad\ \ \wedge\ \exists\, k : CB.move(k) \rangle_{Buffer} \\
& \quad \Rightarrow \\
& \quad \vee\ FirstFull' \\
& \quad \vee\ \exists\, j \in 0 \,..\, MaxDescending : \\
& \qquad\quad \wedge\ j < i \\
& \qquad\quad \wedge\ NotStuffed' \\
& \qquad\quad \wedge\ NotEmpty' \\
& \qquad\quad \wedge\ Descending(Buffer') = j
\end{array}
$$

This reduces to proving

$$
\begin{array}{l}
\langle 10 \rangle 2. \ \land \ \mathit{NotStuffed} \\
\qquad \land \ \mathit{NotEmpty} \\
\qquad \land \ \mathit{Descending}(\mathit{Buffer}) = i \\
\qquad \land \ \langle \exists\, k \ : \ \mathit{CB}.\mathit{move}(k) \rangle_{\mathit{Buffer}} \\
\qquad\quad \Rightarrow \\
\qquad\quad \lor \ \mathit{FirstFull}' \\
\qquad\quad \lor \ \exists\, j \in 0 \,..\, \mathit{MaxDescending} \ : \\
\qquad\qquad\qquad \land \ j < i \\
\qquad\qquad\qquad \land \ \mathit{NotStuffed}' \\
\qquad\qquad\qquad \land \ \mathit{NotEmpty}' \\
\qquad\qquad\qquad \land \ \mathit{Descending}(\mathit{Buffer}') = j
\end{array}
$$

And the third premise of WF1,

$P \;\Rightarrow\; Enabled\,\langle \mathcal{A} \rangle_f$

$$\begin{array}{ll}\langle 9 \rangle 4. & \land\; NotStuffed \\ & \land\; NotEmpty \\ & \land\; Descending(Buffer) = i \\ & \quad \Rightarrow \\ & \quad Enabled\langle \exists\, k \;:\; CB.move(k) \rangle_{Buffer}\end{array}$$

Proof: Follows by predicate logic from definitions. $\square$

The proofs of the first two premises involve some arithmetic.

In fact, we can prove

$$
\begin{array}{l}
\wedge\ Descending(Buffer) = i \\
\wedge\ \exists\, k\ :\ CB.move(k) \\
\quad \Rightarrow \\
\quad \exists\, j < i\ :\ Descending(Buffer') = j
\end{array}
$$

which enables us to prove both the first two premises.

Why is this? Let's go back to the definition
$Descending(Buffer) \triangleq \sum_{i=1}^{N}(Buffer[i] = \perp).2^{N-i}$

$$Descending(Buffer) \triangleq \sum_{i=1}^{N}(Buffer[i] = \perp).2^{N-i}$$

So, under a $CB.move(k)$, where $2 \leq k \leq N$,

$Buffer[k] \quad = a$ (say) contributing 0
$Buffer[k-1] = \perp$ " $\quad 2^{N-(k-1)}$

and

$Buffer'[k] \quad = \perp$ contributing 2^{N-k}
$Buffer'[k-1] = a$ " $\quad 0$

so $Descending(Buffer) =$
$$Descending(Buffer') + 2^{N-k}.(2-1)$$

So $Descending(Buffer') < Descending(Buffer)$
which is what we need

I'll spare you details of the arithmetic, but it's necessary to have a TLA+ module for the $Naturals$