UNIVERSITY OF CAMBRIDGE

Hoare Logic and Model Checking

Alan Mycroft

Computer Laboratory, University of Cambridge, UK
http://www.cl.cam.ac.uk/~am21

CST Part II – 2015/16

Acknowledgement: slides heavily based on those for Mike Gordon's 2014/15 courses

Lecture 1: Introduction

What's the course about?

- Showing programs do what they're intended to do (this includes them not doing bad things, e.g. buffer overflow).
- Can't testing do this?

  "Testing shows the presence, not the absence of bugs" [Edsger W. Dijkstra, winner of the 1972 Turing Award]

- Incidentally, *Software Testing* is more than "just writing a few unit tests and systems tests".
  Off-scope for this course, but see for example:

https://en.wikipedia.org/wiki/Code_coverage
https://en.wikipedia.org/wiki/Software_testing

- So we're going to *prove* our programs correct.

Rough Outline

- Introduction
- Hoare Logic (5–6 lectures)
- Lecture 6: 30 mins Q&A with (Sir) Tony Hoare (Wednesday 24 February 2016)
- Temporal Logic and Model Checking (5–6 Lectures)

Two different techniques for proving programs correct.

Based on Mike Gordon's two courses given in 2014/15 (many exercises still apply).

Example 1

What does this program do?

```
R := 1;
WHILE (N != 0) DO ( R := R*N; N := N-1 )
```

It tries to compute factorial – but what *formal specification* do we want it to satisfy? (E.g. is it OK that `N` is corrupted, or that `N<0` causes a loop?)
And how do we prove this it satisfies its specification?

Note: the factorial function doesn't appear in the source code anywhere.

We'll use *Hoare Logic* for this.

Example 2

```
bool flag[2] = {false, false};     int turn;

Thread 1: flag[0] = true;
          turn = 1;
          while (flag[1] && turn == 1); // busy wait
          // critical section
          flag[0] = false;
          // non-critical stuff
          repeat;

Thread 2: flag[1] = true;
          turn = 0;
          while (flag[0] && turn == 0); // busy wait
          // critical section
          flag[1] = false;
          // non-critical stuff
          repeat;
```

We can use *Model Checking* to prove this implements mutual exclusion without using locks (Peterson's algorithm).

Programs and Specifications

- We can't prove a program correct in isolation.
- Need a *specification* of what it is intended to do.
  - Hoare triples $\{P\}$ C $\{Q\}$ for Hoare Logic.
  - Temporal Logic formulae, e.g. **G**(**F** `available`), for Model Checking.

Logic and Proof reminder

For Hoare Logic:

- $\Gamma \models \phi$ (validity/truth)
  Every model which satisfies the formulae in Γ also satisfies formula ϕ.
- $\Gamma \vdash_R \phi$ (provability)
  Given a set R of axioms and rules there is a proof derivation of ϕ (perhaps using the assumptions in Γ).

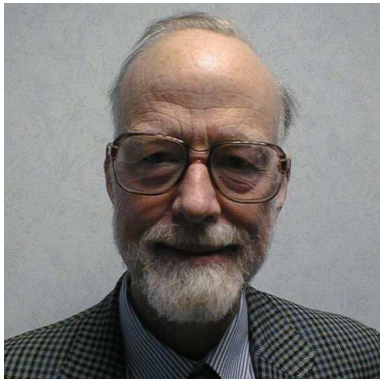
For Model Checking we instead use (note the clash of notation):

- $\mathcal{M} \models \phi$ (model satisfaction)
  Model $\mathcal{M}$ satisfies ϕ.

Part 1: Hoare Logic

Hoare Logic

- Program specification using Hoare notation
- Axioms and rules of Hoare Logic
- Soundness and completeness
- Mechanised program verification
- Pointers, the frame problem and separation logic



A Little Programming Language

Expressions:

$E ::= \quad N \mid V \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid \ldots$

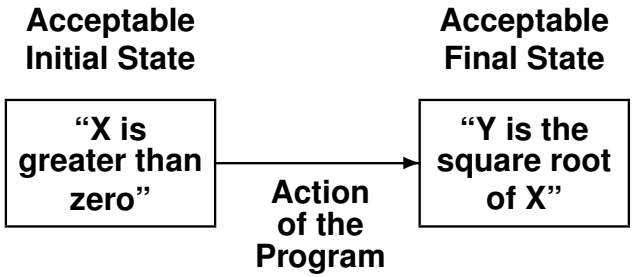
Boolean expressions:

$B ::= \quad \texttt{T} \mid \texttt{F} \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \ldots$

Commands:

$$\begin{array}{rl} C ::= & V := E \\ \mid & C_1 \texttt{ ; } C_2 \\ \mid & \texttt{IF } B \texttt{ THEN } C_1 \texttt{ ELSE } C_2 \\ \mid & \texttt{WHILE } B \texttt{ DO } C \end{array}$$

Specification of Imperative Programs



Hoare notation

- Tony Hoare introduced the following notation called a *partial correctness specification* for specifying what a program does:

$$\{P\}\ C\ \{Q\}$$

  where:
  - C is a command
  - P and Q are conditions on the program variables used in C
- Conditions on program variables will be written using standard mathematical notations together with *logical operators* like:
  - $\wedge$ ('and'), $\vee$ ('or'), $\neg$ ('not'), $\Rightarrow$ ('implies')
- Hoare's original notation was $P\ \{C\}\ Q$ not $\{P\}\ C\ \{Q\}$, but the latter form is now more widely used

Meaning of Hoare's Notation

- $\{P\}\ C\ \{Q\}$ is true (i.e. $\models \{P\}\ C\ \{Q\}$) if
  - whenever C is executed in a state satisfying P
  - and *if* the execution of C terminates
  - then the state in which C terminates satisfies Q
- Example: $\{\mathtt{X} = 1\}\ \mathtt{X{:=}X{+}1}\ \{\mathtt{X} = 2\}$
  - P is the condition that the value of `X` is `1`
  - Q is the condition that the value of `X` is `2`
  - C is the assignment command `X:=X+1`
- $\{\mathtt{X} = 1\}\ \mathtt{X{:=}X{+}1}\ \{\mathtt{X} = 2\}$ is true
- $\{\mathtt{X} = 1\}\ \mathtt{X{:=}X{+}1}\ \{\mathtt{X} = 3\}$ is false

Hoare Logic and Verification Conditions

- Hoare Logic is a deductive proof system for Hoare triples $\{P\}\ C\ \{Q\}$
- Can use Hoare Logic directly to verify programs
  - original proposal by Hoare
  - tedious and error prone
  - impractical for large programs
- Can 'compile' proving $\{P\}\ C\ \{Q\}$ to verification conditions
  - more natural
  - basis for computer assisted verification
- Proof of verification conditions equivalent to proof with Hoare Logic
  - Hoare Logic can be used to explain verification conditions

Partial Correctness Specification

- The formula $\{P\}\ C\ \{Q\}$ is called a *partial correctness specification*
  - P is called its *precondition*
  - Q its *postcondition*
- $\{P\}\ C\ \{Q\}$ is true (i.e. $\models \{P\}\ C\ \{Q\}$) if
  - whenever C is executed in a state satisfying P
  - and *if* the execution of C terminates
  - then the state in which C's execution terminates satisfies Q
- These specifications are 'partial' because for $\{P\}\ C\ \{Q\}$ to be true it is *not* necessary for the execution of C to terminate when started in a state satisfying P
- It is only required that *if* the execution terminates, *then* Q holds

Total Correctness Specification

- A stronger kind of specification is a *total correctness specification*
  - there is no standard notation for such specifications
  - we shall use $[P]\ C\ [Q]$
- $[P]\ C\ [Q]$ is true (i.e. $\models [P]\ C\ [Q]$) if
  - whenever C is executed in a state satisfying P the execution of C terminates
  - after C terminates Q holds
- $[\mathtt{X} = 1]\ \mathtt{Y{:=}X;\ WHILE\ T\ DO\ X := X}\ [\mathtt{Y} = 1]$
  - this says that the execution of `Y:=X;WHILE T DO X:=X` terminates when started in a state satisfying $\mathtt{X} = 1$
  - after which $\mathtt{Y} = 1$ will hold
  - this is clearly false

Total Correctness

- Informally: *Total correctness = Termination + Partial correctness*
- Total correctness is the ultimate goal
- usually easier to show partial correctness and termination separately
- Termination is usually straightforward to show, but there are examples where it is not: no one knows whether the program below terminates for all values of X

```
WHILE X>1 DO
  IF ODD(X) THEN X := 3*X + 1
            ELSE X := X DIV 2
```

  - DIV is C-style integer division
  - the Collatz conjecture is that this terminates with X=1
- Microsoft's T2 tool proves systems code terminates

Auxiliary Variables

- $\{X{=}x \wedge Y{=}y\}$ R:=X; X:=Y; Y:=R $\{X{=}y \wedge Y{=}x\}$
  - this says that *if* the execution of R:=X; X:=Y; Y:=R terminates (which it does)
  - *then* the values of X and Y are exchanged
- The variables x and y, which don't occur in the command and are used to name the initial values of program variables X and Y
- They are called *auxiliary* variables or *ghost* variables
- Informal convention:
  - program variable are upper case
  - auxiliary variable are lower case

More simple examples not lectured

- $\{X{=}x \wedge Y{=}y\}$ X:=Y; Y:=X $\{X{=}y \wedge Y{=}x\}$
  - this says that X:=Y; Y:=X exchanges the values of X and Y
  - this is not true
- $\{\mathtt{T}\}\ C\ \{Q\}$
  - this says that whenever C halts, Q holds
- $\{P\}\ C\ \{\mathtt{T}\}$
  - this specification is true for every condition P and every command C
  - because T is always true
- $[P]\ C\ [\mathtt{T}]$
  - this says that C terminates if initially P holds
  - it says nothing about the final state
- $[\mathtt{T}]\ C\ [P]$
- this says that C always terminates and ends in a state where P holds

A More Complicated Example not lectured

- $\{\mathtt{T}\}$

$$\left.\begin{array}{l}\mathtt{RM:=X;}\\ \mathtt{QU:=0;}\\ \mathtt{WHILE\ Y{\le}RM\ DO}\\ \quad\mathtt{(RM:=RM-Y;\ QU:=QU+1)}\end{array}\right\} C$$

$\{\mathtt{RM} < \mathtt{Y} \wedge \mathtt{X} = \mathtt{RM} + (\mathtt{Y} \times \mathtt{QU})\}$

- This is $\{\mathtt{T}\}\ C\ \{\mathtt{RM} < \mathtt{Y} \wedge \mathtt{X} = \mathtt{RM} + (\mathtt{Y} \times \mathtt{QU})\}$
  - where C is the command indicated by the braces above
  - the specification is true if whenever the execution of C halts, then QU is quotient and RM is the remainder resulting from dividing Y into X
  - it is true (even if X is initially negative!)
  - Using Q as a program variable can lead to confusion, hence QU and RM here.

Some Easy Exercises not lectured

- When is [T] *C* [T] true?
- Write a partial correctness specification which is true if and only if the command *C* has the effect of multiplying the values of X and Y and storing the result in X
- Write a specification which is true if the execution of *C* always halts when execution is started in a state satisfying *P*.
  (This implies that Hoare Logic is undecidable – see later.)

Specification can be Tricky

- "The program must set Y to the maximum of X and Y"
  - [T] *C* [Y = max(X,Y)]
- A suitable program:
  - `IF X >= Y THEN Y := X ELSE X := X`
- Another?
  - `IF X >= Y THEN X := Y ELSE X := X`
- Or even?
  - `Y := X`
- Later you will be able to prove that *all* these programs are "correct" which doesn't seem quite right
- The postcondition "`Y=max(X,Y)`" says "Y is the maximum of X and Y *in the final state*"

Specification can be Tricky (ii)

- The intended specification was probably *not* properly captured by

  $\vdash$ `{T} C {Y=max(X,Y)}`

- The correct formalisation of what was intended is probably

  $\vdash$ `{X=x` $\wedge$ `Y=y} C {Y=max(x,y)}`

- The lesson
  - it is easy to write the wrong specification!
  - a proof system will not help since the incorrect programs could have been proved "correct"
  - testing would have helped!

Review of Predicate Calculus

In first-order logic there are two separate syntactic classes

- Terms (or expressions): these denote values (e.g. numbers)

$$t ::= x \mid f(t_1, \ldots, t_n)$$

- Formulae (sometimes statements): these are either true or false

$$\phi ::= P(t_1, \ldots, t_n) \mid \phi \wedge \phi \mid \neg\phi \mid \forall x.\phi \qquad \text{etc.}$$

Hoare logic adjusts these by adding:

- $t ::= E$ (program expressions)
- $\phi ::= \{P\}\ C\ \{Q\} \mid B$ (program boolean expressions)

Floyd-Hoare Logic

- To construct formal proofs of partial correctness specifications,
  axioms and *rules of inference are needed*
- This is what Floyd-Hoare logic provides
  - the formulation of the deductive system is due to Hoare
  - some of the underlying ideas originated with Floyd
- A proof (or 'proof derivation') in Floyd-Hoare logic is a tree each of whose nodes and leaves are instances of the *rules* and *axioms* of the logic, and whose root (usually drawn at the bottom of the derivation!) is what we have proved.
- A formal proof makes explicit what axioms and rules of inference are used to arrive at a conclusion

Preview of Floyd-Hoare rules

$$\text{(ASS)}\ \frac{}{\vdash \{Q[E/V]\}\ V := E\ \{Q\}} \qquad \text{(SEQ)}\ \frac{\vdash \{P \wedge B\}\ C\ \{Q\} \quad \vdash \{P \wedge \neg B\}\ C'\ \{Q\}}{\vdash \{P\}\ \texttt{IF}\ B\ \texttt{THEN}\ C\ \texttt{ELSE}\ C'\ \{Q\}}$$

$$\text{(IF)}\ \frac{\vdash \{P\}\ C\ \{Q\} \quad \vdash \{Q\}\ C'\ \{R\}}{\vdash \{P\}\ C;\ C'\ \{R\}} \qquad \text{(WHILE)}\ \frac{\vdash \{P \wedge B\}\ C\ \{P\}}{\vdash \{P\}\ \texttt{WHILE}\ B\ \texttt{DO}\ C\ \{P \wedge \neg B\}}$$

$$\text{(PRE)}\ \frac{\vdash_{arith} P \Rightarrow P' \quad \vdash \{P'\}\ C\ \{Q\}}{\vdash \{P\}\ C\ \{Q\}} \qquad \text{(POST)}\ \frac{\vdash \{P\}\ C\ \{Q'\} \quad \vdash_{arith} Q' \Rightarrow Q}{\vdash \{P\}\ C\ \{Q\}}$$

Notes (we'll explain the rules in detail next):

- These are like typing rules – one for each syntactic form of the language ('syntax-directed') along with with additional glue rules (PRE) and (POST); these are a bit like sub-typing rules or rules for polymorphism.
- Note the references to rules of arithmetic $\vdash_{arith}$.
- We'll neither need assumptions Γ in $\Gamma \vdash \{P\}\ C\ \{Q\}$ nor use $\{P\}\ C\ \{Q\}$ other than (as implicitly universally quantified) at the top-level of a formula.

Reminder of our little programming language

Expressions

$E \quad ::= \quad N \mid V \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid \ldots$

Boolean expressions

$B \quad ::= \quad \texttt{T} \mid \texttt{F} \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \ldots$

Commands

C	$::=$	$V := E$	Assignments
	$\mid$	$C_1\ ;\ C_2$	Sequences
	$\mid$	$\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2$	Conditionals
	$\mid$	$\texttt{WHILE}\ B\ \texttt{DO}\ C$	WHILE-commands

Remark: The Floyd-Hoare rules constitute an *axiomatic semantics* of our programming language. (This is a third alternative to operational and denotational formulations.)

Judgements not lectured

- Three kinds of things that could be true or false:
  - formulae of mathematics, e.g. $(\mathrm{x}+1)^2 = \mathrm{x}^2 + 2 \times \mathrm{x} + 1$
  - partial correctness specifications $\{P\}\ C\ \{Q\}$
  - total correctness specifications $[P]\ C\ [Q]$
- These three kinds of things are examples of *judgements*
  - a logical system gives rules for proving judgements
  - Floyd-Hoare logic provides rules for proving partial correctness specifications
  - the laws of arithmetic provide ways of proving formulae about integers
- $\vdash S$ means formula S can be proved
  - how to prove predicate calculus formulae assumed known
  - this course covers axioms and rules for proving *program correctness formulae*

Syntactic Conventions not lectured

- Symbols V, V_1, ... , V_n stand for program variables
  - examples of particular variables are $\mathtt{X}$, $\mathtt{R}$, $\mathtt{Q}$ etc (using $\mathtt{Q}$ can be confusing!).
- Symbols x, x', y stand for auxiliary (mathematical) variables
- Symbols E, E_1, ... , E_n stand for arbitrary expressions (or terms)
  - these are things like $\mathtt{X}+1$, $\sqrt{2}$ etc. which denote values (usually numbers)
- Symbols S, S_1, ... , S_n stand for arbitrary formulae
  - these are conditions like $\mathtt{X} < \mathtt{Y}$, $\mathtt{X}^2 = 1$ etc. which are either true or false
  - will also use P, Q, R to range over pre and postconditions
- Symbols C, C_1, ... , C_n stand for arbitrary commands

Substitution Notation reminder not lectured

- $Q[E/V]$ is the result of replacing all occurrences of (program variable) V in formula Q by term E
  - read $Q[E/V]$ as 'Q with E substituted for V'
  - for example: $(\mathtt{X}+1 > \mathtt{X})[\mathtt{Y}+\mathtt{Z}/\mathtt{X}] = ((\mathtt{Y}+\mathtt{Z})+1 > \mathtt{Y}+\mathtt{Z})$
  - In this course we won't have local variable bindings so don't have to worry about variable capture)
  - In this course we will only use substitution on *program variables* not *auxiliary variables*
  - Same notation for substituting into terms, e.g. $E_1[E_2/V]$

The Assignment Axiom (Hoare)

- **Syntax:** $V := E$
- **Semantics:** value of V in final state is value of E in initial state
- **Example:** $\mathtt{X}:=\mathtt{X}+1$ (adds one to the value of the variable $\mathtt{X}$)

The Assignment Axiom

$$\vdash \{Q[E/V]\}\ V:=E\ \{Q\}$$

for any variable V, expression E and formula Q.

- Instances of the assignment axiom are
  - $\vdash \{\mathtt{E}=\mathtt{x}\}\ \mathtt{V}:=\mathtt{E}\ \{\mathtt{V}=\mathtt{x}\}$
  - $\vdash \{\mathtt{Y}=2\}\ \mathtt{X}:=2\ \{\mathtt{Y}=\mathtt{X}\}$
  - $\vdash \{\mathtt{X}+1=\mathtt{n}+1\}\ \mathtt{X}:=\mathtt{X}+1\ \{\mathtt{X}=\mathtt{n}+1\}$
  - $\vdash \{E=E\}\ \mathtt{X}:=E\ \{\mathtt{X}=E\}$ (if $\mathtt{X}$ does not occur in E)

The Backwards Fallacy

- Many people feel the assignment axiom is 'backwards'
- One common erroneous intuition is that it should be

$$\vdash \{P\}\ V:=E\ \{P[V/E]\}$$

  - which isn't really a proper substitution
  - this has the false consequence $\vdash \{\mathtt{X}=0\}\ \mathtt{X}:=1\ \{\mathtt{X}=0\}$ (since $(\mathtt{X}=0)[\mathtt{X}/1]$ equals $\mathtt{X}=0$ (1 doesn't occur in $\mathtt{X}=0$))
- Another erroneous intuition is that it should be

$$\vdash \{P\}\ V:=E\ \{P[E/V]\}$$

  - this has the false consequence $\vdash \{\mathtt{X}=0\}\ \mathtt{X}:=1\ \{1=0\}$ (got by taking P to be $\mathtt{X}=0$, V to be $\mathtt{X}$ and E to be 1)

Validity

- Important to establish the validity of axioms and rules
- Later will give a *formal semantics* of our little programming language
  - then *prove* axioms and rules of inference of Floyd-Hoare logic are sound
  - this will only increase our confidence in the axioms and rules to the extent that we believe the correctness of the formal semantics!
- The Assignment Axiom is not valid for 'real' programming languages
  - In an early PhD on Hoare Logic G. Ligler showed that the assignment axiom can fail to hold in six different ways for the language Algol 60

Expressions with Side-effects (just say 'no')

- The validity of the assignment axiom depends on expressions not having side effects
- Reason 1. It would break substitution in the assignment rule
- Reason 2. Suppose that our language had a C-like comma-expression:

$$\texttt{((Y:=1), 2)}$$

  - this expression has value `2`, but its evaluation also 'side effects' the variable `Y` by storing `1` in it
- If the assignment axiom applied to comma expressions, then it could be used to deduce

$$\vdash \{\texttt{Y=0}\}\ \texttt{X := ((Y:=1), 2)}\ \{\texttt{Y=0}\}$$

  - since $\texttt{(Y=0)[E/X] = (Y=0)}$ as $\texttt{Y=0}$ does not contain $\texttt{X}$
  - this is unsound; after the assignment $\texttt{Y=1}$

Floyd's Forwards Assignment Axiom not examinable

The original semantics of assignment due to Floyd:

- $\vdash \{P\}\ V\texttt{:=}E\ \{\exists v.\ V = E[v/V]\ \wedge\ P[v/V]\}$
  - where v is a new auxiliary variable (i.e. doesn't equal V or occur in P or E)
- Example instance
  $\vdash \{\texttt{X=1}\}\ \texttt{X:=X+1}\ \{\exists v.\ \texttt{X} = \texttt{X+1}[v/\texttt{X}]\ \wedge\ \texttt{X=1}[v/\texttt{X}]\}$
- Simplifying the postcondition
  $\vdash \{\texttt{X=1}\}\ \texttt{X:=X+1}\ \{\exists v.\ \texttt{X} = \texttt{X+1}[v/\texttt{X}]\ \wedge\ \texttt{X=1}[v/\texttt{X}]\}$
  $\vdash \{\texttt{X=1}\}\ \texttt{X:=X+1}\ \{\exists v.\ \texttt{X} = v + 1\ \wedge\ v = 1\}$
  $\vdash \{\texttt{X=1}\}\ \texttt{X:=X+1}\ \{\exists v.\ \texttt{X} = 1 + 1\ \wedge\ v = 1\}$
  $\vdash \{\texttt{X=1}\}\ \texttt{X:=X+1}\ \{\texttt{X} = 1 + 1\ \wedge\ \exists v.\ v = 1\}$
  $\vdash \{\texttt{X=1}\}\ \texttt{X:=X+1}\ \{\texttt{X} = 2\ \wedge\ \texttt{T}\}$
  $\vdash \{\texttt{X=1}\}\ \texttt{X:=X+1}\ \{\texttt{X} = 2\}$

Forwards Axiom equivalent to standard one but harder to use

Hoare Logic – Axioms and Rules

The Assignment Axiom (reprise)

The Assignment Axiom

$$\vdash \{Q[E/V]\}\ V := E\ \{Q\}$$

for any variable V, expression E and formula Q.

Precondition Strengthening

Precondition strengthening

$$\frac{\vdash P \Rightarrow P' \qquad \vdash \{P'\}\ C\ \{Q\}}{\vdash \{P\}\ C\ \{Q\}}$$

- Note the two hypotheses are different kinds of judgements. You may prefer to write $\vdash_{arith}$ for the first one.

Example

Here we're using an auxiliary (mathematical) variable $\mathtt{n}$ instead of using a specific number like 42.

- The assignment axiom allows us to deduce $\vdash \{\mathtt{X}+1=\mathtt{n}+1\}\ \mathtt{X} := \mathtt{X}+1\ \{\mathtt{X}=\mathtt{n}+1\}$ (but *cannot* prove $\vdash \{\mathtt{X}=\mathtt{n}\}\ \mathtt{X} := \mathtt{X}+1\ \{\mathtt{X}=\mathtt{n}+1\}$)
- But we have $\vdash_{arith} \{\mathtt{X}=\mathtt{n}\} \Rightarrow \{\mathtt{X}+1=\mathtt{n}+1\}$.
- Combining the two previous facts with the precondition-strengthening rule gives the desired $\vdash \{\mathtt{X}=\mathtt{n}\}\ \mathtt{X} := \mathtt{X}+1\ \{\mathtt{X}=\mathtt{n}+1\}$

In other words – precondition strengthening acts as a *glue* rule.

Exercise: prove $\vdash \{\mathtt{X}=\mathtt{n}-1\}\ \mathtt{X} := \mathtt{X}+1\ \{\mathtt{X}=\mathtt{n}\}$

Postcondition weakening

Just as the previous rule allows the precondition of a partial correctness specification to be *strengthened*, the following one allows us to *weaken* the postcondition:

Postcondition weakening

$$\frac{\vdash \{P\}\ C\ \{Q'\} \qquad \vdash Q' \Rightarrow Q}{\vdash \{P\}\ C\ \{Q\}}$$

Also acts as a *glue* rule.

An Example Formal Proof

- Here is a little formal proof (exercise: draw this as a proof tree):

1.	$\vdash$ {R=X ∧ 0=0} Q:=0 {R=X ∧ Q=0}	By the assignment axiom
2.	$\vdash_{arith}$ R=X ⇒ R=X ∧ 0=0	By pure logic
3.	$\vdash$ {R=X} Q:=0 {R=X ∧ Q=0}	By precondition strengthening
4.	$\vdash_{arith}$ R=X ∧ Q=0 ⇒ R=X+(Y × Q)	By laws of arithmetic
5.	$\vdash$ {R=X} Q:=0 {R=X+(Y × Q)}	By postcondition weakening

- The rules precondition strengthening and postcondition weakening are sometimes called the *rules of consequence*

The sequencing rule

- **Syntax:** C_1 ; ⋯ ; C_n
- **Semantics:** the commands $C_1, \cdots, C_n$ are executed in that order
- **Example:** R:=X; X:=Y; Y:=R
  - the values of X and Y are swapped using R as a temporary variable
  - note *side effect*: value of R changed to the old value of X

The sequencing rule

$$\frac{\vdash \{P\}\ C_1\ \{Q\}, \qquad \vdash \{Q\}\ C_2\ \{R\}}{\vdash \{P\}\ C_1 ; C_2\ \{R\}}$$

Example Proof

By the assignment axiom:

i $\vdash$ {X=x∧Y=y} R:=X {R=x∧Y=y}

ii $\vdash$ {R=x∧Y=y} X:=Y {R=x∧X=y}

iii $\vdash$ {R=x∧X=y} Y:=R {Y=x∧X=y}

Hence by (i), (ii) and the sequencing rule

iv $\vdash$ {X=x∧Y=y} R:=X; X:=Y {R=x∧X=y}

Hence by (iv) and (iii) and the sequencing rule

v $\vdash$ {X=x∧Y=y} R:=X; X:=Y; Y:=R {Y=x∧X=y}

(which is what we expect).

Conditionals

- **Syntax:** IF S THEN C_1 ELSE C_2
- **Semantics:**
  - if the statement S is true in the current state, then C_1 is executed
  - if S is false, then C_2 is executed
- **Example:** IF X<Y THEN MAX:=Y ELSE MAX:=X
  - the value of the variable MAX it set to the maximum of the values of X and Y

The Conditional Rule

The conditional rule

$$\frac{\vdash \{P \wedge S\}\ C_1\ \{Q\}, \qquad \vdash \{P \wedge \neg S\}\ C_2\ \{Q\}}{\vdash \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}}$$

- From Assignment Axiom + Precondition Strengthening and

  $\vdash \quad (X \geq Y \Rightarrow X = max(X,Y)) \wedge (\neg(X \geq Y) \Rightarrow Y = max(X,Y))$

  it follows that

  $\vdash$ {T $\wedge$ X$\geq$Y} MAX:=X {MAX=max(X,Y)}

  and

  $\vdash$ {T $\wedge$ $\neg$(X$\geq$Y)} MAX:=Y {MAX=max(X,Y)}

- Then by the conditional rule it follows that

  $\vdash$ {T} IF X$\geq$Y THEN MAX:=X ELSE MAX:=Y {MAX=max(X,Y)}

WHILE-commands

- **Syntax:** WHILE S DO C
- **Semantics:**
  - if the statement S is true in the current state, then C is executed and the WHILE-command is repeated
  - if S is false, then nothing is done
  - thus C is repeatedly executed until the value of S becomes false
  - if S never becomes false, then the execution of the command never terminates
- **Example:** WHILE $\neg$(X=0) DO X:= X−2
  - if the value of X is non-zero, then its value is decreased by 2 and then the process is repeated
- This WHILE-command will terminate (with X having value 0) if the value of X is an even non-negative number
  - in all other states it will not terminate

Invariants

- Suppose $\vdash \{P \wedge S\}\ C\ \{P\}$
- P is said to be an *invariant* of C whenever S holds
- The WHILE-rule says that
  - if P is an invariant of the body of a WHILE-command whenever the test condition holds
  - then P is an invariant of the whole WHILE-command
- In other words
  - if executing C *once* preserves the truth of P
  - then executing C *any number of times* also preserves the truth of P
- The WHILE-rule also expresses the fact that after a WHILE-command has terminated, the test must be false
  - otherwise, it wouldn't have terminated

The WHILE-Rule

The WHILE-rule

$$\frac{\vdash \{P \wedge S\}\ C\ \{P\}}{\vdash \{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ \{P \wedge \neg S\}}$$

- It is easy to show

  $\vdash$ {X=R+(Y×Q)$\wedge$Y$\leq$R} R:=R−Y; Q:=Q+1 {X=R+(Y×Q)}

- Hence by the WHILE-rule with P = 'X=R+(Y×Q)' and S = 'Y$\leq$R'

  $\vdash$ {X=R+(Y×Q)}
  WHILE Y$\leq$R DO
  (R:=R−Y; Q:=Q+1)
  {X=R+(Y×Q) $\wedge$ $\neg$(Y$\leq$R)}

Example

- From the previous slide

$$\vdash \{X=R+(Y\times Q)\}$$
$$\quad \texttt{WHILE } Y\leq R \texttt{ DO}$$
$$\quad\quad (R:=R-Y;\ Q:=Q+1)$$
$$\{X=R+(Y\times Q) \wedge \neg(Y\leq R)\}$$

- It is easy to deduce that

$$\vdash \{T\}\ R:=X;\ Q:=0\ \{X=R+(Y\times Q)\}$$

- Hence by the sequencing rule and postcondition weakening

$$\vdash \{T\}$$
$$\quad R:=X;$$
$$\quad Q:=0;$$
$$\quad \texttt{WHILE } Y\leq R \texttt{ DO}$$
$$\quad\quad (R:=R-Y;\ Q:=Q+1)$$
$$\{R<Y \wedge X=R+(Y\times Q)\}$$

Summary

- We have given:
  - a notation for specifying what a program does
  - a way of proving that it meets its specification
- Now we look at ways of finding proofs and organising them:
  - finding invariants
  - derived rules
  - backwards proofs
  - annotating programs prior to proof
- Then we see how to automate program verification
  - the automation mechanises some of these ideas

How does one find an invariant?

The `WHILE`-rule

$$\frac{\vdash \{P \wedge S\}\ C\ \{P\}}{\vdash \{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ \{P \wedge \neg S\}}$$

- Look at the facts:
  - invariant P must hold initially
  - with the negated test $\neg S$ the invariant P must establish the result
  - when the test S holds, the body must leave the invariant P unchanged
- Think about how the loop works – the invariant should say that:
  - what has been done so far together with what remains to be done
  - holds at each iteration of the loop
  - and gives the desired result when the loop terminates

Example

- Consider a factorial program

$$\{X=n \wedge Y=1\}$$
$$\quad \texttt{WHILE } X\neq 0 \texttt{ DO}$$
$$\quad\quad (Y:=Y\times X;\ X:=X-1)$$
$$\{X=0 \wedge Y=n!\}$$

- Look at the facts
  - initially `X=n` and `Y=1`
  - finally `X=0` and `Y=n!`
  - on each loop `Y` is increased and, `X` is decreased
- Think how the loop works
  - `Y` holds the result so far
  - `X!` is what remains to be computed
  - `n!` is the desired result
- The invariant is `X!`$\times$`Y = n!`
  - 'stuff to be done' $\times$ 'result so far' = 'desired result'
  - decrease in `X` combines with increase in `Y` to make invariant

Related example

```
{X=0 ∧ Y=1}
 WHILE X<N DO (X:=X+1; Y:=Y×X)
{Y=N!}
```

- Look at the Facts
  - initially `X=0` and `Y=1`
  - finally `X=N` and `Y=N!`
  - on each iteration both `X` an `Y` increase: `X` by `1` and `Y` by `X`
- An invariant is `Y = X!`
- At end need `Y = N!`, but `WHILE`-rule only gives $\neg$(`X<N`)
- Ah Ha! Invariant needed: $\boxed{\texttt{Y = X!} \wedge \texttt{X}\leq\texttt{N}}$
- At end $\texttt{X} \leq \texttt{N} \wedge \neg(\texttt{X<N}) \Rightarrow \texttt{X=N}$
- Often need to strengthen invariants to get them to work
  - typical to add stuff to 'carry along' like `X`$\leq$`N`

Conjunction and Disjunction

Specification conjunction

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \qquad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \wedge P_2\}\ C\ \{Q_1 \wedge Q_2\}}$$

Specification disjunction

$$\frac{\vdash \{P_1\}\ C\ \{Q_1\}, \qquad \vdash \{P_2\}\ C\ \{Q_2\}}{\vdash \{P_1 \vee P_2\}\ C\ \{Q_1 \vee Q_2\}}$$

- Useful for splitting a proof into independent bits
  - they enable $\{P\}\ C\ \{Q_1 \wedge Q_2\}$ to be proved by proving separately that both $\{P\}\ C\ \{Q_1\}$ and also that $\{P\}\ C\ \{Q_2\}$
- Any proof with these rules could be done without using them
  - i.e. they are theoretically redundant (proof omitted)
  - however, useful in practice

Derived rules for finding proofs

- Suppose the goal is to prove
  $\{\mathit{Precondition}\}\ \mathit{Command}\ \{\mathit{Postcondition}\}$
- If there were a rule of the form

$$\frac{\vdash H_1, \cdots, \vdash H_n}{\vdash \{P\}\ C\ \{Q\}}$$

  then we could instantiate (think Prolog)

  $P \mapsto \mathit{Precondition}$, $C \mapsto \mathit{Command}$, $Q \mapsto \mathit{Postcondition}$

  to get instances of $H_1, \cdots, H_n$ as subgoals
- Some of the rules are already in this form e.g. the sequencing rule
- We will derive rules of this form for all commands
- Then we use these derived rules for mechanising Hoare Logic proofs

Derived Rules

We will establish derived rules of the following form:

$$\frac{\cdots}{\vdash \{P\}\ V:=E\ \{Q\}}$$

$$\frac{\cdots}{\vdash \{P\}\ C_1; C_2\ \{Q\}}$$

$$\frac{\cdots}{\vdash \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}}$$

$$\frac{\cdots}{\vdash \{P\}\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ \{Q\}}$$

- These support 'backwards proof' starting from a goal $\{P\}\ C\ \{Q\}$
- Useful intuition: think of Hoare-Logic rules as a 3-argument Prolog predicate. This just means the P and Q parameter are variables which match everything, and are not restricted to special forms $P \wedge \neg B$.

The Derived Assignment Rule

- An example proof
  1. $\vdash \{\mathtt{R{=}X} \wedge \mathtt{0{=}0}\}\ \mathtt{Q:=0}\ \{\mathtt{R{=}X} \wedge \mathtt{Q{=}0}\}$ By the assignment axiom.
  2. $\vdash \mathtt{R{=}X} \Rightarrow \mathtt{R{=}X} \wedge \mathtt{0{=}0}$ By pure logic.
  3. $\vdash \{\mathtt{R{=}X}\}\ \mathtt{Q:=0}\ \{\mathtt{R{=}X} \wedge \mathtt{Q{=}0}\}$ By precondition strengthening.
- Can generalise this proof to a proof schema:
  1. $\vdash \{Q[E/V]\}\ \mathrm{V}:=\mathrm{E}\ \{Q\}$ By the assignment axiom.
  2. $\vdash P \Rightarrow Q[E/V]$ By assumption.
  3. $\vdash \{P\}\ \mathrm{V}:=\mathrm{E}\ \{Q\}$ By precondition strengthening.
- This proof schema justifies:

Derived Assignment Rule

$$\frac{\vdash P \Rightarrow Q[E/V]}{\vdash \{P\}\ V:=E\ \{Q\}}$$

- Note: $Q[E/V]$ is the weakest liberal precondition $\mathrm{wlp}(V:=E, Q)$ – see later.
- Example proof above can now be done in one less step
  1. $\vdash \mathtt{R{=}X} \Rightarrow \mathtt{R{=}X} \wedge \mathtt{0{=}0}$ By pure logic.
  2. $\vdash \{\mathtt{R{=}X}\}\ \mathtt{Q:=0}\ \{\mathtt{R{=}X} \wedge \mathtt{Q{=}0}\}$ By derived assignment.

Derived Sequenced Assignment Rule

- The following rule will be useful later

Derived Sequenced Assignment Rule

$$\frac{\vdash \{P\}\ C\ \{Q[E/V]\}}{\vdash \{P\}\ C; V:=E\ \{Q\}}$$

- Intuitively work backwards:
  - push Q 'through' $V:=E$, changing it to $Q[E/V]$
- Example: By the assignment axiom:
- $\vdash \{X{=}x \wedge Y{=}y\}\ R:=X\ \{R{=}x \wedge Y{=}y\}$
- Hence by the sequenced assignment rule
- $\vdash \{X{=}x \wedge Y{=}y\}\ R:=X;\ X:=Y\ \{R{=}x \wedge X{=}y\}$

The Derived While Rule

Derived While Rule

$$\frac{\vdash P \Rightarrow R \quad \vdash \{R \wedge S\}\ C\ \{R\} \quad \vdash R \wedge \neg S \Rightarrow Q}{\vdash \{P\}\ \mathtt{WHILE}\ S\ \mathtt{DO}\ C\ \{Q\}}$$

This follows from the While Rule and the rules of consequence.
As an example: it is easy to show

- $\vdash$ R=X $\wedge$ Q=0 $\Rightarrow$ X=R+(Y$\times$Q)
- $\vdash$ {X=R+(Y$\times$Q)$\wedge$Y$\leq$R} R:=R-Y; Q:=Q+1 {X=R+(Y$\times$Q)}
- $\vdash$ X=R+(Y$\times$Q)$\wedge\neg$(Y$\leq$R) $\Rightarrow$ X=R+(Y$\times$Q)$\wedge\neg$(Y$\leq$R)

Then, by the derived While rule

$\vdash$ {R=X $\wedge$ Q=0}
WHILE Y$\leq$R DO (R:=R-Y; Q:=Q+1)
{X=R+(Y$\times$Q) $\wedge$ $\neg$(Y$\leq$R)}

The Derived Sequencing Rule

- The rule below follows from the sequencing and consequence rules

The Derived Sequencing Rule

$$\frac{\begin{array}{ll}\vdash P \Rightarrow P_1 & \\ \vdash \{P_1\}\ C_1\ \{Q_1\} & \vdash Q_1 \Rightarrow P_2 \\ \vdash \{P_2\}\ C_2\ \{Q_2\} & \vdash Q_2 \Rightarrow P_3 \\ \ldots & \ldots \\ \vdash \{P_n\}\ C_n\ \{Q_n\} & \vdash Q_n \Rightarrow Q\end{array}}{\vdash \{P\}\ C_1; \ldots; C_n\ \{Q\}}$$

- Exercise: why no derived conditional rule?

Forwards and backwards proof

- Previously it was shown how to prove $\{P\}C\{Q\}$ by
  - proving properties of the components of C and
  - then putting these together, with appropriate proof rules, to get the desired property of C
- For example, to prove $\vdash \{P\}C_1 ; C_2\{Q\}$
  - First prove $\vdash \{P\}C_1\{R\}$ and $\vdash \{R\}C_2\{Q\}$
  - then deduce $\vdash \{P\}C_1 ; C_2\{Q\}$ by sequencing rule
- This method is called *forward proof*
  - move forward from axioms via rules to conclusion
- The problem with forwards proof is that it is not always easy to see what you need to prove to get where you want to be
- It is often more natural to work backwards (think Prolog):
  - starting from the goal of showing $\{P\}C\{Q\}$
  - generate subgoals until problem solved

Mechanised verification

NEW TOPIC: Mechanising Program Verification

- The architecture of a simple program verifier will be described
- Justified with respect to the rules of Floyd-Hoare logic
- It is clear that
  - proofs are long and boring, even if the program being verified is quite simple
  - lots of fiddly little details to get right, many of which are trivial, e.g.

$$\vdash (\mathtt{R}{=}\mathtt{X} \wedge \mathtt{Q}{=}0) \Rightarrow (\mathtt{X} = \mathtt{R} + \mathtt{Y}{\times}\mathtt{Q})$$

Mechanisation

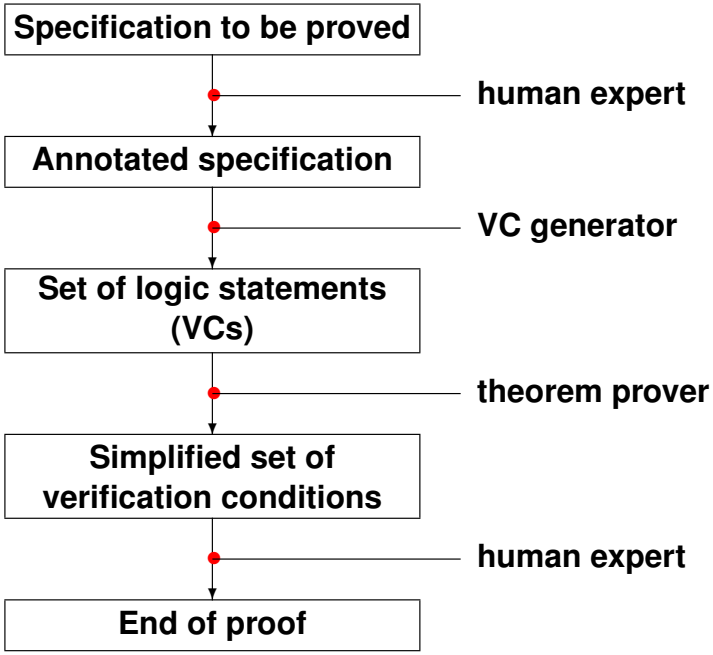
Goal: automate the routine bits of proofs in Floyd-Hoare logic

- Unfortunately, logicians have shown that it is impossible in principle to design a *decision procedure* to decide automatically the truth or falsehood of an arbitrary mathematical statement
- This does not mean that one cannot have procedures that will prove many *useful* theorems
  - the non-existence of a general decision procedure merely shows that one cannot prove *everything* automatically
  - in practice, it is quite possible to build a system that will mechanise the boring and routine aspects of verification

The standard approach to this will be described in the course

- ideas very old (JC King's 1969 CMU PhD, Stanford verifier in 1970s)
- used by program verifiers (e.g. Gypsy and SPARK verifier)
- provides a verification front end to different provers (see *Why* system)

Architecture of a Verifier



Commentary

- Input: a Hoare triple annotated with mathematical statements (formulae)
  - these annotations describe relationships between variables
  - think C-like `assert()` but checked before run time
- The system generates a set of purely mathematical statements called *verification conditions* (or VCs)
- If the verification conditions are provable ($\vdash_{arith}$) then the original specification can be deduced ($\vdash$) from the axioms and rules of Hoare logic
- The verification conditions are passed to a *theorem prover* program which attempts to prove them automatically
  - if it fails, advice is sought from the user

Verification conditions

The three steps in proving $\{P\}C\{Q\}$ with a verifier:

1. The program C is *annotated* by inserting statements (*assertions*) expressing conditions that are meant to hold at intermediate points
   - tricky: needs intelligence and good understanding of how the program works
   - automating it is an artificial intelligence problem
2. A set of logic statements called *verification conditions* (VCs) is then generated from the annotated specification
   - this is purely mechanical and easily done by a program
3. The verification conditions are proved
   - needs automated theorem proving

- To improve automated verification one can try to
  - reduce the number and complexity of the annotations required
  - increase the power of the theorem prover
  - still a research area

Validity of Verification Conditions

- It will be shown that
  - if one can prove all the verification conditions generated from $\{P\}C\{Q\}$
  - then $\vdash \{P\}C\{Q\}$
- Step 2 converts a verification problem into a conventional mathematical problem
- The process will be illustrated with:

```
{T}
    R:=X;
    Q:=0;
    WHILE Y≤R DO
       (R:=R−Y; Q:=Q+1)
{X = R+Y×Q ∧ R<Y}
```

- Beware the difference between `Q` (program variable) and Q (postcondition).

Example

- Step **1** is to insert annotations (formulae) ϕ_1 and ϕ_2

```
{T}
    R:=X;
    Q:=0;  {R=X ∧ Q=0}  ⟵ φ1
    WHILE Y≤R DO {X = R+Y×Q}  ⟵ φ2
       (R:=R-Y;  Q:=Q+1)
{X = R+Y×Q ∧ R<Y}
```

- The annotations ϕ_1 and ϕ_2 state conditions which are intended to hold *whenever* control reaches them
- Control only reaches the point at which ϕ_1 is placed once
- It reaches ϕ_2 each time the `WHILE` body is executed
  - whenever this happens `X=R+Y×Q` holds, even though the values of `R` and `Q` vary
  - ϕ_2 is an *invariant* of the `WHILE`-command

Generating and Proving Verification Conditions

- Step **2** will generate the following four verification conditions
  - i `T ⇒ (X=X ∧ 0=0)`
  - ii `(R=X ∧ Q=0) ⇒ (X = R+(Y×Q))`
  - iii `(X = R+(Y×Q)) ∧ Y≤R) ⇒ (X = (R-Y)+(Y×(Q+1)))`
  - iv `(X = R+(Y×Q)) ∧ ¬(Y≤R) ⇒ (X = R+(Y×Q) ∧ R<Y)`
- Notice that these are statements of arithmetic
  - the constructs of our programming language have been 'compiled away'
- Step **3** consists in proving the four verification conditions
  - easy with modern automatic theorem provers

Annotation of Commands

- An annotated command is a command with statements (*assertions*) embedded within it
- A command is *properly annotated* if statements have been inserted at the following places
  - i before C_2 in C_1 ; C_2 if C_2 is *not* an assignment command
  - ii after the word `DO` in `WHILE` commands
- The inserted assertions should express the conditions one expects to hold *whenever* control reaches the point at which the assertion occurs
- Can reduce number of annotations using weakest preconditions
  (see later)

Annotation of Specifications

- A properly annotated specification is a specification $\{P\}C\{Q\}$ where C is a properly annotated command
- Example: To be properly annotated, assertions should be at points ℓ_1 and ℓ_2 of the specification below

```
{X = n}
    Y:=1;  ⟵ ℓ1
    WHILE X≠0 DO  ⟵ ℓ2
       (Y:=Y×X;  X:=X-1)
{X = 0  ∧  Y = n!}
```

- Suitable statements would be
  - ℓ_1: {`Y = 1 ∧ X = n`}
  - ℓ_2: {`Y × X! = n!`}

Verification Condition Generation

- The VCs generated from an annotated specification $\{P\}C\{Q\}$ are obtained by considering the various possibilities for C
- We will describe it command by command using rules of the form:
- The VCs for $C\langle C_1, C_2\rangle$ are
  - vc_1, ... , vc_n generated by C itself
  - together with the VCs for its subphrases C_1 and C_2
- Each VC rule corresponds to either a primitive or derived rule

A VC Generation Program

- The algorithm for generating verification conditions is *recursive* on the structure of commands
- The rule just given corresponds to a simple recursive function:

  VC $(C\langle C_1, C_2\rangle)$ **=** $[vc_1$**, ... ,**$vc_n]$ **@ (VC** C_1**) @ (VC** C_2**)**
- The rules are chosen so that only one VC rule applies in each case
  - applying them is then purely mechanical
  - the choice is based on the syntax
  - only one rule applies in each case so VC generation is deterministic

Justification of VCs

- This process will be justified by showing that

  $$\vdash \{P\}C\{Q\}$$

  if all the verification conditions can be proved
- We will prove that for any C
  - assuming the VCs of $\{P\}C\{Q\}$ are provable ($\vdash_{arith}$)
  - then $\vdash \{P\}C\{Q\}$ is a theorem of the logic

Justification of Verification Conditions

- The argument that the verification conditions are sufficient will be by *induction* on the structure of C
- Such inductive arguments have two parts
  - show the result holds for atomic commands, i.e. assignments
  - show that when C is not an atomic command, then if the result holds for the constituent commands of C (this is called the *induction hypothesis*), then it holds also for C
- The first of these parts is called the *basis* of the induction
- The second is called the *step*
- The basis and step entail that the result holds for all commands

VC for Assignments

Assignment commands

The single verification condition generated by

$$\{P\}\ V := E\ \{Q\}$$

is

$$P \Rightarrow Q[E/V]$$

- Example: The verification condition for

  `{X=0} X:=X+1 {X=1}`

  is

  `X=0` $\Rightarrow$ `(X+1)=1` *(true by arithmetic)*

- Note: $Q[E/V] = \mathrm{wlp}(V := E, Q)$ (see later)

Justification of Assignment VC

- We must show that if the VCs of $\{P\}\ V := E\ \{Q\}$ are provable
  then $\vdash\ \{P\}\ V := E\ \{Q\}$
- Proof:
  - Assume $\vdash\ P \Rightarrow Q[E/V]$ as it is the VC
  - From derived assignment rule it follows that
    $\vdash\ \{P\}\ V := E\ \{Q\}$

VCs for Conditionals

Conditionals

The verification conditions generated from

$$\{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}$$

are

i the verification conditions generated by
$\{P \wedge S\}\ C_1\ \{Q\}$

ii the verifications generated by
$\{P \wedge \neg S\}\ C_2\ \{Q\}$

Example: The verification conditions for

`{T} IF X`$\geq$`Y THEN R:=X ELSE R:=Y {R=max(X,Y)}`

are

i the VCs for `{T` $\wedge$ `X`$\geq$`Y} R:=X {R=max(X,Y)}`

ii the VCs for `{T` $\wedge$ $\neg$`(X`$\geq$`Y)} R:=Y {R=max(X,Y)}`

Justification for the Conditional VCs (1)

- Must show that if VCs of
  $\{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}$
  are provable, then
  $\vdash\ \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}$
- Proof:
  - Assume the VCs $\{P \wedge S\}\ C_1\ \{Q\}$ and $\{P \wedge \neg S\}\ C_2\ \{Q\}$
  - The inductive hypotheses tell us that if these VCs are provable then the corresponding Hoare Logic theorems are provable
  - i.e. by induction $\vdash\ \{P \wedge S\}\ C_1\ \{Q\}$ and
    $\vdash\ \{P \wedge \neg S\}\ C_2\ \{Q\}$
  - Hence by the conditional rule
    $\vdash\ \{P\}\ \texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}$

Review of Annotated Sequences

- If $C_1 ; C_2$ is properly annotated, then either

  Case 1: it is of the form $C_1 ; \{R\} C_2$ and C_2 is not an assignment

  Case 2: it is of the form $C ; V := E$

- And C, C_1 and C_2 are properly annotated

VCs for Sequences

Sequences

1. The verification conditions generated by

   $$\{P\}\ C_1\ \{R\}\ C_2\ \{Q\}$$

   (where C_2 is not an assignment) are the union of:
   - i the verification conditions generated by $\{P\}\ C_1\ \{R\}$
   - ii the verifications generated by $\{R\}\ C_2\ \{Q\}$
2. The verification conditions generated by

   $$\{P\}\ C ; V := E\ \{Q\}$$

   are the verification conditions generated by

   $$\{P\}\ C\ \{Q[E/V]\}$$

Example

- The verification conditions generated from

  ```
  {X=x ∧ Y=y} R:=X; X:=Y; X:=R
  {X=y ∧ R=x}
  ```

- Are those generated by

  ```
  {X=x ∧ Y=y} R:=X; X:=Y
  {(X=y ∧ Y=x)[R/Y]}
  ```

- This simplifies to

  ```
  {X=x ∧ Y=y} R:=X; X:=Y {X=y ∧ R=x}
  ```

- The verification conditions generated by this are those generated by

  ```
  {X=x ∧ Y=y} R:=X {(X=y ∧ R=x)[Y/X]}
  ```

- Which simplifies to

  ```
  {X=x ∧ Y=y} R:=X {Y=y ∧ R=x}
  ```


Example Continued

- The only verification condition generated by

  ```
  {X=x ∧ Y=y} R:=X {Y=y ∧ R=x}
  ```

  is

  ```
  X=x ∧ Y=y ⇒ (Y=y ∧ R=x)[X/R]
  ```

- Which simplifies to

  ```
  X=x ∧ Y=y ⇒ Y=y ∧ X=x
  ```

- Thus the single verification condition from

  ```
  {X=x ∧ Y=y} R:=X; X:=Y; X:=R
  {X=y ∧ R=x}
  ```

  is

  ```
  X=x ∧ Y=y ⇒ Y=y ∧ X=x
  ```

  which is logically equivalent to true.

Justification of VCs for Sequences (1)

- Case 1: If the verification conditions for
  $$\{P\}\ C_1\ ;\ \{R\}\ C_2\ \{Q\}$$
  are provable
- Then the verification conditions for
  $$\{P\}\ C_1\ \{R\} \textit{ and } \{R\}\ C_2\ \{Q\}$$
  must both be provable
- Hence by induction
  $$\vdash\ \{P\}\ C_1\ \{R\} \textit{ and } \vdash\ \{R\}\ C_2\ \{Q\}$$
- Hence by the sequencing rule
  $$\vdash\ \{P\}\ C_1 ; C_2\ \{Q\}$$

Justification of VCs for Sequences (2)

- Case 2: If the verification conditions for
  $$\{P\}\ C;\ V := E\ \{Q\}$$
  are provable, then the verification conditions for
  $$\{P\}\ C\ \{Q[E/V]\}$$
  are also provable
- Hence by induction
  $$\vdash\ \{P\}\ C\ \{Q[E/V]\}$$
- Hence by the derived sequenced assignment rule
  $$\vdash\ \{P\}\ C;\ V := E\ \{Q\}$$

VCs for WHILE-Commands

- A correctly annotated specification of a WHILE-command has the form
  $$\{P\}\ \mathtt{WHILE}\ S\ \mathtt{DO}\ \{R\}\ C\ \{Q\}$$
- The annotation R is called an invariant

WHILE-commands

The verification conditions generated by

$$\{P\}\ \mathtt{WHILE}\ S\ \mathtt{DO}\ \{R\}\ C\ \{Q\}$$

are

i $P \Rightarrow R$

ii $R \wedge \neg S \Rightarrow Q$

iii the verification conditions generated by $\{R \wedge S\}\ C\{R\}$

Example

- The verification conditions for
  $\{R=X \wedge Q=0\}$
  WHILE $Y \leq R$ DO $\{X=R+Y\times Q\}$
  (R:=R−Y; Q:=Q+1)
  $\{X = R+(Y\times Q) \wedge R<Y\}$
  are:
  i $R=X \wedge Q=0 \Rightarrow (X = R+(Y\times Q))$
  ii $X = R+Y\times Q \wedge \neg(Y\leq R) \Rightarrow (X = R+(Y\times Q) \wedge R<Y)$
  together with the verification condition for
  $\{X = R+(Y\times Q) \wedge (Y\leq R)\}$
  (R:=R−Y; Q:=Q+1)
  $\{X=R+(Y\times Q)\}$
  which consists of the single condition
  iii $X = R+(Y\times Q) \wedge (Y\leq R) \Rightarrow X = (R-Y)+(Y\times(Q+1))$

Example Summarised

- By previous slide

  $\vdash$ {R=X $\wedge$ Q=0}
  WHILE Y$\leq$R DO
  (R:=R−Y; Q:=Q+1)
  {X = R+(Y$\times$Q) $\wedge$ R<Y}

  if

  $\vdash$ R=X $\wedge$ Q=0 $\Rightarrow$
  (X = R+(Y$\times$Q))

  and

  $\vdash$ X = R+(Y$\times$Q) $\wedge$ $\neg$(Y$\leq$R)$\Rightarrow$
  (X = R+(Y$\times$Q) $\wedge$ R<Y)

  and

  $\vdash$ X = R+(Y$\times$Q) $\wedge$ (Y$\leq$R) $\Rightarrow$
  X = (R−Y)+(Y$\times$(Q+1))

Justification of WHILE VCs

- If the verification conditions for
  $\{P\}$ *WHILE* S *DO* $\{R\}$ C $\{Q\}$
  are provable, then
  $\vdash P \Rightarrow R$
  $\vdash (R \wedge \neg S) \Rightarrow Q$
  and the verification conditions for
  $\{R \wedge S\}$ C $\{R\}$
  are provable
- By induction
  $\vdash \{R \wedge S\}$ C $\{R\}$
- Hence by the derived WHILE-rule
  $\vdash \{P\}$ *WHILE* S *DO* C $\{Q\}$

Summary

- Have outlined the design of an automated program verifier
- Annotated specifications compiled to mathematical statements
  - if the statements (VCs) can be proved, the program is verified
- Human help is required to give the annotations and prove the VCs
- The algorithm was justified by an inductive proof
  - it appeals to the derived rules
- All the techniques introduced earlier are used
  - backwards proof
  - derived rules
  - annotation

Other uses of $\{P\}$ C $\{Q\}$

- So far we've assumed P, C, Q are given, and tried to automate finding a proof derivation ending in $\vdash \{P\}$ C $\{Q\}$.
- But we've all used Prolog.
- What if we're given P and C, can we infer a Q?
  Is there a best? ('Strongest Postcondition')
- What if we're given C and Q, can we infer a P?
  Is there a best? ('Weakest Precondition')
- What if we're given P and Q, can we infer a C?
  ('Program Refinement' or 'Program synthesis').

(Aside: you can even see the derived rules above as a Prolog-like re-writing of a rule to "accept any argument by making its formal parameters into variables").

Weakest Preconditions, Strongest Postconditions, Refinement

Dijkstra's weakest preconditions

- Weakest preconditions is a *theory of refinement*
  - idea is to calculate a program to achieve a postcondition
  - not a theory of post-hoc verification
- Non-determinism a key idea in Dijkstra's presentation
  - start with a non-deterministic high level pseudo-code
  - refine to deterministic and efficient code
- Weakest preconditions (wp) are for total correctness
- Weakest *liberal* preconditions (wlp) for partial correctness
- If C is a command and Q a predicate, then informally:
  $\mathrm{wlp}(C, Q)$ ='The weakest predicate P such that $\{P\}\ C\ \{Q\}$'
  $\mathrm{wp}(C, Q)$ ='The weakest predicate P such that $[P]\ C\ [Q]$'

(If P and Q are formulae then $Q \Rightarrow P$ means P is 'weaker' than Q)

Rules for weakest preconditions

- Relation with Hoare specifications:

$$\begin{array}{lcl} \{P\}\ C\ \{Q\} & \Leftrightarrow & P \Rightarrow \mathrm{wlp}(C, Q) \\ [P]\ C\ [Q] & \Leftrightarrow & P \Rightarrow \mathrm{wp}(C, Q) \end{array}$$

- Dijkstra gives rules for computing weakest preconditions:

$$\begin{array}{l} \mathrm{wp}(V := E, Q) = Q[E/V] \\ \mathrm{wp}(C_1 ; C_2,\ Q) = \mathrm{wp}(C_1, \mathrm{wp}(C_2,\ Q)) \\ \mathrm{wp}(\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2,\ Q) \\ \qquad = (S \Rightarrow \mathrm{wp}(C_1, Q)) \wedge (\neg S \Rightarrow \mathrm{wp}(C_2, Q)) \end{array}$$

- for deterministic loop-free code the same equations hold for wlp
- Rule for `WHILE`-commands doesn't give a first-order result
- Weakest preconditions closely related to VCs
- VCs for $\{P\}\ C\ \{Q\}$ are related to $P \Rightarrow \mathrm{wlp}(C, Q)$
  - VCs use annotations to ensure first-order formulae can be generated

Using wlp to improve verification condition method

- If C is loop-free then VC for $\{P\}\ C\ \{Q\}$ is $P \Rightarrow \mathrm{wlp}(C, Q)$
  - no annotations needed in sequences!
- Cannot in general compute a finite formula for $\mathrm{wlp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ Q)$
- The following holds
  $\mathrm{wlp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ Q) \Leftrightarrow$
  $\quad$ *if* S *then* $\mathrm{wlp}(C,\ \mathrm{wlp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ Q))$ *else* Q
  but this doesn't in general define $\mathrm{wlp}(C, Q)$ as a finite formula
- Could use a hybrid VC and wlp method

Strongest postconditions

- Define $\mathrm{sp}(C, P)$ to be 'strongest' Q such that $\{P\}\ C\ \{Q\}$
  - partial correctness: $\{P\}\ C\ \{\mathrm{sp}(C, P)\}$
  - strongest means if $\{P\}\ C\ \{Q\}$ then $\mathrm{sp}(C, P) \Rightarrow Q$
- Note that wlp goes 'backwards', but sp goes 'forwards'
  - verification condition for $\{P\}\ C\ \{Q\}$ is: $\mathrm{sp}(C, P) \Rightarrow Q$
- By 'strongest' and Hoare logic postcondition weakening
  - $\{P\}\ C\ \{Q\}$ if and only if $\mathrm{sp}(C, P) \Rightarrow Q$

Strongest postconditions for loop-free code

- Only considering loop-free code

$$\begin{array}{l}\mathrm{sp}(V := E,\ P) = \exists v.\ V = E[v/V] \wedge P[v/V] \\ \mathrm{sp}(C_1\,;\,C_2,\ P) = \mathrm{sp}(C_2,\ \mathrm{sp}(C_1,\ P)) \\ \mathrm{sp}(\texttt{IF}\ S\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2,\ P) \\ \qquad\qquad = \mathrm{sp}(C_1,\ P \wedge S) \vee \mathrm{sp}(C_2,\ P \wedge \neg S)\end{array}$$

- $\mathrm{sp}(V := E,\ P)$ corresponds to Floyd assignment axiom
- Can *dynamically prune* conditionals because $\mathrm{sp}(C, \mathsf{F}) = \mathsf{F}$
- Computing strongest postconditions is *symbolic execution*

Computing sp versus wlp

- Computing sp is like execution
  - can simplify as one goes along with the 'current state'
  - may be able to resolve branches, so can avoid executing them
  - Floyd assignment rule complicated in general
  - sp used for symbolically exploring 'reachable states' (related to *model checking*)
- Computing wlp is like backwards proof
  - don't have 'current state', so can't simplify using it
  - can't determine conditional tests, so get big `if-then-else` trees
  - Hoare assignment rule simpler for arbitrary formulae
  - wlp used for improved verification conditions

Exercises not lectured

- Compute

$$\begin{array}{l}\mathrm{sp}(\mathtt{R} := 0; \\ \quad \mathtt{K} := 0; \\ \quad \texttt{IF}\ \mathtt{I} < \mathtt{J}\ \texttt{THEN}\ \mathtt{K} := \mathtt{K} + 1\ \texttt{ELSE}\ \mathtt{K} := \mathtt{K}; \\ \quad \texttt{IF}\ \mathtt{K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \texttt{THEN}\ \mathtt{R} := \mathtt{J} - \mathtt{I}\ \texttt{ELSE}\ \mathtt{R} := \mathtt{I} - \mathtt{J}, \\ \quad (\mathtt{I} = i \wedge \mathtt{J} = j \wedge j \leq i))\end{array}$$

- Hence show

$$\begin{array}{l}\{(\mathtt{I} = i \wedge \mathtt{J} = j \wedge j \leq i\} \\ \quad \mathtt{R} := 0; \\ \quad \mathtt{K} := 0; \\ \quad \texttt{IF}\ \mathtt{I} < \mathtt{J}\ \texttt{THEN}\ \mathtt{K} := \mathtt{K} + 1\ \texttt{ELSE}\ \mathtt{K} := \mathtt{K}; \\ \quad \texttt{IF}\ \mathtt{K} = 1 \wedge \neg(\mathtt{I} = \mathtt{J})\ \texttt{THEN}\ \mathtt{R} := \mathtt{J} - \mathtt{I}\ \texttt{ELSE}\ \mathtt{R} := \mathtt{I} - \mathtt{J}) \\ \{R = i - j\}\end{array}$$

- Do same example use wlp

Using sp to generate verification conditions

- If C is loop-free then VC for $\{P\}\ C\ \{Q\}$ is $\mathrm{sp}(C,\ P) \Rightarrow Q$
- Cannot in general compute a finite formula for $\mathrm{sp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ P)$
- The following holds
  $$\mathit{consp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ P) \Leftrightarrow \mathrm{sp}(\texttt{WHILE}\ S\ \texttt{DO}\ C,\ \mathrm{sp}(C,\ (P \wedge S))) \vee (P \wedge \neg S)$$
  but this doesn't in general define $\mathrm{wlp}(C, Q)$ as a finite formula
- As with wlp, can use a hybrid VC and sp method

Summary

- Annotate then generate VCs is the classical method
  - practical tools: Gypsy (1970s), SPARK (current)
  - weakest preconditions are alternative explanation of VCs
  - wlp needs fewer annotations than VC method described earlier
  - wlp also used for refinement
- VCs and wlp go backwards, sp goes forward
  - sp provides verification method based on symbolic simulation
  - widely used for loop-free code
  - current research potential for forwards full proof of correctness
  - probably need mixture of forwards and backwards methods (Hoare's view)

Range of methods for proving $\{P\}C\{Q\}$

- Bounded model checking (BMC)
  - unwind loops a finite number of times
  - then symbolically execute
  - check states *reached* satisfy decidable properties
  - therefore not fully sound
- Full proof of correctness
  - add invariants to loops
  - generate verification conditions
  - prove verification conditions with a theorem prover
- Research goal: unifying framework for a spectrum of methods

decidable checking *proof of correctness*

New Topic: Refinement

- So far we have focused on proving programs meet specifications
- An alternative is to ensure a program is correct by construction
- The proof is performed in conjunction with the development
  - errors are spotted earlier in the design process
  - the reasons for design decisions are available
- Programming becomes less of a black art and more like an engineering discipline
- Rigorous development methods such as the B-Method, SPARK and the Vienna Development Method (VDM) are based on this idea
- The approach here is based on "Programming From Specifications"
  - a book by Carroll Morgan
  - simplified and with a more concrete semantics

Refinement Laws

- Laws of Programming refine a specification to a program
- As each law is applied, proof obligations are generated
- The laws are derived from the Hoare logic rules
- Several laws will be applicable at a given time
  - corresponding to different design decisions
  - and thus different implementations
- The "Art" of Refinement is in choosing appropriate laws to give an efficient implementation
- For example, given a specification that an array should be sorted:
  - one sequence of laws will lead to Bubble Sort
  - a different sequence will lead to Insertion Sort
  - see Morgan's book for an example of this

Refinement Specifications

- A *refinement specification* has the form $[P,\ Q]$
  - P is the precondition
  - Q is the postcondition
- Unlike a partial or total correctness specification, a refinement specification does not include a command
- Goal: derive a command that satisfies the specification
- P and Q correspond to the pre and post condition of a total correctness specification
- A command is required which, if started in a state satisfying P, will terminate in a state satisfying Q

Example

- [T, X=1]
  - this specifies that the code provided should terminate in a state where X has value 1 whatever state it is started in
- [X>0, Y=X^2]
  - from a state where X is greater than zero, the program should terminate with Y the square of X

A Little Wide-Spectrum Programming Language

- Let P, Q range over (predicate calculus) formulae
- Add specifications to commands

$$E ::= N \mid V \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid \ldots$$

$$B ::= \mathtt{T} \mid \mathtt{F} \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \ldots$$

$C ::=$ SKIP (does nothing, SKIP-Axiom is $\vdash [P]\ \mathtt{SKIP}\ [P]$)
$\mid\ V := E$
$\mid\ C_1 ; C_2$
$\mid\ \mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2$
$\mid\ \mathtt{BEGIN}\ \mathtt{VAR}\ V_1 ; \ldots\ \mathtt{VAR}\ V_1 ; C\ \mathtt{END}$
$\mid\ \mathtt{WHILE}\ B\ \mathtt{DO}\ C$
$\mid\ [P,\ Q]$

Specifications as Sets of Commands

- Refinement specifications can be mixed with other commands but are not in general executable
- Example

```
R:=X;
Q:=0;
```

$[R{=}X \wedge Y{>}0 \wedge Q{=}0,\quad X{=}R{+}Y{\times}Q]$

- Think of a specification as defining the set of implementations

$$[P,\ Q] = \{\ C \mid \vdash [P]\ C\ [Q]\ \}$$

  - Don't confuse use of $\{\cdots\}$ as set brackets and in Hoare triples
- For example

$$[\mathtt{T},\ \mathtt{X{=}1}] = \{\mathtt{"X:=1"},\ \mathtt{"IF}\ \neg\mathtt{(X=1)}\ \mathtt{THEN\ X:=1"},\ \mathtt{"X:=2;X:=X-1"},\ \cdots\}$$

Refinement-based program development

- The client provides a non-executable program (the specification)
- The programmer's job is to transform it into an executable program
- It will pass through a series of stages in which some parts are executable, but others are not
- Specifications give lots of freedom about how a result is obtained
  - executable code has no freedom
  - mixed programs have some freedom
- We use the notation $p_1 \supseteq p_2$ to mean program p_2 is more refined (i.e. has less freedom) than program p_1
  - **N.B.** The standard notation is $p_1 \sqsubseteq p_2$
- A program development takes us from the specification, through a series of mixed programs to (we hope) executable code $spec \supseteq mixed_1 \supseteq \ldots \supseteq mixed_n \supseteq code$

Various Laws (based on Hoare Logic) are used

The Skip Law

$[P,\ P] \supseteq \{\mathtt{SKIP}\}$

The Assignment Law

$[P[E/V],\ P] \supseteq \{V := E\}$

Precondition Weakening (beware name)

$[P,\ Q] \supseteq [R,\ Q]$ provided $\vdash P \Rightarrow R$

Postcondition Strengthening (beware name)

$[P,\ Q] \supseteq [P,\ R]$ provided $\vdash R \Rightarrow Q$

Derived Assignment Law

$[P,\ Q] \supseteq \{V{:=}E\}$ provided $\vdash P \Rightarrow Q[E/V]$

The Sequencing Law

$[P,\ Q] \supseteq [P,\ R];\ [R,\ Q]$

The Conditional Law

$[P,\ Q] \supseteq \mathtt{IF}\ S\ \mathtt{THEN}\ [P \wedge S,\ Q]\ \mathtt{ELSE}\ [P \wedge \neg S,\ Q]$

(Details beyond this course, see Mike Gordon's notes)

Summary

- Refinement 'laws' based on the Hoare logic can be used to develop programs formally
  - See Mike Gordon's notes for details of the laws
- A program is gradually converted from an unexecutable specification to executable code
- By applying different laws, different programs are obtained
  - may reach unrefinable specifications (blind alleys)
  - but will never get incorrect code
- A program developed in this way will meet its formal specification
  - one approach to 'Correct by Construction' (CbC) software engineering

(There is also a notion of 'Data Refinement Laws' which enable programs using abstract data types to be refined to use concrete data types. Our laws are really just 'Operation Refinement Laws'.)

Total Correctness

Total Correctness Specification

- So far our discussion has been concerned with partial correctness
  - what about termination
- A total correctness specification $[P]\ C\ [Q]$ is true if and only if
  - whenever C is executed in a state satisfying P, then the execution of C terminates
  - after C terminates Q holds
- Except for the `WHILE`-rule, all the axioms and rules described so far are sound for total correctness as well as partial correctness

Termination of `WHILE`-Commands

- `WHILE`-commands are the only commands that might not terminate
- Consider now the following proof

1. $\vdash \{\mathtt{T}\}\ \mathtt{X := X}\ \{\mathtt{T}\}$ (assignment axiom)
2. $\vdash \{\mathtt{T} \wedge \mathtt{T}\}\ \mathtt{X := X}\ \{\mathtt{T}\}$ (precondition strengthening)
3. $\vdash \{\mathtt{T}\}\ \mathtt{WHILE\ T\ DO\ X := X}\ \{\mathtt{T} \wedge \neg\mathtt{T}\}$ (2 and the `WHILE`-rule)

- If the `WHILE`-rule worked for total correctness, then this would show:

$$\vdash [\mathtt{T}]\ \mathtt{WHILE\ T\ DO\ X := X}\ [\mathtt{T} \wedge \neg\mathtt{T}]$$

- Thus the `WHILE`-rule is unsound for total correctness

Rules for Non-Looping Commands

- Replace $\{$ and $\}$ by $[$ and $]$, respectively, in:
  - Assignment axiom (see next slide for discussion)
  - Consequence rules
  - Conditional rule
  - Sequencing rule
- The following is a valid derived rule

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash [P]\ C\ [Q]}$$

if C contains no `WHILE`-commands

Total Correctness Assignment Axiom

- Assignment axiom for total correctness

$$\vdash\ [P[E/V]]\ V := E\ [P]$$

- Note that the assignment axiom for total correctness states that assignment commands *always* terminate
- So all function applications in expressions must terminate
- This might not be the case if functions could be defined recursively
- Consider $X := fact(-1)$, where $fact(n)$ is defined recursively:

$$fact(n)\ =\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times fact(n-1)$$

- (See the restrictions in Agda or Coq about all functions being total.)

Error Termination

- We assume erroneous expressions like $1/0$ don't cause problems
- Most programming languages will raise an error on division by zero
- In our logic it follows that

$$\vdash\ [\mathtt{T}]\ \mathtt{X} := 1/0\ [\mathtt{X} = 1/0]$$

- The assignment $\mathtt{X} := 1/0$ halts in a state in which $\mathtt{X} = 1/0$ holds
- This assumes that $1/0$ denotes some value that $\mathtt{X}$ can have

Two Possibilities

- There are two possibilities
  - i $1/0$ *denotes some number;*
  - ii $1/0$ *denotes some kind of 'error value'.*
- It seems at first sight that adopting (ii) is the most natural choice
  - this makes it tricky to see what arithmetical laws should hold
  - is $(1/0) \times 0$ equal to 0 or to some 'error value'?
  - if the latter, then it is no longer the case that $\forall n.\ n \times 0 = 0$ is valid
- It is possible to make everything work with undefined and/or error values, but the resultant theory is a bit messy

Example

- We assume that arithmetic expressions *always* denote numbers
- In some cases exactly what the number is will be not fully specified
  - for example, we will assume that m/n denotes a number for any m and n
  - only assume: $\neg(n = 0) \ \Rightarrow\ (m/n) \times n = m$
  - it is not possible to deduce anything about $m/0$ from this
  - in particular it is not possible to deduce that $(m/0) \times 0 = 0$
  - but $(m/0) \times 0 = 0$ does follow from $\forall n.\ n \times 0 = 0$
- People still argue about this – e.g. advocate "three-valued" logics

WHILE-rule for Total Correctness (i)

- WHILE-commands are the only commands in our little language that can cause non-termination
  - they are thus the only kind of command with a non-trivial termination rule
- The idea behind the WHILE-rule for total correctness is
  - to prove WHILE S DO C terminates
  - show that some non-negative quantity decreases on each iteration of C
  - this decreasing quantity is called a variant

WHILE-Rule for Total Correctness (ii)

- In the rule below, the variant is E, and the fact that it decreases is specified with an auxiliary variable n
- The hypothesis $\vdash P \wedge S \Rightarrow E \geq 0$ ensures the variant is non-negative

WHILE-rule for total correctness

$$\frac{\vdash [P \wedge S \wedge (E = n)]\ C\ [P \wedge (E < n)], \quad \vdash P \wedge S \Rightarrow E \geq 0}{\vdash [P]\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ [P \wedge \neg S]}$$

where E is an integer-valued expression
and n is an identifier not occurring in P, C, S or E.

Example

- We show

$$\vdash [\texttt{Y} > 0]\ \texttt{WHILE Y}\leq\texttt{R DO (R:=R-Y; Q:=Q+1)}\ [\texttt{T}]$$

- Take

$$\begin{array}{lcl} P & = & \texttt{Y} > 0 \\ S & = & \texttt{Y} \leq \texttt{R} \\ E & = & \texttt{R} \\ C & = & \texttt{(R:=R-Y; Q:=Q+1)} \end{array}$$

- We want to show $\vdash [P]\ \texttt{WHILE}\ S\ \texttt{DO}\ C\ [\texttt{T}]$
- By the WHILE-rule for total correctness it is sufficient to show

i $\vdash [P \wedge S \wedge (E = \texttt{n})]\ C\ [P \wedge (E < \texttt{n})]$
ii $\vdash P \wedge S \Rightarrow E \geq 0$

Example Continued (1)

- From previous slide:

$$\begin{array}{lcl} P & = & \texttt{Y} > 0 \\ S & = & \texttt{Y} \leq \texttt{R} \\ E & = & \texttt{R} \\ C & = & \texttt{(R:=R-Y; Q:=Q+1)} \end{array}$$

- We want to show

i $\vdash [P \wedge S \wedge (E = \texttt{n})]\ C\ [P \wedge (E < \texttt{n})]$
ii $\vdash P \wedge S \Rightarrow E \geq 0$

- The first of these, (i), can be proved by establishing

$$\vdash \{P \wedge S \wedge (E = \texttt{n})\}\ C\ \{P \wedge (E < \texttt{n})\}$$

- Then using the total correctness rule for non-looping commands

Example Continued (2)

- From previous slide:

$$\begin{array}{lcl} P & = & \mathtt{Y} > 0 \\ S & = & \mathtt{Y} \leq \mathtt{R} \\ E & = & \mathtt{R} \\ C & = & \mathtt{R:=R-Y;\ Q:=Q+1)} \end{array}$$

- The verification condition for $\{P \wedge S \wedge (E = \mathtt{n})\}\ C\ \{P \wedge (E < \mathtt{n})\}$ is:

$$\mathtt{Y} > 0 \wedge \mathtt{Y} \leq \mathtt{R} \wedge \mathtt{R} = \mathtt{n} \Rightarrow (\mathtt{Y} > 0 \wedge \mathtt{R} < \mathtt{n})[\mathtt{Q}+1/\mathtt{Q}][\mathtt{R}-\mathtt{Y}/\mathtt{R}]$$

  i.e. $\mathtt{Y} > 0 \wedge \mathtt{Y} \leq \mathtt{R} \wedge \mathtt{R} = \mathtt{n} \Rightarrow \mathtt{Y} > 0 \wedge \mathtt{R}-\mathtt{Y} < \mathtt{n}$ which follows from the laws of arithmetic
- The second subgoal, (ii), is just $\vdash \mathtt{Y} > 0 \wedge \mathtt{Y} \leq \mathtt{R} \Rightarrow \mathtt{R} \geq 0$

Termination Specifications

- The relation between partial and total correctness is informally given by the equation

  Total correctness = Termination + Partial correctness
- This informal equation can be represented by the following two rules of inferences

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad \vdash [P]\ C\ [\mathtt{I}]}{\vdash [P]\ C\ [Q]}$$

$$\frac{\vdash [P]\ C\ [Q]}{\vdash \{P\}\ C\ \{Q\} \qquad \vdash [P]\ C\ [\mathtt{I}]}$$

Derived Rules

- Multiple step rules for total correctness can be derived in the same way as for partial correctness
  - the rules are the same up to the brackets used
  - same derivations with total correctness rules replacing partial correctness ones

The Derived While Rule

- The derived WHILE-rule needs to handle the variant

Derived WHILE-rule for total correctness

$$\frac{\begin{array}{l} \vdash P \Rightarrow R \\ \vdash R \wedge S \Rightarrow E \geq 0 \\ \vdash R \wedge \neg S \Rightarrow Q \\ \vdash [R \wedge S \wedge (E = n)]\ C\ [R \wedge (E < n)] \end{array}}{\vdash [P]\ \mathtt{WHILE}\ S\ \mathtt{DO}\ C\ [Q]}$$

VCs for Termination

- Verification conditions are easily extended to total correctness
- To generate total correctness verification conditions for WHILE-commands, it is necessary to add a variant as an annotation in addition to an invariant
- Variant added directly after the invariant, in square brackets
- No other extra annotations are needed for total correctness
- VCs for WHILE-free code same as for partial correctness

WHILE Annotation

- A correctly annotated total correctness specification of a WHILE-command thus has the form

$$[P]\ \texttt{WHILE}\ S\ \texttt{DO}\ \{R\}[E]\ C\ [Q]$$

where R is the invariant and E the variant

- Note that the variant is intended to be a non-negative expression that decreases each time around the WHILE loop
- The other annotations, which are enclosed in curly brackets, are meant to be conditions that are true whenever control reaches them (as before)

WHILE VCs

- A correctly annotated specification of a WHILE-command has the form

$$[P]\ \texttt{WHILE}\ B\ \texttt{DO}\ \{R\}[E]\ C\ [Q]$$

WHILE-commands

The verification conditions generated from

$$[P]\ \texttt{WHILE}\ B\ \texttt{DO}\ \{R\}[E]\ C\ [Q]$$

are

i $P \Rightarrow R$

ii $R \wedge \neg B \Rightarrow Q$

iii $R \wedge B \Rightarrow E \geq 0$

iv the verification conditions generated by $[R \wedge B \wedge (E = n)]\ C[R \wedge (E < n)]$ where n is a variable not occurring in P, R, E, C, B, or Q.

Example

- The verification conditions for

$[\texttt{R=X} \wedge \texttt{Q=0}]$
$\texttt{WHILE Y}\leq\texttt{R DO \{X=R+Y}\times\texttt{Q\}[R]}$
$\texttt{(R:=R-Y; Q=Q+1)}$
$[\texttt{X = R+(Y}\times\texttt{Q)} \wedge \texttt{R<Y}]$

are:

i $\texttt{R=X} \wedge \texttt{Q=0} \Rightarrow \texttt{(X = R+(Y}\times\texttt{Q))}$

ii $\texttt{X = R+Y}\times\texttt{Q} \wedge \neg\texttt{(Y}\leq\texttt{R)} \Rightarrow \texttt{(X = R+(Y}\times\texttt{Q)} \wedge \texttt{R<Y)}$

iii $\texttt{X = R+Y}\times\texttt{Q} \wedge \texttt{Y}\leq\texttt{R} \Rightarrow \texttt{R}\geq\texttt{0}$

together with the verification condition for

$[\texttt{X = R+(Y}\times\texttt{Q)} \wedge \texttt{(Y}\leq\texttt{R)} \wedge \texttt{(R=n)}]$
$\texttt{(R:=R-Y; Q:=Q+1)}$
$[\texttt{X=R+(Y}\times\texttt{Q)} \wedge \texttt{(R<n)}]$

Example Continued

- The single verification condition for
  `[X = R+(Y×Q) ∧ (Y≤R) ∧ (R=n)]`
  `(R:=R−Y; Q:=Q+1)`
  `[X=R+(Y×Q) ∧ (R<n)]`
  is
  iv `X = R+(Y×Q) ∧ (Y≤R) ∧ (R=n) ⇒`
  `X = (R−Y)+(Y×(Q+1)) ∧ ((R−Y)<n)`
- But this isn't true
  - take `Y=0`
- To prove `R−Y<n` we need to know `Y>0`
- *Exercise:* Explain why one would not expect to be able to prove the verification conditions of this last example
- *Hint:* Consider the original specification

Summary

- We have given rules for total correctness
- They are similar to those for partial correctness
- The main difference is in the `WHILE`-rule
  - because `WHILE` commands are the only ones that can fail to terminate
- Must prove a non-negative expression is decreased by the loop body
- Derived rules and VC generation rules for partial correctness easily extended to total correctness
- Interesting stuff on the web
  - http://www.crunchgear.com/2008/12/31/zune-bug-explained-in-detail
  - http://research.microsoft.com/en-us/projects/t2/

Soundness, Completeness

Summary: soundness, decidability, completeness

- Hoare logic is sound
- Hoare logic is undecidable
  - deciding $\{\mathtt{T}\}C\{\mathtt{F}\}$ is halting problem
- Hoare logic for our simple language is *relatively complete*
  - All failures to prove $\vdash \{P\}\ \mathrm{C}\ \{Q\}$, for a valid statement $\models \{P\}\ \mathrm{C}\ \{Q\}$, can be traced back to a failure to prove $\vdash_{arith} \phi$ for some valid arithmetic statement $\models_{arith} \phi$.

> The incompleteness of the proof system for simple Hoare logic stems from the weakness of the proof system of the assertion language logic, not any weakness of the Hoare logic proof system.

- Clarke showed relative completeness fails for more-complex languages

Richer Languages, Separation Logic

Limits to Hoare Logic as presented

- All the axioms and rules given so far were quite straightforward
  - may have given a false sense of simplicity
- Hard to give rules for anything other than *very* simple constructs
  - an incentive for using simple languages
- We already saw with the assignment axiom that intuition over how to formulate a rule might be wrong

Some sources of additional difficulty:

- blocks and local variables need additional work
- pointers and aliasing can cause problems
- concurrency can cause problems

We'll look at some ways to address these issues.

Array assignments

- **Syntax:** $V(E_1) := E_2$
- **Semantics:** the state is changed by assigning the value of the term E_2 to the E_1-th component of the array variable V
- **Example:** `A(X+1) := A(X)+2`
  - if the the value of `X` is x
  - and the value of the x-th component of `A` is n
  - then the value stored in the $(x+1)$-th component of `A` becomes $n+2$

Naive Array Assignment Axiom Fails

- The axiom
  $\vdash \{P[E_2/A(E_1)]\}\ A(E_1) := E_2\ \{P\}$
  doesn't work
- Take $P \equiv$ '`X=Y` $\wedge$ `A(Y)=0`', $E_1 \equiv$ '`X`', $E_2 \equiv$ '`1`'
  - since `A(X)` does not occur in P
  - it follows that P`[1/A(X)]` $= P$
  - hence the axiom yields: $\vdash$ `{X=Y` $\wedge$ `A(Y)=0} A(X):=1 {X=Y` $\wedge$ `A(Y)=0}`
- Must take into account possibility that changes to `A(X)` may change `A(Y)`, `A(Z)` etc
  - since `X` might equal `Y`, `Z` etc (i.e. aliasing)
- Related to the *Frame Problem* in AI

Reasoning About Arrays

- The naive array assignment axiom

$$\vdash \{P[E_2/A(E_1)]\}\ A(E_1) := E_2\ \{P\}$$

  fails: changes to $A(X)$ may also change $A(Y)$, $A(Z)$, ...
- The solution, due to Hoare, is to treat an array assignment

$$A(E_1) := E_2$$

  as an ordinary assignment (albeit one which overwrites the whole array)

$$A := A\{E_1 \leftarrow E_2\}$$

  where the term $A\{E_1 \leftarrow E_2\}$ denotes an array identical to A, except that the E_1-th component is changed to have the value E_2
- Side-steps the general problem of how to treat *aliasing*

Array Assignment axiom

Array assignment is now a special case of ordinary assignment:

$$A := A\{E_1 \leftarrow E_2\}$$

- So the array assignment axiom is just ordinary assignment axiom

$$\vdash \{P[A\{E_1 \leftarrow E_2\}/A]\}\ A := A\{E_1 \leftarrow E_2\}\ \{P\}$$

- Thus:

The array assignment axiom

$$\vdash \{P[A\{E_1 \leftarrow E_2\}/A]\}\ A(E_1) := E_2\ \{P\}$$

Here A is an array variable, E_1 is an integer valued expression, P is any statement and the notation $A\{E_1 \leftarrow E_2\}$ denotes the array identical to A, except that $A(E_1) = E_2$.

Array Axioms

In order to reason about arrays, we need the following axioms in $\vdash_{arith}$ to define the meaning of the notation $A\{E_1 \leftarrow E_2\}$

The array axioms

$$\vdash A\{E_1 \leftarrow E_2\}(E_1) = E_2$$
$$\vdash E_1 \neq E_3 \Rightarrow A\{E_1 \leftarrow E_2\}(E_3) = A(E_3)$$

- Second of these is a *Frame Axiom*
  - it captures that E_1 and E_3 are equal as values, not just syntactically equal as when used in substitution
  - don't confuse with Frame Rule of Separation Logic (later)
  - "frame" is a rather overloaded word!

Concurrency

Hoare logic as we have seen it so far has a fundamental problem with shared-variable concurrency. Consider the two commands:

- $C_1 \equiv$ `(X := X+1; X := X+1)`
- $C_2 \equiv$ `(X := X+2)`
- In sequential code C_1 and C_2 have identical meanings and hence $\{P\}\ C_1\ \{Q\} \Leftrightarrow \{P\}\ C_2\ \{Q\}$
- But suppose the program maintains `X` as an even integer. An unfortunate read of `X` in C_2 might see an odd integer but this is not possible in C_1.
- Solutions:
  - Rely-guarantee reasoning – as well as precondition P and postcondition Q our logic has assumptions A it assumes and guarantees G it makes.
  - Concurrent *Separation Logic*. We'll now look at the sequential part of separation logic.

New Topic: Separation logic

- One of several competing methods for reasoning about pointers
- Details took 30 years to evolve
- Shape predicates due to Rod Burstall in the 1970s
- Separation logic: by O'Hearn, Reynolds and Yang around 2000
- Several partially successful attempts before separation logic
- Very active research area
- QMUL, UCL, Cambridge, Harvard, Princeton, Yale
- Microsoft

Pointers and the state

- So far the state just determined the values of variables
- values assumed to be numbers
- preconditions and postconditions are first-order logic statements
- state same as a valuation $s : Var \to Val$
- To model pointers – e.g. as in C – add *heap* to state
- heap maps *locations* (pointers) to their contents
- *store* maps variables to values (previously called state)
- contents of locations can be locations or values

  $\mathtt{x} \mapsto l_1 \mapsto l_2 \mapsto v$

  store *heap* *heap*
- or if you prefer, values are integers but any integer can be treated as a (pointer to a) location

Adding pointer operations to our language

Expressions:

$E ::= N \mid V \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid \ldots$

Boolean expressions:

$B ::= \mathtt{T} \mid \mathtt{F} \mid E_1 = E_2 \mid E_1 \leq E_2 \mid \ldots$

Commands:

$C ::= V := E$		*value assignments*
$\mid V := [E]$		*fetch assignments*
$\mid [E_1] := E_2$		*heap assignments (heap mutation)*
$\mid V := \mathtt{cons}(E_1, \ldots, E_n)$		*allocation assignments*
$\mid \mathtt{dispose}(E)$		*pointer disposal*
$\mid C_1 \ ; \ C_2$		*sequences*
$\mid \mathtt{IF}\ B\ \mathtt{THEN}\ C_1\ \mathtt{ELSE}\ C_2$		*conditionals*
$\mid \mathtt{WHILE}\ B\ \mathtt{DO}\ C$		*while commands*

Pointer manipulation constructs and faulting

- Commands executed in a state (s, h)
- Reading, writing or disposing pointers might *fault*
- **Fetch assignments:** $V := [E]$
- evaluate E to get a location l
- fault if l is not in the heap
- otherwise assign contents of l in heap to the variable V
- **Heap assignments:** $[E_1] := E_2$
- evaluate E_1 to get a location l
- fault if the l is not in the heap
- otherwise store the value of E_2 as the new contents of l in the heap
- **Pointer disposal:** $\mathtt{dispose}(E)$
- evaluate E to get a pointer l (a number)
- fault if l is not in the heap
- otherwise remove l from the heap

Allocation assignments

- **Allocation assignments:** V`:=cons(`$E_1, \ldots, E_n$`)`
- choose n consecutive locations that are not in the heap, say $l, l{+}1, \ldots$
- extend the heap by adding $l, l{+}1, \ldots$ to it
- assign l to the variable V in the store
- make the values of $E_1, E_2, \ldots$ be the new contents of $l, l{+}1, \ldots$ in the heap
- Allocation assignments never fault
- Allocation assignments are *non-deterministic*
- any suitable $l, l{+}1, \ldots$ not in the heap can be chosen
- always exists because the heap is finite during execution

Example (different from the background reading)

`X:=cons(0,1,2); [X]:=Y+1; [X+1]:=Z; Y:=[Y+Z]`

- `X:=cons(0,1,2)` allocates three new pointers, say l, $l{+}1$, $l{+}2$
- l initialised with contents `0`, $l{+}1$ with `1` and $l{+}2$ with `2`
- variable `X` is assigned l as its value in store
- `[X]:=Y+1` changes the contents of l
- l gets value of `Y+1` as new contents in heap
- `[X+1]:=Z` changes the contents of $l{+}1$
- $l{+}1$ gets the value of `Z` as new contents in heap
- `Y:=[Y+Z]` changes the value of `Y` in the store
- `Y` assigned in the store the contents of `Y+Z` in the heap
- faults if the value of `Y+Z` is not in the heap

Separating Conjunction

- Separating conjunction $P \star Q$
  - heap can be split into two disjoint components
  - P is true of one component and Q of the other
  - allows local reasoning – aliases are temporarily banned
  - $\star$ is commutative and associative

We've already said that Hoare Logic cannot deal with shared-variable concurrency, so the rule

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad \vdash \{P'\}\ C'\ \{Q'\}}{\vdash \{P \wedge P'\}\ C\ \texttt{PAR}\ C'\ \{Q \wedge Q'\}}$$

is unsound. But the rule

$$\frac{\vdash \{P\}\ C\ \{Q\} \qquad \vdash \{P'\}\ C'\ \{Q'\}}{\vdash \{P \star P'\}\ C\ \texttt{PAR}\ C'\ \{Q \star Q'\}}$$

is sound as the heaps used by C and C' must be disjoint (we're also assuming that C and C' use disjoint variables).

Separation logic formulae

There are more formulae for the heap component of states:

- emp is true only of an empty heap
- $l \mapsto v$ is true only for a heap with one heap location l which stores the value v

Along with the frame rule

The frame rule

$$\frac{\vdash \{P\}\ C\ \{Q\}}{\vdash \{P \star R\}\ C\ \{Q \star R\}}$$

where no variable modified by C occurs free in R.

These enable us to reason about the heap effectively (details beyond these notes), when used along with the traditional Hoare-logic axioms (but using a Floyd-like Assignment rule).

Part 2: Temporal Logic and Model Checking

Temporal Logic and Model Checking

- Model
  - mathematical structure extracted from hardware or software
- Temporal logic
  - provides a language for specifying functional properties
- Model checking
  - checks whether a given property holds of a model

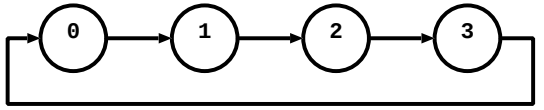
- Model checking is a kind of static verification
  - dynamic verification is simulation (HW) or testing (SW)

Models

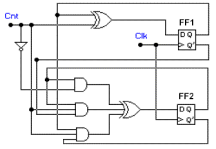
- A model is (for now) specified by a pair (S, R)
  - S is a set of *states*
  - R is a *transition relation*
- Models will get more components later
  - (S, R) also called a transition system
- $R\ s\ s'$ means s' can be reached from s in one step
  - here $R : S \to (S \to \mathbb{B})$ (where $\mathbb{B} = \{true, false\}$)
  - more conventional to have $R \subseteq S \times S$, which is equivalent
  - i.e. $R_{\text{(this course)}}\ s\ s' \Leftrightarrow (s, s') \in R_{\text{(some textbooks)}}$

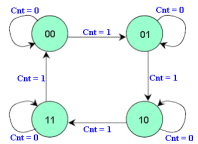
A simple example model

- A simple model: $(\underbrace{\{0,1,2,3\}}_{S}, \underbrace{\lambda n\ n'.\ n' = n{+}1 (mod\ 4)}_{R})$
  - where "$\lambda x.\ \cdots x \cdots$" is the function mapping x to $\cdots x \cdots$
  - so $R\ n\ n' = (n' = n{+}1 (mod\ 4))$
  - e.g. $R\ 0\ 1 \wedge R\ 1\ 2 \wedge R\ 2\ 3 \wedge R\ 3\ 0$



- Might be extracted from:





[Acknowledgement: http://eelab.usyd.edu.au/digital_tutorial/part3/t-diag.htm]

DIV: a software example

- Perhaps a familiar program:

```
0:  R:=X;
1:  Q:=0;
2:  WHILE Y≤R DO
3:   (R:=R-Y;
4:    Q:=Q+1)
5:
```

- State (pc, x, y, r, q)
  - $pc \in \{0,1,2,3,4,5\}$ program counter
  - x, y, r, $q \in \mathbb{Z}$ are the values of X, Y, R, Q
- Model $(S_{\text{DIV}}, R_{\text{DIV}})$ where:

$$S_{\text{DIV}} = [0..5] \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \quad \text{(where } [m..n] = \{m, m{+}1, \ldots, n\}\text{)}$$

$$\begin{array}{ll} \forall x\ y\ r\ q.\ R_{\text{DIV}}\ (0,x,y,r,q)\ (1,x,y,x,q) & \wedge \\ \qquad R_{\text{DIV}}\ (1,x,y,r,q)\ (2,x,y,r,0) & \wedge \\ \qquad R_{\text{DIV}}\ (2,x,y,r,q)\ ((\mathit{if}\ y{\leq}r\ \mathit{then}\ 3\ \mathit{else}\ 5),x,y,r,q) & \wedge \\ \qquad R_{\text{DIV}}\ (3,x,y,r,q)\ (4,x,y,(r{-}y),q) & \wedge \\ \qquad R_{\text{DIV}}\ (4,x,y,r,q)\ (2,x,y,r,(q{+}1)) & \end{array}$$

- [Above changed from lecture to make R_{DIV} partial!]

Deriving a transition relation from a state machine

- State machine transition function : $\delta : \mathit{Inp} \times \mathit{Mem} \to \mathit{Mem}$
  - Inp is a set of inputs
  - Mem is a memory (set of storable values)
- Model: (S_δ, R_δ) where:

$$S_\delta = \mathit{Inp} \times \mathit{Mem}$$

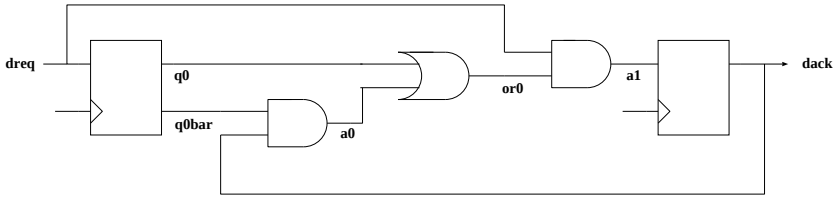
$$R_\delta\ (i,m)\ (i',m') \ = \ (m' \ = \ \delta(i,m))$$

and

  - i' arbitrary: determined by environment not by machine
  - m' determined by input and current state of machine
- Deterministic machine, non-deterministic transition relation
  - inputs unspecified (determined by environment)
  - so called “input non-determinism”

RCV: a state machine specification of a circuit

- Part of a handshake circuit:



- Input: dreq, Memory: $(q0, \mathit{dack})$
- Relationships between Boolean values on wires:

$$\begin{array}{lcl} q0\mathit{bar} & = & \neg q0 \\ a0 & = & q0\mathit{bar} \wedge \mathit{dack} \\ or0 & = & q0 \vee a0 \\ a1 & = & \mathit{dreq} \wedge or0 \end{array}$$

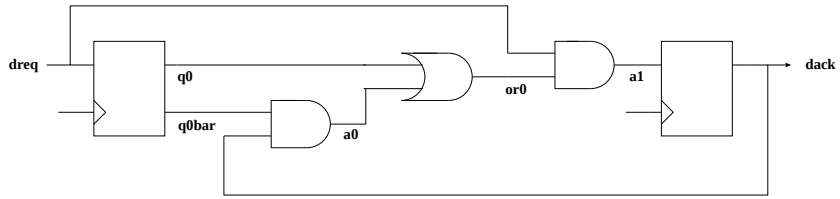
- State machine: $\delta_{\text{RCV}} : \mathbb{B} \times (\mathbb{B}{\times}\mathbb{B}) \to (\mathbb{B}{\times}\mathbb{B})$

$$\delta_{\text{RCV}}\ (\underbrace{\mathit{dreq}}_{\mathit{Inp}},\ \underbrace{(q0, \mathit{dack})}_{\mathit{Mem}}) = (\mathit{dreq},\ \ \mathit{dreq} \wedge (q0 \vee (\neg q0 \wedge \mathit{dack})))$$

- RTL model – could have lower level model with clock edges

RCV: a model of the circuit

- Circuit from previous slide:



- State represented by a triple of Booleans $(\mathit{dreq}, q0, \mathit{dack})$
- By De Morgan Law: $q0 \vee (\neg q0 \wedge \mathit{dack}) = q0 \vee \mathit{dack}$
- Hence δ_{RCV} corresponds to model $(S_{\text{RCV}}, R_{\text{RCV}})$ where:

$$S_{\text{RCV}} = \mathbb{B} \times \mathbb{B} \times \mathbb{B}$$

$$R_{\text{RCV}}\ (\mathit{dreq}, q0, \mathit{dack})\ (\mathit{dreq}', q0', \mathit{dack}') = (q0' = \mathit{dreq}) \wedge (\mathit{dack}' = (\mathit{dreq} \wedge (q0 \vee \mathit{dack})))$$

[Note: we are identifying $\mathbb{B} \times \mathbb{B} \times \mathbb{B}$ with $\mathbb{B} \times (\mathbb{B} \times \mathbb{B})$]

Some comments

- R_{RCV} is non-deterministic and total
  - $R_{\text{RCV}}\ (1,1,1)\ (0,1,1)$ and $R_{\text{RCV}}\ (1,1,1)\ (1,1,1)$ (where $1 = \mathit{true}$ and $0 = \mathit{false}$)
  - $R_{\text{RCV}}\ (\mathit{dreq}, q0, \mathit{dack})\ (\mathit{dreq}', \mathit{dreq}, (\mathit{dreq} \wedge (q0 \vee \mathit{dack})))$
- R_{DIV} is deterministic and partial
  - at most one successor state
  - no successor when $\mathit{pc} = 5$
- Non-deterministic models are very common, e.g. from:
  - asynchronous hardware
  - parallel software (more than one thread)
- Can extend any transition relation R to be total:

$$\begin{aligned} R_{\mathit{total}}\ s\ s' &= \mathit{if}\ (\exists s''.\ R\ s\ s'')\ \mathit{then}\ R\ s\ s'\ \mathit{else}\ (s' = s) \\ &= R\ s\ s' \vee (\neg(\exists s''.\ R\ s\ s'') \wedge (s' = s)) \end{aligned}$$

  - sometimes totality required (e.g. in the book *Model Checking* by Clarke et. al)

JM1: a non-deterministic software example

- From Jhala and Majumdar's tutorial:

Thread 1	Thread 2
0: IF LOCK=0 THEN LOCK:=1;	0: IF LOCK=0 THEN LOCK:=1;
1: X:=1;	1: X:=2;
2: IF LOCK=1 THEN LOCK:=0;	2: IF LOCK=1 THEN LOCK:=0;
3:	3:

- Two program counters, state: $(\mathit{pc}_1, \mathit{pc}_2, \mathit{lock}, x)$

$$S_{\text{JM1}} = [0..3] \times [0..3] \times \mathbb{Z} \times \mathbb{Z}$$

$$\begin{aligned} \forall \mathit{pc}_1\ \mathit{pc}_2\ \mathit{lock}\ x.\ & R_{\text{JM1}}\ (0, \mathit{pc}_2, 0, x)\ (1, \mathit{pc}_2, 1, x) \wedge \\ & R_{\text{JM1}}\ (1, \mathit{pc}_2, \mathit{lock}, x)\ (2, \mathit{pc}_2, \mathit{lock}, 1) \wedge \\ & R_{\text{JM1}}\ (2, \mathit{pc}_2, 1, x)\ (3, \mathit{pc}_2, 0, x) \wedge \\ & R_{\text{JM1}}\ (\mathit{pc}_1, 0, 0, x)\ (\mathit{pc}_1, 1, 1, x) \wedge \\ & R_{\text{JM1}}\ (\mathit{pc}_1, 1, \mathit{lock}, x)\ (\mathit{pc}_1, 2, \mathit{lock}, 2) \wedge \\ & R_{\text{JM1}}\ (\mathit{pc}_1, 2, 1, x)\ (\mathit{pc}_1, 3, 0, x) \end{aligned}$$

- Not-deterministic:
  $R_{\text{JM1}}\ (0,0,0,x)\ (1,0,1,x)$
  $R_{\text{JM1}}\ (0,0,0,x)\ (0,1,1,x)$
- Not so obvious that R_{JM1} is a correct model

Atomic properties (properties of states)

- Atomic properties are true or false of individual *states*
  - an atomic property p is a function $p : S \to \mathbb{B}$
  - can also be regarded as a subset of state: $p \subseteq S$
- Example atomic properties of RCV (where $1 = \mathit{true}$ and $0 = \mathit{false}$)

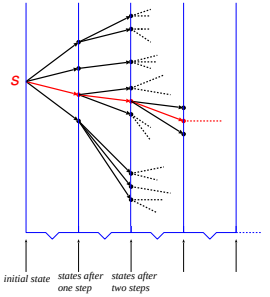
$$\begin{aligned} \texttt{Dreq}(\mathit{dreq}, q0, \mathit{dack}) &= (\mathit{dreq} = 1) \\ \texttt{NotQ0}(\mathit{dreq}, q0, \mathit{dack}) &= (q0 = 0) \\ \texttt{Dack}(\mathit{dreq}, q0, \mathit{dack}) &= (\mathit{dack} = 1) \\ \texttt{NotDreqAndQ0}(\mathit{dreq}, q0, \mathit{dack}) &= (\mathit{dreq}{=}0) \wedge (q0{=}1) \end{aligned}$$

- Example atomic properties of DIV

$$\begin{aligned} \texttt{AtStart}\ (\mathit{pc}, x, y, r, q) &= (\mathit{pc} = 0) \\ \texttt{AtEnd}\ (\mathit{pc}, x, y, r, q) &= (\mathit{pc} = 5) \\ \texttt{InLoop}\ (\mathit{pc}, x, y, r, q) &= (\mathit{pc} \in \{3, 4\}) \\ \texttt{YleqR}\ (\mathit{pc}, x, y, r, q) &= (y \leq r) \\ \texttt{Invariant}\ (\mathit{pc}, x, y, r, q) &= (x = r + (y \times q)) \end{aligned}$$

Model behaviour viewed as a computation tree

- Atomic properties are true or false of individual states
- General properties are true or false of whole behaviour
- Behaviour of (S, R) starting from $s \in S$ as a tree:



- A path is shown in red
- Properties may look at all paths, or just a single path
  - CTL: Computation Tree Logic (all paths from a state)
  - LTL: Linear Temporal Logic (a single path)

Paths

- A path of (S, R) is represented by a function $\pi : \mathbb{N} \to S$
  - $\pi(i)$ is the ith element of π (first element is $\pi(0)$)
  - might sometimes write $\pi\ i$ instead of $\pi(i)$
  - $\pi{\downarrow}i$ is the i-th tail of π so $\pi{\downarrow}i(n) = \pi(i+n)$
  - successive states in a path must be related by R

- Path $R\ s\ \pi$ is true if and only if π is a path starting at s:

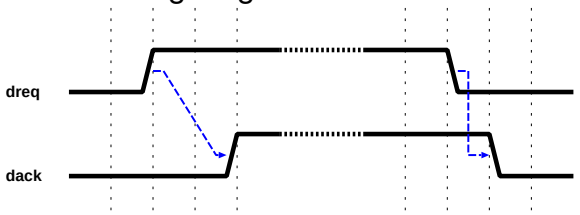
  $$\text{Path}\ R\ s\ \pi\ =\ (\pi(0) = s)\ \wedge\ \forall i.\ R\ (\pi(i))\ (\pi(i{+}1))$$

  where:

  $$\text{Path} : \underbrace{(S \to S \to \mathbb{B})}_{\text{transition relation}} \to \underbrace{S}_{\text{initial state}} \to \underbrace{(\mathbb{N} \to S)}_{\text{path}} \to \mathbb{B}$$

RCV: example hardware properties

- Consider this timing diagram:



- Two handshake properties representing the diagram:
  - following a rising edge on `dreq`, the value of `dreq` remains 1 (i.e. *true*) until it is acknowledged by a rising edge on `dack`
  - following a falling edge on `dreq`, the value on `dreq` remains 0 (i.e. *false*) until the value of `dack` is 0
- A property language is used to formalise such properties

DIV: example program properties

```
0:  R:=X;
1:  Q:=0;
2:  WHILE Y≤R DO
3:   (R:=R-Y;
4:    Q:=Q+1)
5:
```

$$\begin{aligned}
\texttt{AtStart}\ (pc, x, y, r, q) &= (pc = 0)\\
\texttt{AtEnd}\ (pc, x, y, r, q) &= (pc = 5)\\
\texttt{InLoop}\ (pc, x, y, r, q) &= (pc \in \{3, 4\})\\
\texttt{YleqR}\ (pc, x, y, r, q) &= (y \le r)\\
\texttt{Invariant}\ (pc, x, y, r, q) &= (x = r + (y \times q))
\end{aligned}$$

- Example properties of the program `DIV`.
  - on every execution if `AtEnd` is true then `Invariant` is true and `YleqR` is not true
  - on every execution there is a state where `AtEnd` is true
  - on any execution if there exists a state where `YleqR` is true then there is also a state where `InLoop` is true
- Compare these with what is expressible in Hoare logic
  - execution: a path starting from a state satisfying `AtStart`

Recall JM1: a non-deterministic program example

Thread 1	Thread 2
0: IF LOCK=0 THEN LOCK:=1;	0: IF LOCK=0 THEN LOCK:=1;
1: X:=1;	1: X:=2;
2: IF LOCK=1 THEN LOCK:=0;	2: IF LOCK=1 THEN LOCK:=0;
3:	3:

$$S_{\text{JM1}} = [0..3] \times [0..3] \times \mathbb{Z} \times \mathbb{Z}$$

$$\begin{aligned}
\forall pc_1\ pc_2\ lock\ x.\ & R_{\text{JM1}}\ (0, pc_2, 0, x)\ \ (1, pc_2, 1, x)\ \ \wedge\\
& R_{\text{JM1}}\ (1, pc_2, lock, x)\ (2, pc_2, lock, 1)\ \wedge\\
& R_{\text{JM1}}\ (2, pc_2, 1, x)\ \ (3, pc_2, 0, x)\ \ \wedge\\
& R_{\text{JM1}}\ (pc_1, 0, 0, x)\ \ (pc_1, 1, 1, x)\ \ \wedge\\
& R_{\text{JM1}}\ (pc_1, 1, lock, x)\ (pc_1, 2, lock, 2)\ \wedge\\
& R_{\text{JM1}}\ (pc_1, 2, 1, x)\ \ (pc_1, 3, 0, x)
\end{aligned}$$

- An atomic property:
  - $\texttt{NotAt11}(pc_1, pc_2, lock, x) = \neg((pc_1 = 1) \wedge (pc_2 = 1))$
- A non-atomic property:
  - all states reachable from $(0, 0, 0, 0)$ satisfy `NotAt11`
  - this is an example of a reachability property

State satisfying `NotAt11` unreachable from $(0,0,0,0)$

Thread 1	Thread 2
`0: IF LOCK=0 THEN LOCK:=1;`	`0: IF LOCK=0 THEN LOCK:=1;`
`1: X:=1;`	`1: X:=2;`
`2: IF LOCK=1 THEN LOCK:=0;`	`2: IF LOCK=1 THEN LOCK:=0;`
`3:`	`3:`

$R_{\text{JM1}}\ (0, pc_2, 0, x)$ $(1, pc_2, 1, x)$
$R_{\text{JM1}}\ (1, pc_2, lock, x)$ $(2, pc_2, lock, 1)$
$R_{\text{JM1}}\ (2, pc_2, 1, x)$ $(3, pc_2, 0, x)$

$R_{\text{JM1}}\ (pc_1, 0, 0, x)$ $(pc_1, 1, 1, x)$
$R_{\text{JM1}}\ (pc_1, 1, lock, x)$ $(pc_1, 2, lock, 2)$
$R_{\text{JM1}}\ (pc_1, 2, 1, x)$ $(pc_1, 3, 0, x)$

- $\texttt{NotAt11}(pc_1, pc_2, lock, x) = \neg((pc_1 = 1) \wedge (pc_2 = 1))$
- Can only reach $pc_1 = 1 \wedge pc_2 = 1$ via:

  $R_{\text{JM1}}\ (0, pc_2, 0, x)$ $(1, pc_2, 1, x)$ i.e. a step $R_{\text{JM1}}\ (0, 1, 0, x)$ $(1, 1, 1, x)$
  $R_{\text{JM1}}\ (pc_1, 0, 0, x)$ $(pc_1, 1, 1, x)$ i.e. a step $R_{\text{JM1}}\ (1, 0, 0, x)$ $(1, 1, 1, x)$
- But:

  $R_{\text{JM1}}\ (pc_1, pc_2, lock, x)\ (pc_1', pc_2', lock', x') \wedge pc_1'{=}0 \wedge pc_2'{=}1 \Rightarrow lock'{=}1$
  $\wedge$
  $R_{\text{JM1}}\ (pc_1, pc_2, lock, x)\ (pc_1', pc_2', lock', x') \wedge pc_1'{=}1 \wedge pc_2'{=}0 \Rightarrow lock'{=}1$
- So can never reach $(0,1,0,x)$ or $(1,0,0,x)$
- So can't reach $(1,1,1,x)$, hence never $(pc_1 = 1) \wedge (pc_2 = 1)$
- Hence all states reachable from $(0,0,0,0)$ satisfy `NotAt11`

Reachability

- $R\ s\ s'$ means s' reachable from s in one step
- $R^n\ s\ s'$ means s' reachable from s in n steps

  $R^0\ s\ s' \quad = \quad (s = s')$
  $R^{n+1}\ s\ s' \quad = \quad \exists s''.\ R\ s\ s'' \wedge R^n\ s''\ s'$
- $R^*\ s\ s'$ means s' reachable from s in finite steps

  $R^*\ s\ s' = \exists n.\ R^n\ s\ s'$
- Note: $R^*\ s\ s' \Leftrightarrow \exists \pi\ n.\ \text{Path}\ R\ s\ \pi \wedge (s' = \pi(n))$
- The set of states reachable from s is $\{s' \mid R^*\ s\ s'\}$
- Verification problem: all states reachable from s satisfy p
  - verify truth of $\forall s'.\ R^*\ s\ s' \Rightarrow p(s')$
  - e.g. all states reachable from $(0,0,0,0)$ satisfy `NotAt11`
  - i.e. $\forall s'.\ R^*_{\text{JM1}}\ (0,0,0,0)\ s' \Rightarrow \texttt{NotAt11}(s')$

Model Checking a Simple Property

Models and model checking

- Assume a model (S, R)
- Assume also a set $S_0 \subseteq S$ of initial states
- Assume also a set AP of atomic properties
  - allows different models to have same atomic properties
- Assume a labelling function $L : S \to \mathcal{P}(AP)$
  - $p \in L(s)$ means "s labelled with p" or "p true of s"
  - previously properties were functions $p : S \to \mathbb{B}$
  - now $p \in AP$ is distinguished from $\lambda s.\ p \in L(s)$
  - assume $\texttt{T}, \texttt{F} \in AP$ with $\forall s$: $\texttt{T} \in L(s)$ and $\texttt{F} \notin L(s)$
- A *Kripke structure* is a tuple (S, S_0, R, L)
  - often the term "model" is used for a Kripke structure
  - i.e. a model is (S, S_0, R, L) rather than just (S, R)
- Model checking computes whether $(S, S_0, R, L) \models \phi$
  - ϕ is a property expressed in a property language
  - informally $M \models \phi$ means "wff ϕ is true in model M"

Minimal property language: ϕ is **AG** p where $p \in AP$

Our first temporal operator in a very restricted form so far.

- Consider properties ϕ of form **AG** p where $p \in AP$
  - "**AG**" stands for "Always Globally"
  - from CTL (same meaning, more elaborately expressed)
- Assume $M = (S, S_0, R, L)$
- Reachable states of M are $\{s' \mid \exists s \in S_0.\ R^*\ s\ s'\}$
  - i.e. the set of states reachable from an initial state
- Define Reachable $M = \{s' \mid \exists s \in S_0.\ R^*\ s\ s'\}$
- $M \models$ **AG** p means p true of all reachable states of M
- If $M = (S, S_0, R, L)$ then $M \models \phi$ formally defined by:

$$M \models \mathbf{AG}\, p \Leftrightarrow \forall s'.\ s' \in \text{Reachable } M \Rightarrow p \in L(s')$$

Model checking $M \models$ **AG** p

- $M \models \mathbf{AG}\, p \Leftrightarrow \forall s'.\ s' \in \text{Reachable } M \Rightarrow p \in L(s')$
  $\Leftrightarrow \text{Reachable } M \subseteq \{s' \mid p \in L(s')\}$

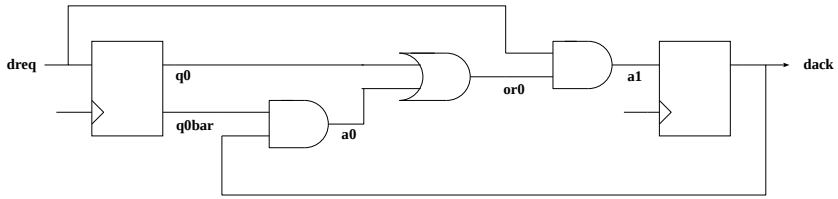
  checked by:
  - first computing Reachable M
  - then checking p true of all its members
- Let $\mathcal{S}$ abbreviate $\{s' \mid \exists s \in S_0.\ R^*\ s\ s'\}$ (i.e. Reachable M)
- Compute $\mathcal{S}$ iteratively: $\mathcal{S} = \mathcal{S}_0 \cup \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_n \cup \cdots$
  - i.e. $\mathcal{S} = \bigcup_{n=0}^{\infty} \mathcal{S}_n$
  - where: $\mathcal{S}_0 = S_0$ (set of initial states)
  - and inductively: $\mathcal{S}_{n+1} = \mathcal{S}_n \cup \{s' \mid \exists s \in \mathcal{S}_n \wedge R\ s\ s'\}$
- Clearly $\mathcal{S}_0 \subseteq \mathcal{S}_1 \subseteq \cdots \subseteq \mathcal{S}_n \subseteq \cdots$
- Hence if $\mathcal{S}_m = \mathcal{S}_{m+1}$ then $\mathcal{S} = \mathcal{S}_m$
- Algorithm: compute $\mathcal{S}_0, \mathcal{S}_1, \ldots$, until no change;
  check all members of computed set labelled with p

compute $\mathcal{S}_0, \mathcal{S}_1, \ldots$, until no change;
check p holds of all members of computed set

- Does the algorithm terminate?
  - yes, if set of states is finite, because then no infinite chains:
    $\mathcal{S}_0 \subset \mathcal{S}_1 \subset \cdots \subset \mathcal{S}_n \subset \cdots$
- How to represent $\mathcal{S}_0, \mathcal{S}_1, \ldots$?
  - explicitly (e.g. lists or something more clever)
  - symbolic expression
- Huge literature on calculating set of reachable states

Example: RCV

- Recall the handshake circuit:



- State represented by a triple of Booleans $(dreq, q0, dack)$
- A model of RCV is M_{RCV} where:

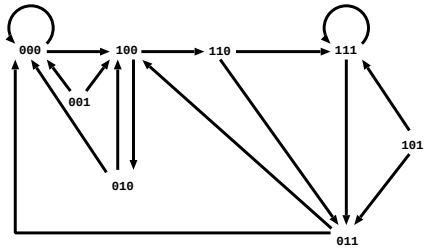
  $M = (S_{\text{RCV}}, \{(1,1,1)\}, R_{\text{RCV}}, L_{\text{RCV}})$

  and

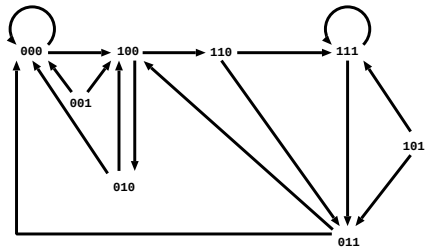
  $R_{\text{RCV}}\ (dreq, q0, dack)\ (dreq', q0', dack') =$
  $(q0' = dreq) \wedge (dack' = (dreq \wedge (q0 \vee dack)))$
- AP and labelling function L_{RCV} discussed later

RCV state transition diagram

- Possible states for RCV:
  $\{000, 001, 010, 011, 100, 101, 110, 111\}$
  where $b_2 b_1 b_0$ denotes state
  $\texttt{dreq} = b_2 \ \wedge \ \texttt{q0} = b_1 \ \wedge \ \texttt{dack} = b_0$
- Graph of the transition relation:



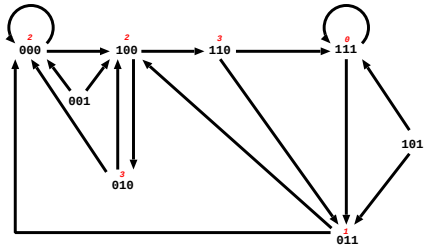
Computing Reachable M_{RCV}



- Define:

$$\begin{aligned}
\mathcal{S}_0 &= \{b_2 b_1 b_0 \mid b_2 b_1 b_0 \in \{111\}\} \\
&= \{111\} \\
\mathcal{S}_{i+1} &= \mathcal{S}_i \ \cup \ \{s' \mid \exists s \in \mathcal{S}_i.\ R_{\text{RCV}}\ s\ s'\ \} \\
&= \mathcal{S}_i \ \cup \ \{b_2' b_1' b_0' \mid \\
&\qquad \exists b_2 b_1 b_0 \in \mathcal{S}_i.\ (b_1' = b_2) \ \wedge \ (b_0' = b_2 \wedge (b_1 \vee b_0))\}
\end{aligned}$$

Computing Reachable M_{RCV} (continued)



- Compute:

$$\begin{aligned}
\mathcal{S}_0 &= \{111\} \\
\mathcal{S}_1 &= \{111\} \ \cup \ \{011\} \\
&= \{111, 011\} \\
\mathcal{S}_2 &= \{111, 011\} \ \cup \ \{000, 100\} \\
&= \{111, 011, 000, 100\} \\
\mathcal{S}_3 &= \{111, 011, 000, 100\} \ \cup \ \{010, 110\} \\
&= \{111, 011, 000, 100, 010, 110\} \\
\mathcal{S}_i &= \mathcal{S}_3 \quad (i > 3)
\end{aligned}$$

- Hence Reachable $M_{\text{RCV}} = \{111, 011, 000, 100, 010, 110\}$

Model checking $M_{\text{RCV}} \models \mathbf{AG}\, p$

- $M = (S_{\text{RCV}}, \{111\}, R_{\text{RCV}}, L_{\text{RCV}})$
- To check $M_{\text{RCV}} \models \mathbf{AG}\, p$
  - compute Reachable $M_{\text{RCV}} = \{111, 011, 000, 100, 010, 110\}$
  - check Reachable $M_{\text{RCV}} \subseteq \{s \mid p \in L_{\text{RCV}}(s)\}$
  - i.e. check if $s \in$ Reachable M_{RCV} then $p \in L_{\text{RCV}}(s)$, i.e.:
    $p \in L_{\text{RCV}}(111) \wedge$
    $p \in L_{\text{RCV}}(011) \wedge$
    $p \in L_{\text{RCV}}(000) \wedge$
    $p \in L_{\text{RCV}}(100) \wedge$
    $p \in L_{\text{RCV}}(010) \wedge$
    $p \in L_{\text{RCV}}(110)$
- Example
  - if $AP = \{\texttt{A}, \texttt{B}\}$
  - and $L_{\text{RCV}}(s) \ = \ \textit{if } s \in \{001, 101\} \textit{ then } \{\texttt{A}\} \textit{ else } \{\texttt{B}\}$
  - then $M_{\text{RCV}} \models \mathbf{AG}\, \texttt{A}$ is not true, but $M_{\text{RCV}} \models \mathbf{AG}\, \texttt{B}$ is true

Symbolic Boolean model checking of reachability

- Assume states are n-tuples of Booleans $(b_1, \ldots, b_n)$
  - $b_i \in \mathbb{B} = \{true, false\}$ $(= \{1, 0\})$
  - $S = \mathbb{B}^n$, so S is finite: 2^n states
- Assume n distinct Boolean variables: v_1,...,v_n
  - e.g. if $n = 3$ then could have $v_1 = \mathtt{x}$, $v_2 = \mathtt{y}$, $v_3 = \mathtt{z}$
- Boolean formula $f(v_1, \ldots, v_n)$ represents a subset of S
  - $f(v_1, \ldots, v_n)$ only contains variables v_1,...,v_n
  - $f(b_1, \ldots, b_n)$ denotes result of substituting b_i for v_i
  - $f(v_1, \ldots, v_n)$ determines $\{(b_1, \ldots, b_n) \mid f(b_1, \ldots, b_n) \Leftrightarrow true\}$
- Example $\neg(\mathtt{x} = \mathtt{y})$ represents $\{(true, false), (false, true)\}$
- Transition relations also represented by Boolean formulae
  - e.g. R_{RCV} represented by:
    $(q0' = dreq) \wedge (dack' = (dreq \wedge (q0 \vee (\neg q0 \wedge dack))))$

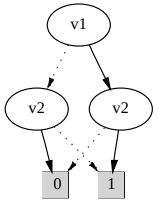
Symbolically represent Boolean formulae as BDDs

- Key features of Binary Decision Diagrams (BDDs):
  - canonical (given a variable ordering)
  - efficient to manipulate
- Variables:
  $\mathtt{v} = \mathtt{if\ v\ then\ 1\ else\ 0}$
  $\neg\mathtt{v} = \mathtt{if\ v\ then\ 0\ else\ 1}$
- Example: BDDs of variable $\mathtt{v}$ and $\neg\mathtt{v}$

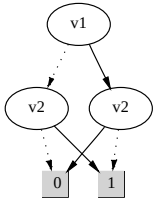
- Example: BDDs of $\mathtt{v1} \wedge \mathtt{v2}$ and $\mathtt{v1} \vee \mathtt{v2}$

More BDD examples

- BDD of $\mathtt{v1} = \mathtt{v2}$



- BDD of $\mathtt{v1} \neq \mathtt{v2}$

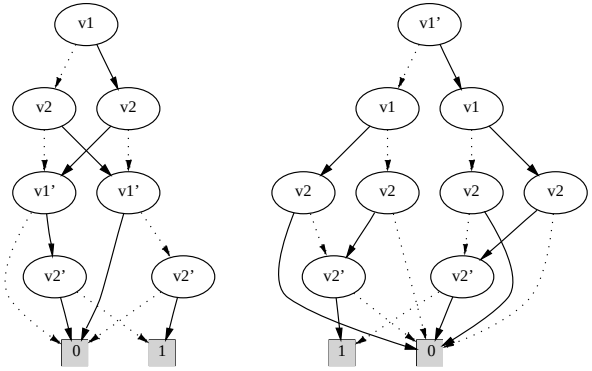


BDD of a transition relation

- BDDs of

  $(\mathtt{v1}' = (\mathtt{v1} = \mathtt{v2})) \wedge (\mathtt{v2}' = (\mathtt{v1} \neq \mathtt{v2}))$

  with two different variable orderings



- **Exercise:** draw BDD of R_{RCV}

Standard BDD operations

- If formulae f_1, f_2 represents sets S_1, S_2, respectively then $f_1 \wedge f_2$, $f_1 \vee f_2$ represent $S_1 \cap S_2$, $S_1 \cup S_2$ respectively
- Standard algorithms compute Boolean operation on BDDs
- Abbreviate $(v_1, \ldots, v_n)$ to $\vec{v}$
- If $f(\vec{v})$ represents S
  and $g(\vec{v}, \vec{v}')$ represents $\{(\vec{v}, \vec{v}') \mid R\ \vec{v}\ \vec{v}')\}$
  then $\exists \vec{u}.\ f(\vec{u}) \wedge g(\vec{u}, \vec{v})$ represents $\{\vec{v} \mid \exists \vec{u}.\ \vec{u} \in S \wedge R\ \vec{u}\ \vec{v}\}$
- Can compute BDD of $\exists \vec{u}.\ h(\vec{u}, \vec{v})$ from BDD of $h(\vec{u}, \vec{v})$
  - e.g. BDD of $\exists v_1.\ h(v_1, v_2)$ is BDD of $h(\mathtt{T}, v_2) \vee h(\mathtt{F}, v_2)$
- From BDD of formula $f(v_1, \ldots, v_n)$ can compute b_1, …, b_n such that if $v_1 = b_1$, …, $v_n = b_n$ then $f(b_1, \ldots, b_n) \Leftrightarrow \mathit{true}$
  - b_1, …, b_n is a satisfying assignment (SAT problem)
  - used for counterexample generation (see later)

Reachable States via BDDs

- Assume $M = (S, S_0, R, L)$ and $S = \mathbb{B}^n$
- Represent R by Boolean formulae $g(\vec{v}, \vec{v'})$
- Iteratively define formula $f_n(\vec{v})$ representing $\mathcal{S}_n$

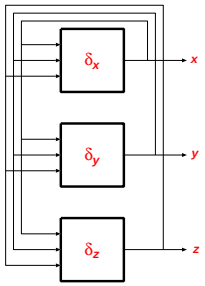
$$\begin{aligned} f_0(\vec{v}) &= \text{formula representing } S_0 \\ f_{n+1}(\vec{v}) &= f_n(\vec{v}) \vee (\exists \vec{u}.\ f_n(\vec{u}) \wedge g(\vec{u}, \vec{v})) \end{aligned}$$

- Let $\mathcal{B}_0$, $\mathcal{B}_R$ be BDDs representing $f_0(\vec{v})$, $g(\vec{v}, \vec{v'})$
- Iteratively compute BDDs $\mathcal{B}_n$ representing f_n

$$\mathcal{B}_{n+1} = \mathcal{B}_n \ \underline{\vee}\ (\underline{\exists \vec{u}.\ \mathcal{B}_n[\vec{u}/\vec{v}]} \ \underline{\wedge}\ \underline{\mathcal{B}_R[\vec{u}, \vec{v}/\vec{v}, \vec{v}']})$$

  - efficient using (blue underlined) standard BDD algorithms (renaming, conjunction, disjunction, quantification)
  - BDD $\mathcal{B}_n$ only contains variables $\vec{v}$: represents $\mathcal{S}_n \subseteq S$
- At each iteration check $\mathcal{B}_{n+1} = \mathcal{B}_n$ efficient using BDDs
  - when $\mathcal{B}_{n+1} = \mathcal{B}_n$ can conclude $\mathcal{B}_n$ represents Reachable M
  - we call this BDD $\mathcal{B}_M$ in a later slide (i.e. $\mathcal{B}_M = \mathcal{B}_n$)

Example BDD optimisation: disjunctive partitioning



Three state transition functions in parallel

$$\delta_x, \delta_y, \delta_z : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \to \mathbb{B}$$

- Transition relation (asynchronous interleaving semantics):

$$\begin{aligned} &R\ (x, y, z)\ (x', y', z') = \\ &\quad (x' = \delta_x(x, y, z) \ \wedge\ y' = y \ \wedge\ z' = z) \vee \\ &\quad (x' = x \ \wedge\ y' = \delta_y(x, y, z) \ \wedge\ z' = z) \vee \\ &\quad (x' = x \ \wedge\ y' = y \ \wedge\ z' = \delta_z(x, y, z)) \end{aligned}$$

Avoiding building big BDDs

- Transition relation for three transition functions in parallel

$$\begin{aligned} &R(x, y, z)\ (x', y', z') = \\ &\quad (x' = \delta_x(x, y, z) \ \wedge\ y' = y \ \wedge\ z' = z) \vee \\ &\quad (x' = x \ \wedge\ y' = \delta_y(x, y, z) \ \wedge\ z' = z) \vee \\ &\quad (x' = x \ \wedge\ y' = y \ \wedge\ z' = \delta_z(x, y, z)) \end{aligned}$$

- Recall symbolic iteration:

$$f_{n+1}(\vec{v}) = f_n(\vec{v}) \vee (\exists \vec{u}.\ f_n(\vec{u}) \wedge g(\vec{u}, \vec{v}))$$

- For this particular R (see next slide):

$$\begin{aligned} &f_{n+1}(x, y, z) \\ &= f_n(x, y, z) \vee (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x}, \overline{y}, \overline{z}) \wedge R\ (\overline{x}, \overline{y}, \overline{z})\ (x, y, z)) \\ &= f_n(x, y, z) \qquad\qquad\qquad\qquad\qquad\qquad \vee \\ &\quad (\exists \overline{x}.\ f_n(\overline{x}, y, z) \ \wedge\ x = \delta_x(\overline{x}, y, z)) \vee \\ &\quad (\exists \overline{y}.\ f_n(x, \overline{y}, z) \ \wedge\ y = \delta_y(x, \overline{y}, z)) \vee \\ &\quad (\exists \overline{z}.\ f_n(x, y, \overline{z}) \ \wedge\ z = \delta_z(x, y, \overline{z})) \end{aligned}$$

- Don't need to calculate BDD of R!

Disjunctive partitioning – Exercise: understand this

$$\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x},\overline{y},\overline{z}) \wedge R\ (\overline{x},\overline{y},\overline{z})\ (x,y,z)$$

$$= \exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x},\overline{y},\overline{z}) \wedge ((x = \delta_x(\overline{x},\overline{y},\overline{z}) \wedge y = \overline{y} \wedge z = \overline{z}) \vee (x = \overline{x} \wedge y = \delta_y(\overline{x},\overline{y},\overline{z}) \wedge z = \overline{z}) \vee (x = \overline{x} \wedge y = \overline{y} \wedge z = \delta_z(\overline{x},\overline{y},\overline{z})))$$

$$= (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x},\overline{y},\overline{z}) \wedge x = \delta_x(\overline{x},\overline{y},\overline{z}) \wedge y = \overline{y} \wedge z = \overline{z}) \vee (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x},\overline{y},\overline{z}) \wedge x = \overline{x} \wedge y = \delta_y(\overline{x},\overline{y},\overline{z}) \wedge z = \overline{z}) \vee (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x},\overline{y},\overline{z}) \wedge x = \overline{x} \wedge y = \overline{y} \wedge z = \delta_z(\overline{x},\overline{y},\overline{z}))$$

$$= (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(\overline{x},y,z) \wedge x = \delta_x(\overline{x},y,z) \wedge y = \overline{y} \wedge z = \overline{z}) \vee (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(x,\overline{y},z) \wedge x = \overline{x} \wedge y = \delta_y(x,\overline{y},z) \wedge z = \overline{z}) \vee (\exists \overline{x}\ \overline{y}\ \overline{z}.\ f_n(x,y,\overline{z}) \wedge x = \overline{x} \wedge y = \overline{y} \wedge z = \delta_z(x,y,\overline{z}))$$

$$= ((\exists \overline{x}.\ f_n(\overline{x},y,z) \wedge x{=}\delta_x(\overline{x},y,z)) \wedge (\exists \overline{y}.\ y{=}\overline{y}) \wedge (\exists \overline{z}.\ z{=}\overline{z})) \vee ((\exists \overline{x}.\ x{=}\overline{x}) \wedge (\exists \overline{y}.\ f_n(x,\overline{y},z) \wedge y{=}\delta_y(x,\overline{y},z)) \wedge (\exists \overline{z}.\ z{=}\overline{z})) \vee ((\exists \overline{x}.\ x{=}\overline{x}) \wedge (\exists \overline{y}.\ y{=}\overline{y}) \wedge (\exists \overline{z}.\ f_n(x,y,\overline{z}) \wedge z{=}\delta_z(x,y,\overline{z})))$$

$$= (\exists \overline{x}.\ f_n(\overline{x},y,z) \wedge x = \delta_x(\overline{x},y,z)) \vee (\exists \overline{y}.\ f_n(x,\overline{y},z) \wedge y = \delta_y(x,\overline{y},z)) \vee (\exists \overline{z}.\ f_n(x,y,\overline{z}) \wedge z = \delta_z(x,y,\overline{z}))$$

Verification and counterexamples

- Typical safety question:
  - is property p true in all reachable states?
  - i.e. check $M \models \mathbf{AG}\, p$
  - i.e. is $\forall s.\ s \in \text{Reachable}\ M \Rightarrow p\ s$
- Check using BDDs
  - compute BDD $\mathcal{B}_M$ of Reachable M
  - compute BDD $\mathcal{B}_p$ of $p(\vec{v})$
  - check if BDD of $\mathcal{B}_M \Rightarrow \mathcal{B}_p$ is the single node $\boxed{1}$
- Valid because *true* represented by a unique BDD (canonical property)
- If BDD is not $\boxed{1}$ can get counterexample

Generating counterexamples (general idea)

BDD algorithms can find satisfying assignments (SAT)

- Suppose not all reachable states of model M satisfy p
- i.e. $\exists s \in \text{Reachable}\ M.\ \neg(p(s))$
- Set of reachable state $\mathcal{S}$ given by: $\mathcal{S} = \bigcup_{n=0}^{\infty} \mathcal{S}_n$
- Iterate to find least n such that $\exists s \in \mathcal{S}_n.\ \neg(p(s))$
- Use SAT to find b_n such that $b_n \in \mathcal{S}_n \wedge \neg(p(b_n))$
- Use SAT to find b_{n-1} such that $b_{n-1} \in \mathcal{S}_{n-1} \wedge R\ b_{n-1}\ b_n$
- Use SAT to find b_{n-2} such that $b_{n-2} \in \mathcal{S}_{n-2} \wedge R\ b_{n-2}\ b_{n-1}$

  ⋮
- Iterate to find $b_0, b_1, \ldots, b_{n-1}, b_n$ where $b_i \in \mathcal{S}_i \wedge R\ b_{i-1}\ b_i$
- Then $b_0\ b_1\ \cdots\ b_{n-1}\ b_n$ is a path to a counterexample

Use SAT to find s_{n-1} such that $s_{n-1} \in \mathcal{S}_{n-1} \wedge R\ s_{n-1}\ s_n$

- Suppose states s, s' symbolically represented by $\vec{v}$, $\vec{v'}$
- Suppose BDD $\mathcal{B}_i$ represents $\vec{v} \in \mathcal{S}_i$ $(1 \le i \le n)$
- Suppose BDD $\mathcal{B}_R$ represents $R\ \vec{v}\ \vec{v'}$
- Then BDD $(\mathcal{B}_{n-1} \mathbin{\underline{\wedge}} \mathcal{B}_R[\vec{b}_n/\vec{v'}])$ represents $\vec{v} \in \mathcal{S}_{n-1} \wedge R\ \vec{v}\ \vec{b}_n$
- Use SAT to find a valuation $\vec{b}_{n-1}$ for $\vec{v}$
- Then BDD $(\mathcal{B}_{n-1} \mathbin{\underline{\wedge}} \mathcal{B}_R[\vec{b}_n/\vec{v'}])[\vec{b}_{n-1}/\vec{v}]$ represents $\vec{b}_{n-1} \in \mathcal{S}_{n-1} \wedge R\ \vec{b}_{n-1}\ \vec{b}_n$

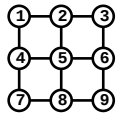
Generating counterexamples with BDDs

BDD algorithms can find satisfying assignments (SAT)

- $M = (S, S_0, R, L)$ and $\mathcal{B}_0$, $\mathcal{B}_1$, ..., $\mathcal{B}_M$, $\mathcal{B}_R$, $\mathcal{B}_p$ as earlier
- Suppose $\mathcal{B}_M \underline{\Rightarrow} \mathcal{B}_p$ is not 1
- Must exist a state $s \in$ Reachable M such that $\neg(p\ s)$
- Let $\mathcal{B}_{\neg p}$ be the BDD representing $\neg(p\ \vec{v})$
- Iterate to find first n such that $\mathcal{B}_n \underline{\wedge} \mathcal{B}_{\neg p}$
- Use SAT to find $\vec{b}_n$ such that $(\mathcal{B}_n \underline{\wedge} \mathcal{B}_{\neg p})\underline{[\vec{b}_n/\vec{v}]}$
- Use SAT to find $\vec{b}_{n-1}$ such that $(\mathcal{B}_{n-1} \underline{\wedge} \mathcal{B}_R\underline{[\vec{b}_n/\vec{v'}]})\underline{[\vec{b}_{n-1}/\vec{v}]}$
- For $0 < i < n$ find $\vec{b}_{i-1}$ such that $(\mathcal{B}_{i-1} \underline{\wedge} \mathcal{B}_R\underline{[\vec{b}_i/\vec{v'}]})\underline{[\vec{b}_{i-1}/\vec{v}]}$
- $\vec{b}_0$,...,$\vec{b}_i$,...,$\vec{b}_n$ is a counterexample trace
- Sometimes can use partitioning to avoid constructing $\mathcal{B}_R$

Example (from an exam)

Consider a 3x3 array of 9 switches



Suppose each switch 1,2,...,9 can either be on or off, and that toggling any switch will automatically toggle all its immediate neighbours. For example, toggling switch 5 will also toggle switches 2, 4, 6 and 8, and toggling switch 6 will also toggle switches 3, 5 and 9.

(a) Devise a state space [4 marks] and transition relation [6 marks] to represent the behaviour of the array of switches

You are given the problem of getting from an initial state in which even numbered switches are on and odd numbered switches are off, to a final state in which all the switches are off.

(b) Write down predicates on your state space that characterises the initial [2 marks] and final [2 marks] states.

(c) Explain how you might use a model checker to find a sequences of switches to toggle to get from the initial to final state. [6 marks]

You are not expected to actually solve the problem, but only to explain how to represent it in terms of model checking.

Solution

A state is a vector `(v1,v2,v3,v4,v5,v6,v7,v8,v9)`, where `vi` $\in \mathbb{B}$

A transition relation `Trans` is then defined by:

```
Trans(v1,v2,v3,v4,v5,v6,v7,v8,v9)(v1',v2',v3',v4',v5',v6',v7',v8',v9')
 = ((v1'=¬v1)∧(v2'=¬v2)∧(v3'=v3)∧(v4'=¬v4)∧(v5'=v5)∧
    (v6'=v6)∧(v7'=v7)∧(v8'=v8)∧(v9'=v9))        (toggle switch 1)
 ∨ ((v1'=¬v1)∧(v2'=¬v2)∧(v3'=¬v3)∧(v4'=v4)∧(v5'=¬v5)∧
    (v6'=v6)∧(v7'=v7)∧(v8'=v8)∧(v9'=v9))        (toggle switch 2)
 ∨ ((v1'=v1)∧(v2'=¬v2)∧(v3'=¬v3)∧(v4'=v4)∧(v5'=v5)∧
    (v6'=¬v6)∧(v7'=v7)∧(v8'=v8)∧(v9'=v9))       (toggle switch 3)
 ∨ ((v1'=v1)∧(v2'=v2)∧(v3'=v3)∧(v4'=¬v4)∧(v5'=¬v5)∧
    (v6'=v6)∧(v7'=¬v7)∧(v8'=v8)∧(v9'=v9))       (toggle switch 4)
 ∨ ((v1'=v1)∧(v2'=¬v2)∧(v3'=v3)∧(v4'=¬v4)∧(v5'=¬v5)∧
    (v6'=¬v6)∧(v7'=v7)∧(v8'=¬v8)∧(v9'=v9))      (toggle switch 5)
 ∨ ((v1'=v1)∧(v2'=v2)∧(v3'=¬v3)∧(v4'=v4)∧(v5'=¬v5)∧
    (v6'=¬v6)∧(v7'=v7)∧(v8'=v8)∧(v9'=¬v9))      (toggle switch 6)
 ∨ ((v1'=v1)∧(v2'=v2)∧(v3'=v3)∧(v4'=¬v4)∧(v5'=v5)∧
    (v6'=v6)∧(v7'=¬v7)∧(v8'=¬v8)∧(v9'=v9))      (toggle switch 7)
 ∨ ((v1'=v1)∧(v2'=v2)∧(v3'=v3)∧(v4'=v4)∧(v5'=¬v5)∧
    (v6'=v6)∧(v7'=¬v7)∧(v8'=¬v8)∧(v9'=¬v9))     (toggle switch 8)
 ∨ ((v1'=v1)∧(v2'=v2)∧(v3'=v3)∧(v4'=v4)∧(v5'=v5)∧
    (v6'=¬v6)∧(v7'=v7)∧(v8'=¬v8)∧(v9'=¬v9))     (toggle switch 9)
```


Solution (continued)

Predicates `Init`, `Final` characterising the initial and final states, respectively, are defined by:

```
Init(v1,v2,v3,v4,v5,v6,v7,v8,v9) =
 ¬v1 ∧ v2 ∧ ¬v3 ∧ v4 ∧ ¬v5 ∧ v6 ∧ ¬v7 ∧ v8 ∧ ¬v9

Final(v1,v2,v3,v4,v5,v6,v7,v8,v9) =
 ¬v1 ∧ ¬v2 ∧ ¬v3 ∧ ¬v4 ∧ ¬v5 ∧ ¬v6 ∧ ¬v7 ∧ ¬v8 ∧ ¬v9
```

Model checkers can find counter-examples to properties, and sequences of transitions from an initial state to a counter-example state. Thus we could use a model checker to find a trace to a counter-example to the property that

```
¬Final(v1,v2,v3,v4,v5,v6,v7,v8,v9)
```

More Interesting Properties (LTL)

More General Properties

- $\forall s{\in}S_0.\forall s'.R^*\ s\ s' \Rightarrow p\ s'$ says p true in all reachable states
- Might want to verify other properties
  1. `DeviceEnabled` holds infinitely often along every path
  2. From any state it is possible to get to a state where `Restart` holds
  3. After a three or more consecutive occurrences of `Req` there will eventually be an `Ack`
- Temporal logic can express such properties
- There are several temporal logics in use
  - LTL is good for the first example above
  - CTL is good for the second example
  - PSL is good for the third example
- Model checking:
  - Emerson, Clarke & Sifakis: Turing Award 2008
  - widely used in industry: first hardware, later software

Temporal logic (originally called “tense logic”)

A. N. Prior
1914-1969

Originally devised for investigating: “the relationship between tense and modality attributed to the Megarian philosopher Diodorus Cronus (ca. 340-280 BCE)”.

Mary Prior, his wife, recalls “I remember his waking me one night [in 1953], coming and sitting on my bed, ... and saying he thought one could make a formalised tense logic”.

- Temporal logic: deductive system for reasoning about time
  - temporal formulae for expressing temporal statements
  - deductive system for proving theorems
- Temporal logic model checking
  - uses semantics to check truth of temporal formulae in models
- Temporal logic proof systems also important in CS
  - use pioneered by Amir Pnueli (1996 Turing Award)
  - not considered in this course

Recommended: `http://plato.stanford.edu/entries/prior/`

Temporal logic formulae (statements)

- Many different languages of temporal statements
  - linear time (LTL)
  - branching time (CTL)
  - finite intervals (SEREs)
  - industrial languages (PSL, SVA)
- Prior used linear time, Kripke suggested branching time:

  ... we perhaps should not regard time as a linear series ... there are several possibilities for what the next moment may be like - and for each possible next moment, there are several possibilities for the moment after that. Thus the situation takes the form, not of a linear sequence, but of a 'tree'. [Saul Kripke, 1958 (aged 17, still at school)]
- CS issues different from philosophical issues
  - Moshe Vardi: “Branching vs. Linear Time: Final Showdown”

http://www.computer.org/portal/web/awards/Vardi

Moshe Vardi
www.computer.org

"For fundamental and lasting contributions to the development of logic as a unifying foundational framework and a tool for modeling computational systems"

2011 Harry H. Goode Memorial Award Recipient

Linear Temporal Logic (LTL)

- Grammar of *well-formed formulae* (wff) ϕ

$$\begin{array}{lcll} \phi & ::= & p & \text{(Atomic formula: } p \in AP\text{)} \\ & | & \neg\phi & \text{(Negation)} \\ & | & \phi_1 \vee \phi_2 & \text{(Disjunction)} \\ & | & \mathbf{X}\phi & \text{(successor)} \\ & | & \mathbf{F}\phi & \text{(sometimes)} \\ & | & \mathbf{G}\phi & \text{(always)} \\ & | & [\phi_1\ \mathbf{U}\ \phi_2] & \text{(Until)} \end{array}$$

- Details differ from Prior's tense logic – but similar ideas
- Semantics define when ϕ true in model M
  - where $M = (S, S_0, R, L)$ – a Kripke structure
  - notation: $M \models \phi$ means ϕ true in model M
  - model checking algorithms compute this (when decidable)
  - previously we only discussed the case $\phi = \mathbf{AG}\, p$

While use temporal operators at all?

Instead of the complexity of new temporal operators, why not make time explicit and just write:

- $\exists t.\phi(t)$ instead of $\mathbf{F}\phi$
- $\forall t.\phi(t)$ instead of $\mathbf{G}\phi$
- $\phi[t+1/t]$ instead of $\mathbf{X}\phi$

along with parameterising all Atomic Formulae with time?

Answer: it's harder to reason about quantifiers and arithmetic on time than it is to reason about temporal operators (which abstract from the above concrete notion of time).

$M \models \phi$ means "wff ϕ is true in model M"

- If $M = (S, S_0, R, L)$ then

  π is an M-path starting from s iff Path $R\ s\ \pi$

- If $M = (S, S_0, R, L)$ then we define $M \models \phi$ to mean:

  ϕ is true on all M-paths starting from a member of S_0

- We will define $[\![\phi]\!]_M(\pi)$ to mean

  ϕ is true on the M-path π

- Thus $M \models \phi$ will be formally defined by:

  $M \models \phi \ \Leftrightarrow\ \forall \pi\ s.\ s \in S_0 \wedge \text{Path}\ R\ s\ \pi \Rightarrow [\![\phi]\!]_M(\pi)$

- It remains to actually define $[\![\phi]\!]_M$ for all wffs ϕ

Definition of $[\![\phi]\!]_M(\pi)$

- $[\![\phi]\!]_M(\pi)$ is the application of function $[\![\phi]\!]_M$ to path π
  - thus $[\![\phi]\!]_M : (\mathbb{N} \to S) \to \mathbb{B}$
- Let $M = (S, S_0, R, L)$

  $[\![\phi]\!]_M$ is defined by structural induction on ϕ

$$\begin{array}{ll} [\![p]\!]_M(\pi) & = p \in L(\pi\ 0) \\ [\![\neg\phi]\!]_M(\pi) & = \neg([\![\phi]\!]_M(\pi)) \\ [\![\phi_1 \vee \phi_2]\!]_M(\pi) & = [\![\phi_1]\!]_M(\pi) \ \vee\ [\![\phi_2]\!]_M(\pi) \\ [\![\mathbf{X}\phi]\!]_M(\pi) & = [\![\phi]\!]_M(\pi{\downarrow}1) \\ [\![\mathbf{F}\phi]\!]_M(\pi) & = \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i) \\ [\![\mathbf{G}\phi]\!]_M(\pi) & = \forall i.\ [\![\phi]\!]_M(\pi{\downarrow}i) \\ [\![[\phi_1\ \mathbf{U}\ \phi_2]]\!]_M(\pi) & = \exists i.\ [\![\phi_2]\!]_M(\pi{\downarrow}i)\ \wedge\ \forall j.\ j{<}i\ \Rightarrow\ [\![\phi_1]\!]_M(\pi{\downarrow}j) \end{array}$$

- We look at each of these semantic equations in turn

$$[\![p]\!]_M(\pi) = p(\pi\ 0)$$

- Assume $M = (S, S_0, R, L)$
- We have: $[\![p]\!]_M(\pi) = p \in L(\pi\ 0)$
  - p is an atomic property, i.e. $p \in AP$
  - $\pi : \mathbb{N} \to S$ so $\pi\ 0 \in S$
  - $\pi\ 0$ is the first state in path π
  - $p \in L(\pi\ 0)$ is *true* iff atomic property p holds of state $\pi\ 0$
- $[\![p]\!]_M(\pi)$ means p holds of the first state in path π
- $\mathtt{T}, \mathtt{F} \in AP$ with $\mathtt{T} \in L(s)$ and $\mathtt{F} \notin L(s)$ for all $s \in S$
  - $[\![\mathtt{T}]\!]_M(\pi)$ is always true
  - $[\![\mathtt{F}]\!]_M(\pi)$ is always false

$$[\![\neg\phi]\!]_M(\pi) = \neg([\![\phi]\!]_M(\pi))$$
$$[\![\phi_1 \vee \phi_2]\!]_M(\pi) = [\![\phi_1]\!]_M(\pi) \vee [\![\phi_2]\!]_M(\pi)$$

- $[\![\neg\phi]\!]_M(\pi) = \neg([\![\phi]\!]_M(\pi))$
  - $[\![\neg\phi]\!]_M(\pi)$ true iff $[\![\phi]\!]_M(\pi)$ is not true
- $[\![\phi_1 \vee \phi_2]\!]_M(\pi) = [\![\phi_1]\!]_M(\pi) \vee [\![\phi_2]\!]_M(\pi)$
  - $[\![\phi_1 \vee \phi_2]\!]_M(\pi)$ true iff $[\![\phi_1]\!]_M(\pi)$ is true or $[\![\phi_2]\!]_M(\pi)$ is true

$$[\![\mathbf{X}\phi]\!]_M(\pi) = [\![\phi]\!]_M(\pi{\downarrow}1)$$

- $[\![\mathbf{X}\phi]\!]_M(\pi) = [\![\phi]\!]_M(\pi{\downarrow}1)$
  - $\pi{\downarrow}1$ is π with the first state chopped off
    $\pi{\downarrow}1(0) = \pi(1+0) = \pi(1)$
    $\pi{\downarrow}1(1) = \pi(1+1) = \pi(2)$
    $\pi{\downarrow}1(2) = \pi(1+2) = \pi(3)$
    $\vdots$
- $[\![\mathbf{X}\phi]\!]_M(\pi)$ true iff $[\![\phi]\!]_M$ true *starting at the second state of* π

$$[\![\mathbf{F}\phi]\!]_M(\pi) = \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i)$$

- $[\![\mathbf{F}\phi]\!]_M(\pi) = \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i)$
  - $\pi{\downarrow}i$ is π with the first i states chopped off
    $\pi{\downarrow}i(0) = \pi(i+0) = \pi(i)$
    $\pi{\downarrow}i(1) = \pi(i+1)$
    $\pi{\downarrow}i(2) = \pi(i+2)$
    $\vdots$
  - $[\![\phi]\!]_M(\pi{\downarrow}i)$ true iff $[\![\phi]\!]_M$ true *starting i states along* π
- $[\![\mathbf{F}\phi]\!]_M(\pi)$ true iff $[\![\phi]\!]_M$ true *starting somewhere along* π
- "$\mathbf{F}\phi$" is read as "sometimes ϕ"

$$[\![\mathbf{G}\phi]\!]_M(\pi) = \forall i.\ [\![\phi]\!]_M(\pi{\downarrow}i)$$

- $[\![\mathbf{G}\phi]\!]_M(\pi) = \forall i.\ [\![\phi]\!]_M(\pi{\downarrow}i)$
  - $\pi{\downarrow}i$ is π with the first i states chopped off
  - $[\![\phi]\!]_M(\pi{\downarrow}i)$ true iff $[\![\phi]\!]_M$ true *starting i states along π*
- $[\![\mathbf{G}\phi]\!]_M(\pi)$ true iff $[\![\phi]\!]_M$ true *starting anywhere along π*
- "$\mathbf{G}\phi$" is read as "always ϕ" or "globally ϕ"
- $M \models \mathbf{AG}\,p$ defined earlier: $M \models \mathbf{AG}\,p \Leftrightarrow M \models \mathbf{G}(p)$
- **G** is definable in terms of **F** and $\neg$: $\mathbf{G}\phi = \neg(\mathbf{F}(\neg\phi))$

$$\begin{aligned}
[\![\neg(\mathbf{F}(\neg\phi))]\!]_M(\pi) &= \neg([\![\mathbf{F}(\neg\phi)]\!]_M(\pi)) \\
&= \neg(\exists i.\ [\![\neg\phi]\!]_M(\pi{\downarrow}i)) \\
&= \neg(\exists i.\ \neg([\![\phi]\!]_M(\pi{\downarrow}i))) \\
&= \forall i.\ [\![\phi]\!]_M(\pi{\downarrow}i) \\
&= [\![\mathbf{G}\phi]\!]_M(\pi)
\end{aligned}$$

$$[\![[\phi_1\ \mathbf{U}\ \phi_2]]\!]_M(\pi) = \exists i.\ [\![\phi_2]\!]_M(\pi{\downarrow}i) \wedge \forall j.\ j{<}i \Rightarrow [\![\phi_1]\!]_M(\pi{\downarrow}j)$$

- $[\![[\phi_1\ \mathbf{U}\ \phi_2]]\!]_M(\pi) = \exists i.\ [\![\phi_2]\!]_M(\pi{\downarrow}i) \wedge \forall j.\ j{<}i \Rightarrow [\![\phi_1]\!]_M(\pi{\downarrow}j)$
  - $[\![\phi_2]\!]_M(\pi{\downarrow}i)$ true iff $[\![\phi_2]\!]_M$ true *starting i states along π*
  - $[\![\phi_1]\!]_M(\pi{\downarrow}j)$ true iff $[\![\phi_1]\!]_M$ true *starting j states along π*
- $[\![[\phi_1\ \mathbf{U}\ \phi_2]]\!]_M(\pi)$ is true iff
  $[\![\phi_2]\!]_M$ is true somewhere along π and up to then $[\![\phi_1]\!]_M$ is true
- "$[\phi_1\ \mathbf{U}\ \phi_2]$" is read as "$\phi_1$ until ϕ_2"
- **F** is definable in terms of $[-\ \mathbf{U}\ -]$: $\mathbf{F}\phi = [\mathtt{T}\ \mathbf{U}\ \phi]$

$$\begin{aligned}
&[\![[\mathtt{T}\ \mathbf{U}\ \phi]]\!]_M(\pi) \\
&= \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i) \wedge \forall j.\ j{<}i \Rightarrow [\![\mathtt{T}]\!]_M(\pi{\downarrow}j) \\
&= \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i) \wedge \forall j.\ j{<}i \Rightarrow \mathit{true} \\
&= \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i) \wedge \mathit{true} \\
&= \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i) \\
&= [\![\mathbf{F}\phi]\!]_M(\pi)
\end{aligned}$$

Review of Linear Temporal Logic (LTL)

- Grammar of *well-formed formulae* (wff) ϕ

ϕ	::=	p	(Atomic formula: $p \in AP$)
	|	$\neg\phi$	(Negation)
	|	$\phi_1 \vee \phi_2$	(Disjunction)
	|	$\mathbf{X}\phi$	(successor)
	|	$\mathbf{F}\phi$	(sometimes)
	|	$\mathbf{G}\phi$	(always)
	|	$[\phi_1\ \mathbf{U}\ \phi_2]$	(Until)

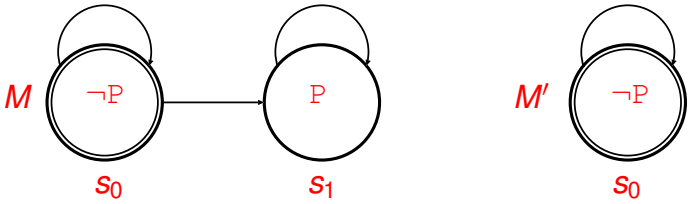
- $M \models \phi$ means ϕ holds on all M-paths
  - $M = (S, S_0, R, L)$
  - $[\![\phi]\!]_M(\pi)$ means ϕ is true on the M-path π
  - $M \models \phi \ \Leftrightarrow\ \forall \pi\ s.\ s \in S_0 \wedge \mathrm{Path}\ R\ s\ \pi \Rightarrow [\![\phi]\!]_M(\pi)$

LTL examples

- "`DeviceEnabled` holds infinitely often along every path"
  $\mathbf{G}(\mathbf{F}\ \mathtt{DeviceEnabled})$
- "Eventually the state becomes permanently `Done`"
  $\mathbf{F}(\mathbf{G}\ \mathtt{Done})$
- "Every `Req` is followed by an `Ack`"
  $\mathbf{G}(\mathtt{Req} \Rightarrow \mathbf{F}\ \mathtt{Ack})$
  Number of `Req` and `Ack` may differ - no counting
- "If `Enabled` infinitely often then `Running` infinitely often"
  $\mathbf{G}(\mathbf{F}\ \mathtt{Enabled}) \Rightarrow \mathbf{G}(\mathbf{F}\ \mathtt{Running})$
- "An upward-going lift at the second floor keeps going up if a passenger requests the fifth floor"
  $\mathbf{G}(\mathtt{AtFloor2} \wedge \mathtt{DirectionUp} \wedge \mathtt{RequestFloor5} \Rightarrow [\mathtt{DirectionUp}\ \mathbf{U}\ \mathtt{AtFloor5}])$

A property not expressible in LTL

- Let $AP = \{\mathtt{P}\}$ and consider models M and M' below



$M = (\{s_0, s_1\}, \{s_0\}, \{(s_0, s_0), (s_0, s_1), (s_1, s_1)\}, L)$
$M' = (\{s_0\}, \{s_0\}, \{(s_0, s_0)\}, L)$
where: $L = \lambda s.\ \mathit{if}\ s = s_0\ \mathit{then}\ \{\}\ \mathit{else}\ \{\mathtt{P}\}$

- Every M'-path is also an M-path
- So if ϕ true on every M-path then ϕ true on every M'-path
- Hence in LTL for any ϕ if $M \models \phi$ then $M' \models \phi$
- Consider $\phi_{\mathtt{P}} \Leftrightarrow$ "can always reach a state satisfying $\mathtt{P}$"
  - $\phi_{\mathtt{P}}$ holds in M but not in M'
  - but in LTL can't have $M \models \phi_{\mathtt{P}}$ and not $M' \models \phi_{\mathtt{P}}$
- hence $\phi_{\mathtt{P}}$ not expressible in LTL

LTL expressibility limitations

"can always reach a state satisfying $\mathtt{P}$"

- In LTL $M \models \phi$ says ϕ holds of all paths of M
- LTL formulae ϕ are *evaluated on paths* .... path formulae
- Want also to say that from any state there exists a path to some state satisfying p
  - $\forall s.\ \exists \pi.$ Path $R\ s\ \pi\ \wedge\ \exists i.\ p \in L(\pi(i))$
  - but this isn't expressible in LTL (see slide 213)

By contrast:

- CTL properties are *evaluated at a state* ... state formulae
  - they can talk about both some or all paths
  - starting from the state they are evaluated at

More Interesting Properties (CTL)

Computation Tree Logic (CTL)

- LTL formulae ϕ are evaluated on paths .... path formulae
- CTL formulae ψ are evaluated on states .. state formulae

- Syntax of CTL well-formed formulae:

$$\begin{array}{rcll} \psi & ::= & p & \text{(Atomic formula } p \in AP\text{)} \\ & | & \neg\psi & \text{(Negation)} \\ & | & \psi_1 \wedge \psi_2 & \text{(Conjunction)} \\ & | & \psi_1 \vee \psi_2 & \text{(Disjunction)} \\ & | & \psi_1 \Rightarrow \psi_2 & \text{(Implication)} \\ & | & \mathbf{AX}\psi & \text{(All successors)} \\ & | & \mathbf{EX}\psi & \text{(Some successors)} \\ & | & \mathbf{A}[\psi_1\ \mathbf{U}\ \psi_2] & \text{(Until – along all paths)} \\ & | & \mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2] & \text{(Until – along some path)} \end{array}$$

- (Some operators can be defined in terms of others)

Semantics of CTL

- Assume $M = (S, S_0, R, L)$ and then define:

$$
\begin{array}{lcl}
[\![p]\!]_M(s) & = & p \in L(s) \\
[\![\neg\psi]\!]_M(s) & = & \neg([\![\psi]\!]_M(s)) \\
[\![\psi_1 \wedge \psi_2]\!]_M(s) & = & [\![\psi_1]\!]_M(s) \ \wedge \ [\![\psi_2]\!]_M(s) \\
[\![\psi_1 \vee \psi_2]\!]_M(s) & = & [\![\psi_1]\!]_M(s) \ \vee \ [\![\psi_2]\!]_M(s) \\
[\![\psi_1 \Rightarrow \psi_2]\!]_M(s) & = & [\![\psi_1]\!]_M(s) \ \Rightarrow \ [\![\psi_2]\!]_M(s) \\
[\![\mathbf{AX}\psi]\!]_M(s) & = & \forall s'.\ R\ s\ s' \ \Rightarrow \ [\![\psi]\!]_M(s') \\
[\![\mathbf{EX}\psi]\!]_M(s) & = & \exists s'.\ R\ s\ s' \ \wedge \ [\![\psi]\!]_M(s') \\
[\![\mathbf{A}[\psi_1\ \mathbf{U}\ \psi_2]]\!]_M(s) & = & \forall \pi.\ \mathrm{Path}\ R\ s\ \pi \\
 & & \quad \Rightarrow \exists i.\ [\![\psi_2]\!]_M(\pi(i)) \\
 & & \qquad \wedge \\
 & & \qquad \forall j.\ j<i \ \Rightarrow \ [\![\psi_1]\!]_M(\pi(j)) \\
[\![\mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2]]\!]_M(s) & = & \exists \pi.\ \mathrm{Path}\ R\ s\ \pi \\
 & & \quad \wedge \ \exists i.\ [\![\psi_2]\!]_M(\pi(i)) \\
 & & \qquad \wedge \\
 & & \qquad \forall j.\ j<i \ \Rightarrow \ [\![\psi_1]\!]_M(\pi(j))
\end{array}
$$

The defined operator **AF**

- Define $\mathbf{AF}\psi = \mathbf{A}[\mathrm{T}\ \mathbf{U}\ \psi]$
- $\mathbf{AF}\psi$ true at s iff ψ true somewhere on every R-path from s

$$
\begin{array}{lcl}
[\![\mathbf{AF}\psi]\!]_M(s) & = & [\![\mathbf{A}[\mathrm{T}\ \mathbf{U}\ \psi]]\!]_M(s) \\
 & = & \forall \pi.\ \mathrm{Path}\ R\ s\ \pi \\
 & & \quad \Rightarrow \\
 & & \quad \exists i.\ [\![\psi]\!]_M(\pi(i)) \ \wedge \ \forall j.\ j < i \ \Rightarrow \ [\![\mathrm{T}]\!]_M(\pi(j)) \\
 & = & \forall \pi.\ \mathrm{Path}\ R\ s\ \pi \\
 & & \quad \Rightarrow \\
 & & \quad \exists i.\ [\![\psi]\!]_M(\pi(i)) \ \wedge \ \forall j.\ j < i \ \Rightarrow \ \mathit{true} \\
 & = & \forall \pi.\ \mathrm{Path}\ R\ s\ \pi \ \Rightarrow \ \exists i.\ [\![\psi]\!]_M(\pi(i))
\end{array}
$$

The defined operator **EF**

- Define $\mathbf{EF}\psi = \mathbf{E}[\mathrm{T}\ \mathbf{U}\ \psi]$
- $\mathbf{EF}\psi$ true at s iff ψ true somewhere on some R-path from s

$$
\begin{array}{lcl}
[\![\mathbf{EF}\psi]\!]_M(s) & = & [\![\mathbf{E}[\mathrm{T}\ \mathbf{U}\ \psi]]\!]_M(s) \\
 & = & \exists \pi.\ \mathrm{Path}\ R\ s\ \pi \\
 & & \quad \wedge \\
 & & \quad \exists i.\ [\![\psi]\!]_M(\pi(i)) \ \wedge \ \forall j.\ j < i \ \Rightarrow \ [\![\mathrm{T}]\!]_M(\pi(j)) \\
 & = & \exists \pi.\ \mathrm{Path}\ R\ s\ \pi \\
 & & \quad \wedge \\
 & & \quad \exists i.\ [\![\psi]\!]_M(\pi(i)) \ \wedge \ \forall j.\ j < i \ \Rightarrow \ \mathit{true} \\
 & = & \exists \pi.\ \mathrm{Path}\ R\ s\ \pi \ \wedge \ \exists i.\ [\![\psi]\!]_M(\pi(i))
\end{array}
$$

- "can reach a state satisfying p" is $\mathbf{EF}\ p$

The defined operator **AG**

- Define $\mathbf{AG}\psi = \neg\mathbf{EF}(\neg\psi)$
- $\mathbf{AG}\psi$ true at s iff ψ true everywhere on every R-path from s

$$
\begin{array}{lcl}
[\![\mathbf{AG}\psi]\!]_M(s) & = & [\![\neg\mathbf{EF}(\neg\psi)]\!]_M(s) \\
 & = & \neg([\![\mathbf{EF}(\neg\psi)]\!]_M(s)) \\
 & = & \neg(\exists \pi.\ \mathrm{Path}\ R\ s\ \pi \wedge \exists i.\ [\![\neg\psi]\!]_M(\pi(i))) \\
 & = & \neg(\exists \pi.\ \mathrm{Path}\ R\ s\ \pi \wedge \exists i.\ \neg[\![\psi]\!]_M(\pi(i))) \\
 & = & \forall \pi.\ \neg(\mathrm{Path}\ R\ s\ \pi \wedge \exists i.\ \neg[\![\psi]\!]_M(\pi(i))) \\
 & = & \forall \pi.\ \neg\mathrm{Path}\ R\ s\ \pi \vee \neg(\exists i.\ \neg[\![\psi]\!]_M(\pi(i))) \\
 & = & \forall \pi.\ \neg\mathrm{Path}\ R\ s\ \pi \vee \forall i.\ \neg\neg[\![\psi]\!]_M(\pi(i)) \\
 & = & \forall \pi.\ \neg\mathrm{Path}\ R\ s\ \pi \vee \forall i.\ [\![\psi]\!]_M(\pi(i)) \\
 & = & \forall \pi.\ \mathrm{Path}\ R\ s\ \pi \Rightarrow \forall i.\ [\![\psi]\!]_M(\pi(i))
\end{array}
$$

- $\mathbf{AG}\psi$ means ψ true at all reachable states
- $[\![\mathbf{AG}(p)]\!]_M(s) \ \equiv \ \forall s'.\ R^*\ s\ s' \ \Rightarrow \ p \in L(s')$
- "can always reach a state satisfying p" is $\mathbf{AG}(\mathbf{EF}\ p)$

The defined operator **EG**

- Define $\mathbf{EG}\psi = \neg\mathbf{AF}(\neg\psi)$
- $\mathbf{EG}\psi$ true at s iff ψ true everywhere on some R-path from s

$$\begin{aligned}
[\![\mathbf{EG}\psi]\!]_M(s) &= [\![\neg\mathbf{AF}(\neg\psi)]\!]_M(s) \\
&= \neg([\![\mathbf{AF}(\neg\psi)]\!]_M(s)) \\
&= \neg(\forall\pi.\ \text{Path}\ R\ s\ \pi \Rightarrow \exists i.\ [\![\neg\psi]\!]_M(\pi(i))) \\
&= \neg(\forall\pi.\ \text{Path}\ R\ s\ \pi \Rightarrow \exists i.\ \neg[\![\psi]\!]_M(\pi(i))) \\
&= \exists\pi.\ \neg(\text{Path}\ R\ s\ \pi \Rightarrow \exists i.\ \neg[\![\psi]\!]_M(\pi(i))) \\
&= \exists\pi.\ \text{Path}\ R\ s\ \pi \wedge \neg(\exists i.\ \neg[\![\psi]\!]_M(\pi(i))) \\
&= \exists\pi.\ \text{Path}\ R\ s\ \pi \wedge \forall i.\ \neg\neg[\![\psi]\!]_M(\pi(i)) \\
&= \exists\pi.\ \text{Path}\ R\ s\ \pi \wedge \forall i.\ [\![\psi]\!]_M(\pi(i))
\end{aligned}$$

The defined operator $\mathbf{A}[\psi_1\ \mathbf{W}\ \psi_2]$

- $\mathbf{A}[\psi_1\ \mathbf{W}\ \psi_2]$ is a 'partial correctness' version of $\mathbf{A}[\psi_1\ \mathbf{U}\ \psi_2]$
- It is true at s if along all R-paths from s:
  - ψ_1 always holds on the path, or
  - ψ_2 holds sometime on the path, and until it does ψ_1 holds
- Define

$$\begin{aligned}
&[\![\mathbf{A}[\psi_1\ \mathbf{W}\ \psi_2]]\!]_M(s) \\
&= [\![\neg\mathbf{E}[(\psi_1\wedge\neg\psi_2)\ \mathbf{U}\ (\neg\psi_1\wedge\neg\psi_2)]]\!]_M(s) \\
&= \neg[\![\mathbf{E}[(\psi_1\wedge\neg\psi_2)\ \mathbf{U}\ (\neg\psi_1\wedge\neg\psi_2)]]\!]_M(s) \\
&= \neg(\exists\pi.\ \text{Path}\ R\ s\ \pi \\
&\qquad \wedge \\
&\qquad \exists i.\ [\![\neg\psi_1\wedge\neg\psi_2]\!]_M(\pi(i)) \\
&\qquad\quad \wedge \\
&\qquad\quad \forall j.\ j<i \Rightarrow [\![\psi_1\wedge\neg\psi_2]\!]_M(\pi(j)))
\end{aligned}$$

- Exercise: understand the next two slides!

$\mathbf{A}[\psi_1\ \mathbf{W}\ \psi_2]$ continued (1)

- Continuing:

$$\begin{aligned}
&\neg(\exists\pi.\ \text{Path}\ R\ s\ \pi \\
&\qquad \wedge \\
&\qquad \exists i.\ [\![\neg\psi_1\wedge\neg\psi_2]\!]_M(\pi(i)) \wedge \forall j.\ j<i \Rightarrow [\![\psi_1\wedge\neg\psi_2]\!]_M(\pi(j))) \\
&= \forall\pi.\ \neg(\text{Path}\ R\ s\ \pi \\
&\qquad\quad \wedge \\
&\qquad\quad \exists i.\ [\![\neg\psi_1\wedge\neg\psi_2]\!]_M(\pi(i)) \wedge \forall j.\ j<i \Rightarrow [\![\psi_1\wedge\neg\psi_2]\!]_M(\pi(j))) \\
&= \forall\pi.\ \text{Path}\ R\ s\ \pi \\
&\qquad \Rightarrow \\
&\qquad \neg(\exists i.\ [\![\neg\psi_1\wedge\neg\psi_2]\!]_M(\pi(i)) \wedge \forall j.\ j<i \Rightarrow [\![\psi_1\wedge\neg\psi_2]\!]_M(\pi(j))) \\
&= \forall\pi.\ \text{Path}\ R\ s\ \pi \\
&\qquad \Rightarrow \\
&\qquad \forall i.\ \neg[\![\neg\psi_1\wedge\neg\psi_2]\!]_M(\pi(i)) \vee \neg(\forall j.\ j<i \Rightarrow [\![\psi_1\wedge\neg\psi_2]\!]_M(\pi(j)))
\end{aligned}$$

$\mathbf{A}[\psi_1\ \mathbf{W}\ \psi_2]$ continued (2)

- Continuing:

$$\begin{aligned}
&= \forall\pi.\ \text{Path}\ R\ s\ \pi \\
&\qquad \Rightarrow \\
&\qquad \forall i.\ \neg[\![\neg\psi_1\wedge\neg\psi_2]\!]_M(\pi(i)) \vee \neg(\forall j.\ j<i \Rightarrow [\![\psi_1\wedge\neg\psi_2]\!]_M(\pi(j))) \\
&= \forall\pi.\ \text{Path}\ R\ s\ \pi \\
&\qquad \Rightarrow \\
&\qquad \forall i.\ \neg(\forall j.\ j<i \Rightarrow [\![\psi_1\wedge\neg\psi_2]\!]_M(\pi(j))) \vee \neg[\![\neg\psi_1\wedge\neg\psi_2]\!]_M(\pi(i)) \\
&= \forall\pi.\ \text{Path}\ R\ s\ \pi \\
&\qquad \Rightarrow \\
&\qquad \forall i.\ (\forall j.\ j<i \Rightarrow [\![\psi_1\wedge\neg\psi_2]\!]_M(\pi(j))) \Rightarrow [\![\psi_1\vee\psi_2]\!]_M(\pi(i))
\end{aligned}$$

- Exercise: explain why this is $[\![\mathbf{A}[\psi_1\ \mathbf{W}\ \psi_2]]\!]_M(s)$?
  - this exercise illustrates the subtlety of writing CTL!

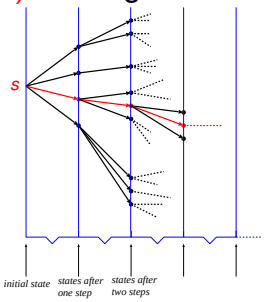
Sanity check: $\mathbf{A}[\psi\ \mathbf{W}\ \mathrm{F}]\ =\ \mathbf{AG}\ \psi$

- From last slide:
  $[\![\mathbf{A}[\psi_1\ \mathbf{W}\ \psi_2]]\!]_M(s)$
  $= \forall\pi.\ \mathrm{Path}\ R\ s\ \pi$
  $\Rightarrow\ \forall i.\ (\forall j.\ j{<}i \Rightarrow [\![\psi_1 \wedge \neg\psi_2]\!]_M(\pi(j))) \Rightarrow [\![\psi_1 \vee \psi_2]\!]_M(\pi(i))$
- Set ψ_1 to ψ and ψ_2 to F:
  $[\![\mathbf{A}[\psi\ \mathbf{W}\ \mathrm{F}]]\!]_M(s)$
  $= \forall\pi.\ \mathrm{Path}\ R\ s\ \pi$
  $\Rightarrow\ \forall i.\ (\forall j.\ j{<}i\ \Rightarrow\ [\![\psi \wedge \neg\mathrm{F}]\!]_M(\pi(j)))\ \Rightarrow\ [\![\psi \vee \mathrm{F}]\!]_M(\pi(i))$
- Simplify:
  $[\![\mathbf{A}[\psi\ \mathbf{W}\ \mathrm{F}]]\!]_M(s)$
  $= \forall\pi.\ \mathrm{Path}\ R\ s\ \pi \Rightarrow \forall i.\ (\forall j.\ j{<}i \Rightarrow [\![\psi]\!]_M(\pi(j))) \Rightarrow [\![\psi]\!]_M(\pi(i))$
- By induction on i:
  $[\![\mathbf{A}[\psi\ \mathbf{W}\ \mathrm{F}]]\!]_M(s)\ =\ \forall\pi.\ \mathrm{Path}\ R\ s\ \pi\ \Rightarrow\ \forall i.\ [\![\psi]\!]_M(\pi(i))$

- Exercises
  1. Describe the property: $\mathbf{A}[\mathrm{T}\ \mathbf{W}\ \psi]$.
  2. Describe the property: $\neg\mathbf{E}[\neg\psi_2\ \mathbf{U}\ \neg(\psi_1 \vee \psi_2)]$.
  3. Define $\mathbf{E}[\psi_1\ \mathbf{W}\ \psi_2] = \mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2] \vee \mathbf{EG}\psi_1$.
     Describe the property: $\mathbf{E}[\psi_1\ \mathbf{W}\ \psi_2]$?

Recall model behaviour computation tree

- Atomic properties are true or false of individual states
- General properties are true or false of whole behaviour
- Behaviour of (S, R) starting from $s \in S$ as a tree:



- A path is shown in red
- Properties may look at all paths, or just a single path
  - CTL: Computation Tree Logic (all paths from a state)
  - LTL: Linear Temporal Logic (a single path)

Summary of CTL operators (primitive + defined)

- CTL formulae:

p	(Atomic formula - $p \in AP$)
$\neg\psi$	(Negation)
$\psi_1 \wedge \psi_2$	(Conjunction)
$\psi_1 \vee \psi_2$	(Disjunction)
$\psi_1 \Rightarrow \psi_2$	(Implication)
$\mathbf{AX}\psi$	(All successors)
$\mathbf{EX}\psi$	(Some successors)
$\mathbf{AF}\psi$	(Somewhere – along all paths)
$\mathbf{EF}\psi$	(Somewhere – along some path)
$\mathbf{AG}\psi$	(Everywhere – along all paths)
$\mathbf{EG}\psi$	(Everywhere – along some path)
$\mathbf{A}[\psi_1\ \mathbf{U}\ \psi_2]$	(Until – along all paths)
$\mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2]$	(Until – along some path)
$\mathbf{A}[\psi_1\ \mathbf{W}\ \psi_2]$	(Unless – along all paths)
$\mathbf{E}[\psi_1\ \mathbf{W}\ \psi_2]$	(Unless – along some path)

Example CTL formulae

- $\mathbf{EF}(\mathit{Started} \wedge \neg \mathit{Ready})$
  It is possible to get to a state where Started holds but Ready does not hold
- $\mathbf{AG}(\mathit{Req} \Rightarrow \mathbf{AF}\mathit{Ack})$
  If a request Req occurs, then it will eventually be acknowledged by Ack
- $\mathbf{AG}(\mathbf{AF}\mathit{DeviceEnabled})$
  DeviceEnabled is always true somewhere along every path starting anywhere: i.e. DeviceEnabled holds infinitely often along every path
- $\mathbf{AG}(\mathbf{EF}\mathit{Restart})$
  From any state it is possible to get to a state for which Restart holds

  Can't be expressed in LTL!

More CTL examples (1)

- $\mathbf{AG}(Req \Rightarrow \mathbf{A}[Req\ \mathbf{U}\ Ack])$
  If a request Req occurs, then it continues to hold, until it is eventually acknowledged
- $\mathbf{AG}(Req \Rightarrow \mathbf{AX}(\mathbf{A}[\neg Req\ \mathbf{U}\ Ack]))$
  Whenever Req is true either it must become false on the next cycle and remains false until Ack, or Ack must become true on the next cycle
  Exercise: is the **AX** necessary?
- $\mathbf{AG}(Req \Rightarrow (\neg Ack \Rightarrow \mathbf{AX}(\mathbf{A}[Req\ \mathbf{U}\ Ack])))$
  Whenever Req is true and Ack is false then Ack will eventually become true and until it does Req will remain true
  Exercise: is the **AX** necessary?

More CTL examples (2)

- $\mathbf{AG}(Enabled \Rightarrow \mathbf{AG}(Start \Rightarrow \mathbf{A}[\neg Waiting\ \mathbf{U}\ Ack]))$
  If Enabled is ever true then if Start is true in any subsequent state then Ack will eventually become true, and until it does Waiting will be false
- $\mathbf{AG}(\neg Req_1 \wedge \neg Req_2 \Rightarrow \mathbf{A}[\neg Req_1 \wedge \neg Req_2\ \mathbf{U}\ (Start \wedge \neg Req_2)])$
  Whenever Req_1 and Req_2 are false, they remain false until Start becomes true with Req_2 still false
- $\mathbf{AG}(Req \Rightarrow \mathbf{AX}(Ack \Rightarrow \mathbf{AF}\ \neg Req))$
  If Req is true and Ack becomes true one cycle later, then eventually Req will become false

Some abbreviations

- $\mathbf{AX}_i\ \psi \equiv \underbrace{\mathbf{AX}(\mathbf{AX}(\cdots(\mathbf{AX}\ \psi)\cdots))}_{i \text{ instances of } \mathbf{AX}}$
  ψ is true on all paths i units of time later
- $\mathbf{ABF}_{i..j}\ \psi \equiv \mathbf{AX}_i \underbrace{(\psi \vee \mathbf{AX}(\psi \vee \cdots \mathbf{AX}(\psi \vee \mathbf{AX}\ \psi)\cdots))}_{j-i \text{ instances of } \mathbf{AX}}$
  ψ is true on all paths sometime between i units of time later and j units of time later
- $\mathbf{AG}(Req \Rightarrow \mathbf{AX}(Ack_1 \wedge \mathbf{ABF}_{1..6}(Ack_2 \wedge \mathbf{A}[Wait\ \mathbf{U}\ Reply])))$
  One cycle after Req, Ack_1 should become true, and then Ack_2 becomes true 1 to 6 cycles later and then eventually Reply becomes true, but until it does Wait holds from the time of Ack_2
- More abbreviations in 'Industry Standard' language PSL

CTL model checking

- For LTL path formulae ϕ recall that $M \models \phi$ is defined by:
  $$M \models \phi \Leftrightarrow \forall \pi\ s.\ s \in S_0 \wedge \text{Path}\ R\ s\ \pi \Rightarrow [\![\phi]\!]_M(\pi)$$
- For CTL state formulae ψ the definition of $M \models \psi$ is:
  $$M \models \psi \Leftrightarrow \forall s.\ s \in S_0 \Rightarrow [\![\psi]\!]_M(s)$$
- M common; LTL, CTL formulae and semantics $[\![\ ]\!]_M$ differ
- CTL model checking algorithm:
  - compute $\{s \mid [\![\psi]\!]_M(s) = true\}$ bottom up
  - check $S_0 \subseteq \{s \mid [\![\psi]\!]_M(s) = true\}$
  - symbolic model checking represents these sets as BDDs

CTL model checking: p, $\mathbf{AX}\psi$, $\mathbf{EX}\psi$

- For CTL formula ψ let $\{\psi\}_M = \{s \mid [\![\psi]\!]_M(s) = \mathit{true}\}$
- When unambiguous will write $\{\psi\}$ instead of $\{\psi\}_M$
- $\{p\} = \{s \mid p \in L(s)\}$
  - scan through set of states S marking states labelled with p
  - $\{p\}$ is set of marked states
- To compute $\{\mathbf{AX}\psi\}$
  - recursively compute $\{\psi\}$
  - marks those states all of whose successors are in $\{\psi\}$
  - $\{\mathbf{AX}\psi\}$ is the set of marked states
- To compute $\{\mathbf{EX}\psi\}$
  - recursively compute $\{\psi\}$
  - marks those states with at least one successor in $\{\psi\}$
  - $\{\mathbf{EX}\psi\}$ is the set of marked states

CTL model checking: $\{\mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2]\}$, $\{\mathbf{A}[\psi_1\ \mathbf{U}\ \psi_2]\}$

- To compute $\{\mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2]\}$
  - recursively compute $\{\psi_1\}$ and $\{\psi_2\}$
  - mark all states in $\{\psi_2\}$
  - mark all states in $\{\psi_1\}$ with a successor state that is marked
  - repeat previous line until no change
  - $\{\mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2]\}$ is set of marked states
- More formally: $\{\mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2]\} = \bigcup_{n=0}^{\infty}\{\mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2]\}_n$ where:

$$\begin{array}{lcl}\{\mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2]\}_0 & = & \{\psi_2\}\\ \{\mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2]\}_{n+1} & = & \{\mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2]\}_n \\ & & \cup \\ & & \{s \in \{\psi_1\} \mid \exists s' \in \{\mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2]\}_n.\ R\ s\ s'\}\end{array}$$

- $\{\mathbf{A}[\psi_1\ \mathbf{U}\ \psi_2]\}$ similar, but with a more complicated iteration
  - details omitted (see Huth and Ryan)

Example: checking **EF** p

- $\mathbf{EF}p = \mathbf{E}[\mathtt{T}\ \mathbf{U}\ p]$
  - holds if ψ holds along some path
- Note $\{\mathtt{T}\} = S$
- Let $\mathcal{S}_n = \{\mathbf{E}[\mathtt{T}\ \mathbf{U}\ p]\}_n$ then:

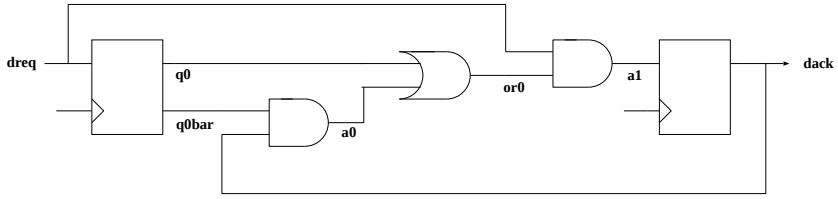
$$\begin{array}{lcl}\mathcal{S}_0 & = & \{\mathbf{E}[\mathtt{T}\ \mathbf{U}\ p]\}_0\\ & = & \{p\}\\ & = & \{s \mid p \in L(s)\}\end{array}$$

$$\begin{array}{lcl}\mathcal{S}_{n+1} & = & \mathcal{S}_n \cup \{s \in \{\mathtt{T}\} \mid \exists s' \in \{\mathbf{E}[\mathtt{T}\ \mathbf{U}\ p]\}_n.\ R\ s\ s'\}\\ & = & \mathcal{S}_n \cup \{s \mid \exists s' \in \mathcal{S}_n.\ R\ s\ s'\}\end{array}$$

  - mark all the states labelled with p
  - mark all with at least one marked successor
  - repeat until no change
  - $\{\mathbf{EF}\ p\}$ is set of marked states

Example: RCV

- Recall the handshake circuit:



- State represented by a triple of Booleans $(\mathit{dreq}, q0, \mathit{dack})$
- A model of RCV is M_{RCV} where:

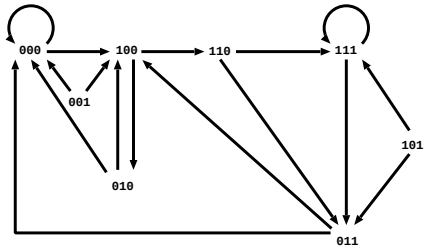
  $M = (S_{\mathrm{RCV}}, S_{0\,\mathrm{RCV}}, R_{\mathrm{RCV}}, L_{\mathrm{RCV}})$

  and

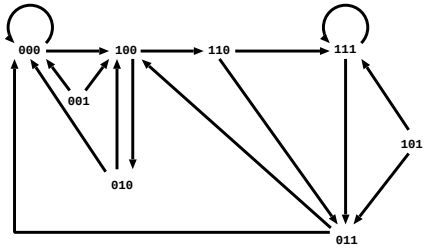
  $R_{\mathrm{RCV}}\ (\mathit{dreq}, q0, \mathit{dack})\ (\mathit{dreq}', q0', \mathit{dack}') =$
  $(q0' = \mathit{dreq}) \wedge (\mathit{dack}' = (\mathit{dreq} \wedge (q0 \vee \mathit{dack})))$

RCV state transition diagram

- Possible states for RCV:
  $\{000, 001, 010, 011, 100, 101, 110, 111\}$
  where $b_2 b_1 b_0$ denotes state
  $\texttt{dreq} = b_2 \ \wedge \ \texttt{q0} = b_1 \ \wedge \ \texttt{dack} = b_0$
- Graph of the transition relation:



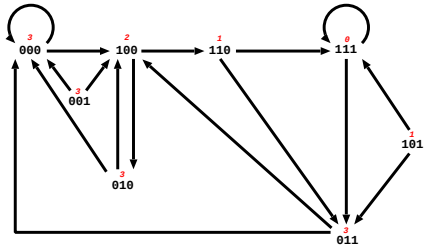
Computing {EF At111} where $\texttt{At111} \in L_{\text{RCV}}(s) \Leftrightarrow s = 111$



- Define:

$$\begin{aligned}
\mathcal{S}_0 &= \{s \mid \texttt{At111} \in L_{\text{RCV}}(s)\} \\
&= \{s \mid s = 111\} \\
&= \{111\} \\
\\
\mathcal{S}_{n+1} &= \mathcal{S}_n \cup \{s \mid \exists s' \in \mathcal{S}_n.\ \mathcal{R}(s, s')\} \\
&= \mathcal{S}_n \cup \{b_2 b_1 b_0 \mid \\
&\qquad \exists b_2' b_1' b_0' \in \mathcal{S}_n.\ (b_1' = b_2) \ \wedge \ (b_0' = b_2 \wedge (b_1 \vee b_0))\}
\end{aligned}$$

Computing {EF At111} (continued)



- Compute:

$$\begin{aligned}
\mathcal{S}_0 &= \{111\} \\
\mathcal{S}_1 &= \{111\} \cup \{101, 110\} \\
&= \{111, 101, 110\} \\
\mathcal{S}_2 &= \{111, 101, 110\} \cup \{100\} \\
&= \{111, 101, 110, 100\} \\
\mathcal{S}_3 &= \{111, 101, 110, 100\} \cup \{000, 001, 010, 011\} \\
&= \{111, 101, 110, 100, 000, 001, 010, 011\} \\
\mathcal{S}_n &= \mathcal{S}_3 \quad (n > 3)
\end{aligned}$$

- $\{\textbf{EF}\ \texttt{At111}\} = \mathbb{B}^3 = S_{\text{RCV}}$
- $M_{\text{RCV}} \models \textbf{EF}\ \texttt{At111} \Leftrightarrow S_{0_{\text{RCV}}} \subseteq S$

Symbolic model checking

- Represent sets of states with BDDs
- Represent Transition relation with a BDD
- If BDDs of $\{\psi\}$, $\{\psi_1\}$, $\{\psi_2\}$ are known, then:
  - BDDs of $\{\neg\psi\}$, $\{\psi_1 \ \wedge \ \psi_2\}$, $\{\psi_1 \ \vee \ \psi_2\}$, $\{\psi_1 \ \Rightarrow \ \psi_2\}$ computed using standard BDD algorithms
  - BDDs of $\{\textbf{AX}\psi\}$, $\{\textbf{EX}\psi\}$, $\{\textbf{A}[\psi_1\ \textbf{U}\ \psi_2]\}$, $\{\textbf{E}[\psi_1\ \textbf{U}\ \psi_2]\}$ computed using straightforward algorithms (see textbooks)
- Model checking CTL generalises reachable-states iteration

History of Model checking

- CTL model checking due to Emerson, Clarke & Sifakis
- Symbolic model checking due to several people:
  - Clarke & McMillan (idea usually credited to McMillan's PhD)
  - Coudert, Berthet & Madre
  - Pixley
- SMV (McMillan) is a popular symbolic model checker:

`http://www.cs.cmu.edu/~modelcheck/smv.html`	(original)
`http://www.kenmcmil.com/smv.html`	(Cadence extension by McMillan)
`http://nusmv.irst.itc.it/`	(new implementation)

- Other temporal logics
  - CTL*: combines CTL and LTL
  - Engineer friendly industrial languages: PSL, SVA

Expressibility of CTL

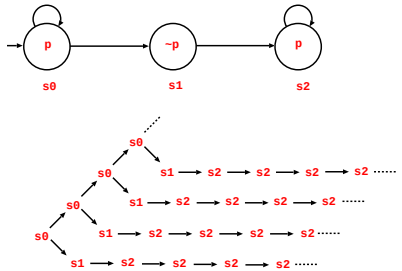
- Consider the property

  "on every path there is a point after which p is always true on that path"

- Consider

(($\star$) non-deterministically chooses `T` or `F`)

```
    0:  P:=1;
s0  1:  WHILE (⋆) DO SKIP;
s1  2:  P:=0;
s2  3:  P:=1;
    4:  WHILE T DO SKIP;
    5:
```



- Property true, but cannot be expressed in CTL
  - would need something like **AF**ψ
  - where ψ is something like "property p *true from now on*"
  - but in CTL ψ must start with a path quantifier **A** or **E**
  - cannot talk about current path, only about all or some paths
  - **AF**(**AG p**) is false (consider path `s0 s0 s0` $\cdots$)

LTL can express things CTL can't

- Recall:

$$[\![\mathbf{F}\phi]\!]_M(\pi) = \exists i.\ [\![\phi]\!]_M(\pi{\downarrow}i)$$
$$[\![\mathbf{G}\phi]\!]_M(\pi) = \forall i.\ [\![\phi]\!]_M(\pi{\downarrow}i)$$

- **FG**ϕ is true if there is a point after which ϕ is always true

$$\begin{aligned}[\![\mathbf{FG}\phi]\!]_M(\pi) &= [\![\mathbf{F}(\mathbf{G}(\phi))]\!]_M(\pi)\\ &= \exists m_1.\ [\![\mathbf{G}(\phi)]\!]_M(\pi{\downarrow}m_1)\\ &= \exists m_1.\ \forall m_2.\ [\![\phi]\!]_M((\pi{\downarrow}m_1){\downarrow}m_2)\\ &= \exists m_1.\ \forall m_2.\ [\![\phi]\!]_M(\pi{\downarrow}(m_1{+}m_2))\end{aligned}$$

- LTL can express things that CTL can't express
- Note: it's tricky to prove CTL can't express **FG**ϕ

CTL can express things that LTL can't express

- **AG**(**EF** *p*) says:

  "from every state it is possible to get to a state for which p holds"

- Can't say this in LTL (easy proof given earlier - slide 213)
- Consider disjunction:

  "on every path there is a point after which p is always true on that path
  or
  from every state it is possible to get to a state for which p holds"

- Can't say this in either CTL or LTL!
- CTL* combines CTL and LTL and can express this property

CTL*

- Both state formulae (ψ) and path formulae (ϕ)
  - state formulae ψ are true of a state s like CTL
  - path formulae ϕ are true of a path π like LTL
- Defined mutually recursively

ψ	$::=$	p	(Atomic formula)
	$\mid$	$\neg\psi$	(Negation)
	$\mid$	$\psi_1 \vee \psi_2$	(Disjunction)
	$\mid$	$\mathbf{A}\phi$	(All paths)
	$\mid$	$\mathbf{E}\phi$	(Some paths)
ϕ	$::=$	ψ	(Every state formula is a path formula)
	$\mid$	$\neg\phi$	(Negation)
	$\mid$	$\phi_1 \vee \phi_2$	(Disjunction)
	$\mid$	$\mathbf{X}\phi$	(Successor)
	$\mid$	$\mathbf{F}\phi$	(Sometimes)
	$\mid$	$\mathbf{G}\phi$	(Always)
	$\mid$	$[\phi_1\ \mathbf{U}\ \phi_2]$	(Until)

- CTL is CTL* with **X**, **F**, **G**, $[-\mathbf{U}-]$ preceded by **A** or **E**
- LTL consists of CTL* formulae of form $\mathbf{A}\phi$, where the only state formulae in ϕ are atomic

CTL* semantics

- Combines CTL state semantics with LTL path semantics:

$$
\begin{array}{ll}
[\![p]\!]_M(s) & = p \in L(s) \\
[\![\neg\psi]\!]_M(s) & = \neg([\![\psi]\!]_M(s)) \\
[\![\psi_1 \vee \psi_2]\!]_M(s) & = [\![\psi_1]\!]_M(s) \ \vee\ [\![\psi_2]\!]_M(s) \\
[\![\mathbf{A}\phi]\!]_M(s) & = \forall\pi.\ \mathrm{Path}\ R\ s\ \pi\ \Rightarrow\ \phi(\pi) \\
[\![\mathbf{E}\phi]\!]_M(s) & = \exists\pi.\ \mathrm{Path}\ R\ s\ \pi\ \wedge\ [\![\phi]\!]_M(\pi)
\end{array}
$$

$$
\begin{array}{ll}
[\![\psi]\!]_M(\pi) & = [\![\psi]\!]_M(\pi(0)) \\
[\![\neg\phi]\!]_M(\pi) & = \neg([\![\phi]\!]_M(\pi)) \\
[\![\phi_1 \vee \phi_2]\!]_M(\pi) & = [\![\phi_1]\!]_M(\pi) \ \vee\ [\![\phi_2]\!]_M(\pi) \\
[\![\mathbf{X}\phi]\!]_M(\pi) & = [\![\phi]\!]_M(\pi{\downarrow}1) \\
[\![\mathbf{F}\phi]\!]_M(\pi) & = \exists m.\ [\![\phi]\!]_M(\pi{\downarrow}m) \\
[\![\mathbf{G}\phi]\!]_M(\pi) & = \forall m.\ [\![\phi]\!]_M(\pi{\downarrow}m) \\
[\![[\phi_1\ \mathbf{U}\ \phi_2]]\!]_M(\pi) & = \exists i.\ [\![\phi_2]\!]_M(\pi{\downarrow}i) \wedge \forall j.\ j{<}i\ \Rightarrow\ [\![\phi_1]\!]_M(\pi{\downarrow}j)
\end{array}
$$

- Note $[\![\psi]\!]_M : S{\to}\mathbb{B}$ and $[\![\phi]\!]_M : (\mathbb{N}{\to}S){\to}\mathbb{B}$

LTL and CTL as CTL*

- As usual: $M = (S, S_0, R, L)$
- If ψ is a CTL* state formula: $M \models \psi \ \Leftrightarrow\ \forall s \in S_0.\ [\![\psi]\!]_M(s)$
- If ϕ is an LTL path formula then: $M \models_{\mathrm{LTL}} \phi \ \Leftrightarrow\ M \models_{\mathrm{CTL^*}} \mathbf{A}\phi$
- If R is total ($\forall s.\ \exists s'.\ R\ s\ s'$) then (exercise):
  $\forall s\ s'.\ R\ s\ s' \ \Leftrightarrow\ \exists\pi.\ \mathrm{Path}\ R\ s\ \pi \wedge (\pi(1) = s')$
- The meanings of CTL formulae are the same in CTL*

$$
\begin{array}{ll}
[\![\mathbf{A}(\mathbf{X}\psi)]\!]_M(s) & \\
= \forall\pi.\ \mathrm{Path}\ R\ s\ \pi \Rightarrow [\![\mathbf{X}\psi]\!]_M(\pi) & \\
= \forall\pi.\ \mathrm{Path}\ R\ s\ \pi \Rightarrow [\![\psi]\!]_M(\pi{\downarrow}1) & (\psi \text{ as path formula}) \\
= \forall\pi.\ \mathrm{Path}\ R\ s\ \pi \Rightarrow [\![\psi]\!]_M((\pi{\downarrow}1)(0)) & (\psi \text{ as state formula}) \\
= \forall\pi.\ \mathrm{Path}\ R\ s\ \pi \Rightarrow [\![\psi]\!]_M(\pi(1)) &
\end{array}
$$

$$
\begin{array}{ll}
[\![\mathbf{AX}\psi]\!]_M(s) & \\
= \forall s'.\ R\ s\ s' \ \Rightarrow\ [\![\psi]\!]_M(s') & \\
= \forall s'.\ (\exists\pi.\ \mathrm{Path}\ R\ s\ \pi \wedge (\pi(1) = s')) \ \Rightarrow\ [\![\psi]\!]_M(s') & \\
= \forall s'.\ \forall\pi.\ \mathrm{Path}\ R\ s\ \pi \wedge (\pi(1) = s') \ \Rightarrow\ [\![\psi]\!]_M(s') & \\
= \forall\pi.\ \mathrm{Path}\ R\ s\ \pi \ \Rightarrow\ [\![\psi]\!]_M(\pi(1)) &
\end{array}
$$

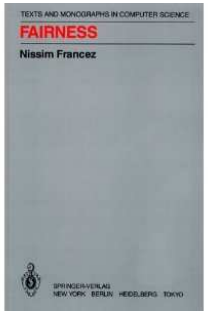
Exercise: do similar proofs for other CTL formulae

Fairness

- May want to assume system or environment is 'fair'
- Example 1: fair arbiter
  the arbiter doesn't ignore one of its requests forever
  - not every request need be granted
  - want to exclude infinite number of requests and no grant
- Example 2: reliable channel
  no message continuously transmitted but never received
  - not every message need be received
  - want to exclude an infinite number of sends and no receive

Handling fairness in CTL and LTL

- Consider:
  p holds infinitely often along a path then so does q
- In LTL is expressible as $\mathbf{G}(\mathbf{F}\ p) \Rightarrow \mathbf{G}(\mathbf{F}\ q)$
- Can't say this in CTL
  - why not – what's wrong with $\mathbf{AG}(\mathbf{AF}\ p) \Rightarrow \mathbf{AG}(\mathbf{AF}\ q)$?
  - in CTL* expressible as $\mathbf{A}\Big(\mathbf{G}(\mathbf{F}\ p) \Rightarrow \mathbf{G}(\mathbf{F}\ q)\Big)$
  - fair CTL model checking implemented in checking algorithm
  - fair LTL just a fairness assumption like $\mathbf{G}(\mathbf{F}\ p) \Rightarrow \cdots$
- Fairness is a tricky and subtle subject
  - many kinds of fairness:
    'weak fairness', 'strong fairness' etc
  - exist whole books on fairness



Richer Logics than LTL and CTL

Richer Logics than LTL and CTL

- Propositional modal μ-calculus
- Industrial Languages, e.g. PSL
- Modal Logics, where modes can be other than time in temporal logic. Examples:
  - Logics including possibility and necessity
  - Logics of belief: "P believes that Q believes F"
  - Logics of authentication, e.g. BAN logic

  More information can be found under "Modal Logic", "Doxastic logic" and "Burrows-Abadi-Needham logic" on Wikipedia.

Propositional modal μ-calculus

- You may learn this in *Topics in Concurrency*
- μ-calculus is an even more powerful property language
  - has fixed-point operators
  - both maximal and minimal fixed points
  - model checking consists of calculating fixed points
  - many logics (e.g. CTL*) can be translated into μ-calculus
- Strictly stronger than CTL*
  - expressibility strictly increases as allowed nesting increases
  - need fixed point operators nested 2 deep for CTL*
- The μ-calculus is very non-intuitive to use!
  - intermediate code rather than a practical property language
  - nice meta-theory and algorithms, but terrible usability!

PSL/Sugar

- Used for real-life hardware verification
- Combines together LTL and CTL
- SEREs: Sequential Extended Regular Expressions
- LTL – Foundation Language formulae
- CTL – Optional Branching Extension
- Relatively simple set of primitives + definitional extension
- Boolean, temporal, verification, modelling layers
- Semantics for static and dynamic verification (needs strong/weak distinction)
- You may learn more about this in *System-on-Chip Design*

Bisimulation equivalence: general idea

- M, M' bisimilar if they have 'corresponding executions'
  - to each step of M there is a corresponding step of M'
  - to each step of M' there is a corresponding step of M
- Bisimilar models satisfy same CTL* properties
- Bisimilar: same truth/falsity of model properties
- Simulation gives property-truth preserving abstraction (see later)

Bisimulation relations

- Let $R : S \to S \to \mathbb{B}$ and $R' : S' \to S' \to \mathbb{B}$ be transition relations
- B is a bisimulation relation between R and R' if:
  - $B : S \to S' \to \mathbb{B}$
  - $\forall s\ s'.\ B\ s\ s' \Rightarrow \forall s_1 \in S.\ R\ s\ s_1 \Rightarrow \exists s'_1.\ R'\ s'\ s'_1 \wedge B\ s_1\ s'_1$ (to each step of R there is a corresponding step of R')
  - $\forall s\ s'.\ B\ s\ s' \Rightarrow \forall s'_1 \in S.\ R'\ s'\ s'_1 \Rightarrow \exists s_1.\ R'\ s\ s_1 \wedge B\ s_1\ s'_1$ (to each step of R' there is a corresponding step of R)

Bisimulation equivalence: definition and theorem

- Let $M = (S, S_0, R, L)$ and $M' = (S', S'_0, R', L')$
- $M \equiv M'$ if:
  - there is a bisimulation B between R and R'
  - $\forall s_0 \in S_0.\ \exists s'_0 \in S'_0.\ B\ s_0\ s'_0$
  - $\forall s'_0 \in S'_0.\ \exists s_0 \in S_0.\ B\ s_0\ s'_0$
  - there is a bijection $\theta : AP \to AP'$
  - $\forall s\ s'.\ B\ s\ s' \Rightarrow L(s) = L'(s')$
- Theorem: if $M \equiv M'$ then for any CTL* state formula ψ: $M \models \psi \ \Leftrightarrow\ M' \models \psi$
- See Q14 in the Exercises

Abstraction and Abstraction Refinement

Abstraction

- Abstraction creates a simplification of a model
  - separate states may get merged
  - an abstract path can represent several concrete paths
- $M \preceq \overline{M}$ means $\overline{M}$ is an abstraction of M
  - to each step of M there is a corresponding step of $\overline{M}$
  - atomic properties of M correspond to atomic properties of $\overline{M}$
- Special case is when $\overline{M}$ is a subset of M such that:
  - $\overline{M} = (\overline{S_0}, \overline{S}, \overline{R}, \overline{L})$ and $M = (S_0, S, R, L)$
    $\overline{S} \subseteq S$
    $\overline{S_0} = S_0$
    $\forall s\ s' \in \overline{S}.\ \overline{R}\ s\ s' \Leftrightarrow R\ s\ s'$
    $\forall s \in \overline{S}.\ \overline{L}\ s = L\ s$
  - $\overline{S}$ contain all reachable states of M
    $\forall s \in \overline{S}.\ \forall s' \in S.\ R\ s\ s' \Rightarrow s' \in \overline{S}$
- All paths of M from initial states are $\overline{M}$-paths
  - hence for all CTL formulae ψ: $\overline{M} \models \psi \ \Rightarrow M \models \psi$

Recall JM1

Thread 1	Thread 2
0: IF LOCK=0 THEN LOCK:=1;	0: IF LOCK=0 THEN LOCK:=1;
1: X:=1;	1: X:=2;
2: IF LOCK=1 THEN LOCK:=0;	2: IF LOCK=1 THEN LOCK:=0;
3:	3:

- Two program counters, state: $(pc_1, pc_2, lock, x)$

  $S_{JM1} = [0..3] \times [0..3] \times \mathbb{Z} \times \mathbb{Z}$

  $R_{JM1}\ (0, pc_2, 0, x)\ \ (1, pc_2, 1, x)$
  $R_{JM1}\ (1, pc_2, lock, x)\ \ (2, pc_2, lock, 1)$
  $R_{JM1}\ (2, pc_2, 1, x)\ \ (3, pc_2, 0, x)$

  $R_{JM1}\ (pc_1, 0, 0, x)\ \ (pc_1, 1, 1, x)$
  $R_{JM1}\ (pc_1, 1, lock, x)\ \ (pc_1, 2, lock, 2)$
  $R_{JM1}\ (pc_1, 2, 1, x)\ \ (pc_1, 3, 0, x)$
- Assume $\texttt{NotAt11} \in L_{JM1}(pc_1, pc_2, lock, x) \Leftrightarrow \neg((pc_1 = 1) \wedge (pc_2 = 1))$
- Model $M_{JM1} = (S_{JM1}, \{(0,0,0,0)\}, R_{JM1}, L_{JM1})$
- S_{JM1} not finite, but actually $lock \in \{0,1\}, x \in \{0,1,2\}$
- Clear by inspection that $M_{JM1} \preceq \overline{M}_{JM1}$ where:

  $\overline{M}_{JM1} = (\overline{S}_{JM1}, \{(0,0,0,0)\}, \overline{R}_{JM1}, \overline{L}_{JM1})$
  - $\overline{S}_{JM1} = [0..3] \times [0..3] \times [0..1] \times [0..2]$
  - $\overline{R}_{JM1}$ is R_{JM1} restricted to arguments from $\overline{S}_{JM1}$
  - $\texttt{NotAt11} \in \overline{L}_{JM1}(pc_1, pc_2, lock, x) \Leftrightarrow \neg((pc_1 = 1) \wedge (pc_2 = 1))$
  - $\overline{L}_{JM1}$ is L_{JM1} restricted to arguments from $\overline{S}_{JM1}$

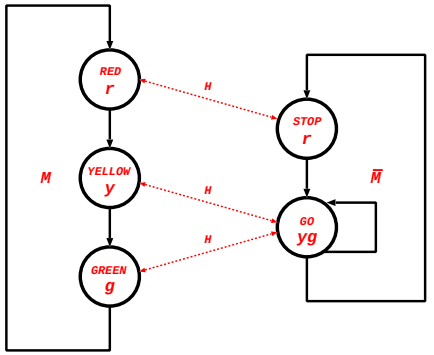
Simulation relations

- Let $R : S \to S \to \mathbb{B}$ and $\overline{R} : \overline{S} \to \overline{S} \to \mathbb{B}$ be transition relations
- H is a simulation relation between R and $\overline{R}$ if:
  - H is a relation between S and $\overline{S}$ – i.e. $H : S \to \overline{S} \to \mathbb{B}$
  - to each step of R there is a corresponding step of $\overline{R}$ – i.e.:
    $\forall s\ \overline{s}.\ H\ s\ \overline{s} \Rightarrow \forall s' \in S.\ R\ s\ s' \Rightarrow \exists \overline{s'} \in \overline{S}.\ \overline{R}\ \overline{s}\ \overline{s'} \wedge H\ s'\ \overline{s'}$
- Also need to consider abstraction of atomic properties
  - $H_{AP} : AP \to \overline{AP} \to \mathbb{B}$
  - details glossed over here

Simulation preorder: definition and theorem

- Let $M = (S, S_0, R, L)$ and $\overline{M} = (\overline{S}, \overline{S_0}, \overline{R}, \overline{L})$
- $M \preceq \overline{M}$ if:
  - there is a simulation H between R and $\overline{R}$
  - $\forall s_0 \in S_0.\ \exists \overline{s_0} \in \overline{S_0}.\ H\ s_0\ \overline{s_0}$
  - $\forall s\ \overline{s}.\ H\ s\ \overline{s} \Rightarrow L(s) = \overline{L}(\overline{s})$
- We *define* ACTL to be the subset of CTL without **E**-properties
  - e.g. **AG AF** p – from anywhere can always reach a p-state
  - useful for abstraction:
- Theorem: if $M \preceq \overline{M}$ then for any ACTL state formula ψ: $\overline{M} \models \psi \Rightarrow M \models \psi$
- If $\overline{M} \models \psi$ fails then cannot conclude $M \models \psi$ false

Example (Grumberg)



H a simulation

H RED STOP $\wedge$
H YELLOW GO $\wedge$
H GREEN GO

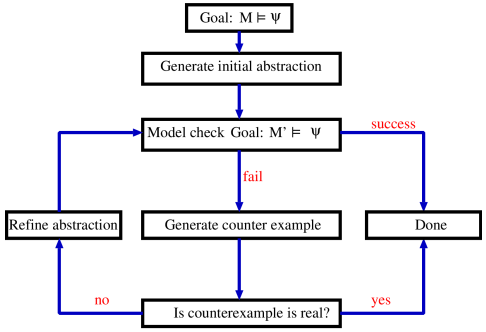
$H_{AP} : \{r, y, g\} \to \{r, yg\} \to \mathbb{B}$

$H_{AP}\ r\ r\ \wedge$
$H_{AP}\ y\ yg\ \wedge$
$H_{AP}\ g\ yg$

- $\overline{M} \models$ **AG AF** $\neg r$ hence $M \models$ **AG AF** $\neg r$
- but $\neg(\overline{M} \models$ **AG AF** $r)$ doesn't entail $\neg(M \models$ **AG AF** $r)$
  - $[\![$**AG AF** $r]\!]_{\overline{M}}(STOP)$ is false
    (consider $\overline{M}$-path π' where $\pi' = STOP.GO.GO.GO.\cdots$)
  - $[\![$**AG AF** $r]\!]_{M}(RED)$ is true
    (abstract path π' doesn't correspond to a real path in M)

CEGAR

- Counter Example Guided Abstraction Refinement



- Lots of details to fill out (several different solutions)
  - how to generate abstraction
  - how to check counterexamples
  - how to refine abstractions
- Microsoft SLAM driver verifier is a CEGAR system

Temporal Logic and Model Checking – Summary

- Various property languages: LTL, CTL, PSL (Prior, Pnueli)
- Models abstracted from hardware or software designs
- Model checking checks $M \models \psi$ (Clarke et al.)
- Symbolic model checking uses BDDs (McMillan)
- Avoid state explosion via simulation and abstraction
- CEGAR refines abstractions by analysing counterexamples
- Triumph of application of computer science theory
  - two Turing awards, McMillan gets 2010 CAV award
  - widespread applications in industry

THE END