

COMP6463: Temporal Logic and Model Checking

3. Explicit State Model Checking

Michael Norrish

Canberra Research Lab., NICTA

Semester 1, 2009



NICTA

Outline

- 1 Model Checking Generally
- 2 An Unusual Example: JPF
- 3 Explicit State Model Checking for CTL
- 4 Model Checking Fair CTL

Model Checking—What, How and Why

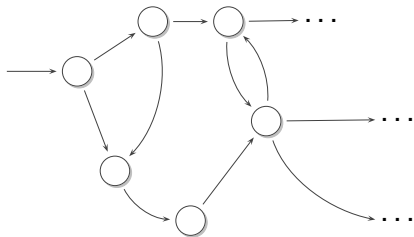
We're doing something “logic-y”, so we have a turnstile. . .



Model Checking—What, How and Why

First, we take a **model**:

(Call it \mathfrak{M})

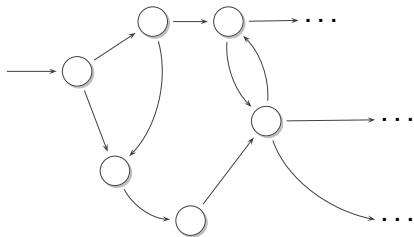


\models

Model Checking—What, How and Why

First, we take a **model**:

(Call it \mathfrak{M})



$\mathfrak{M} \models$

Model Checking—What, How and Why

Then we want a **description/property**.

For example:

the machine acknowledges every request it receives

(Call the property ϕ .)

$$\mathcal{M} \models$$

Model Checking—What, How and Why

Then we want a **description/property**.

For example:

the machine acknowledges every request it receives

(Call the property ϕ .)

$$\mathfrak{M} \models \phi$$

Model Checking—What, How and Why

The model checking question:

Does \mathfrak{M} exhibit/satisfy ϕ ?

$$\mathfrak{M} \models \phi$$

Model Checking vs Entailment

The

$$\Gamma \models \phi$$

notation is also used to mean

Entailment: If all of the formulas in the set Γ are true, then so too is the single formula ϕ .

Clearly, the two are related, but they're not the same.

Models

Models that are checked are typically things with evolving behaviour.

- (Java) Programs
- Hardware Designs
- Communication Protocols

Properties

A typical model checking **property** does not give a complete statement of functional correctness.

Safety properties:

- bad stuff doesn't happen (for possible values of "bad stuff")

Liveness properties:

- good stuff keeps happening

Collection of all checked properties may approximate a complete functional characterisation.

Java Pathfinder

*the (swiss army knife | vulcan mind probe |
spiral of death | [what do you want it to be today?](#))
of Java program verification*

<http://javapathfinder.sourceforge.net>

This is an [explicit state](#), [software](#) model checker for Java programs.

Developed at NASA;

has “helped detect defects in real spacecraft”.

JPF Model Checking

JPF is not really a “traditional” model checker. . .

The typical **property** checked is

- absence of deadlocks
- unhandled exceptions

Properties to be checked can be extended.

The **model** is a Java program.

JPF In Action

The exercise is to look at all execution paths of the Java program.

- doesn't handle I/O very well
- looks at all possible interleavings of a multi-threaded program

Suitable for verifying smallish (10kLOC) concurrent programs

How JPF Works

The JPF is essentially a **custom JVM**:

- it executes Java byte codes
 - (errors it encounters will be genuine)
- states are recorded as they are encountered
 - a traversal of “all” paths is possible
- Traversal is to a specified trace depth
 - 18 threads to a depth of > 1000 = an afternoon's work

JPF is a Bug-finder

Formal methods typically give guarantees of the absence of errors

- this is supposed to be the difference between proof and testing

So, JPF isn't doing proof.

- It can't explore the whole state-space (which is probably infinite).
- Big deal! **A bug is a bug**
- Bugs come with traces leading to their reproduction

Outline

- 1 Model Checking Generally
- 2 An Unusual Example: JPF
- 3 Explicit State Model Checking for CTL**
- 4 Model Checking Fair CTL

Model Checking CTL Properties

Preconditions

- have a system (W, R, V)
- the system to be checked has a finite number of states;
- we begin with the graph of the system;
- the property to be checked ϕ is expressed in CTL

Problem:

- Determine the subset of states in W that satisfy ϕ
- The system satisfies the property if its initial states are a subset of the satisfying set

The Basic Idea: Labelling States with Propositions

Have access to the complete graph.

Approach is to **label** each state with those formulas known to hold there.

When we're done, states satisfying ϕ will be exactly those that have been labelled with ϕ .

(Implicit) First Step

- Label states with the propositional variables that hold there.

Use as Few Operators as Possible

From last time, we know that CTL can be expressed with just

$p, q, r, \dots, \vee, \neg, EX, EG$ and $E[_ U _]$

Procedure is a bottom-up recursion over structure of formula:

$label(\neg\phi)$ Call $label(\phi)$. Label every state not labelled with ϕ with $\neg\phi$

$label(\phi_1 \vee \phi_2)$ Call $label(\phi_1)$ and $label(\phi_2)$. Every state labelled with either ϕ_1 or ϕ_2 must now be labelled with $\phi_1 \vee \phi_2$

$label(EX \phi)$ Call $label(\phi)$. Every w that can reach a state labelled with ϕ now gets labelled with $EX \phi$

Labelling the Hard Cases—Until

Handling $\text{label}(E[\phi_1 \cup \phi_2])$:

- 1 Call $\text{label}(\phi_1)$ and $\text{label}(\phi_2)$
- 2 Everything labelled with ϕ_2 gets an $E[\phi_1 \cup \phi_2]$ label
- 3 Initialise T to be the set of worlds labelled with $E[\phi_1 \cup \phi_2]$
- 4 While $T \neq \emptyset$, remove a t from T . Then, for each u that can reach t :
 - if u is labelled with ϕ_1 and not already labelled with $E[\phi_1 \cup \phi_2]$, then label it with $E[\phi_1 \cup \phi_2]$ and add it to T

Interlude: Strongly Connected Components

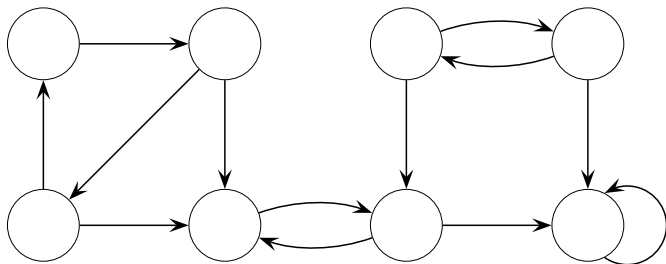
A **strongly connected component** of a graph (V, E) is a set $U \subseteq V$, such that for all $u, v \in U$, $u \rightarrow^* v$ and $v \rightarrow^* u$.

Alternatively, the **strongly connected components** (SCCs) of a graph are the equivalence classes of vertices under the “are mutually reachable” relation.

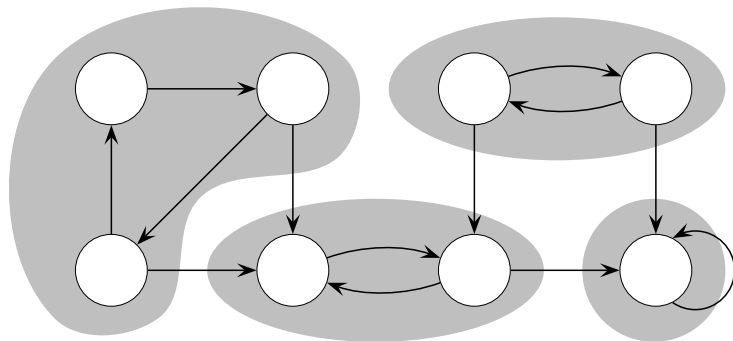
Call a singleton node that can not reach itself a **trivial** component.

(See algorithms books like Cormen, Leiserson and Rivest.)

Example: Strongly Connected Components



Example: Strongly Connected Components



Bottom-right node is a non-trivial SCC.

Calculating Strongly Connected Components

Finding a graph's strongly connected components can be done by making **two** depth-first traversals (one after the other).

- (See texts for description of this cool algorithm. Due to Tarjan.)

A depth-first traversal takes linear time in the size of the graph.

So determining a graph's SCCs is also linear time.

Labelling the Hard Cases—EG

Handling `label(EG ϕ)`:

- 1 Call `label(ϕ)`
- 2 Create a sub-graph (W', R') consisting of just those nodes labelled with ϕ (and their incident edges)
- 3 Calculate the SCCs of (W', R')
- 4 Add `EG ϕ` as a label to all nodes in a non-trivial SCC.
(Call this set of nodes T)
- 5 While $T \neq \emptyset$, remove a t from T . Then, for each $u \in W'$ that can reach t :
 - if u is not already labelled with `EG ϕ` , label it with `EG ϕ` and add u to T .

EG Labelling—Intuition

Let u be a node labelled with ϕ in a non-trivial SCC of the ϕ -graph.

Then there exists an infinite path looping through u that has ϕ true at each point.

Then, scan backwards in ϕ -graph from the nodes in a known loop.

- If a node t can reach node u (which is in a loop), then t also has an infinite ϕ -path ahead of it

Thus, if the algorithm labels a node with EG ϕ , that node really does satisfy the formula.

EG Labelling—Intuition

Converse Question: does the algorithm find all nodes that should be labelled with EG ϕ ?

If a world w has an infinite ϕ -path emanating from it:

- There must come a point at which the path loops (finite model!)
- A loop must be part of a strongly connected component
- The non-loop prefix will be found as the algorithm follows links backwards from the SCCs.

Complexity of CTL Model Checking

Each call of `label` on a sub-formula may take time linear in the model

- (linear in number of states + number of edges)

In a formula ϕ of size n , there are n sub-formulas, so the algorithm will run in time

$$O(|\phi| \times (|W| + |R|))$$

For a fixed model, checking formulas is linear in the size of the formula.

(This is *fantastically good*.)

One Last Observation

The algorithm requires that all states be available at once.

- Can't calculate SCCs for EG checking otherwise

This means that we can't do “on-the-fly” model-checking, where the graph is explored incrementally.

- Contrast JPF, as well as Spin checker (to come)

Outline

- 1 Model Checking Generally
- 2 An Unusual Example: JPF
- 3 Explicit State Model Checking for CTL
- 4 Model Checking Fair CTL**

Fairness

A path π is **fair** with respect to a fairness constraint F if the constraint is satisfied infinitely often on the path.

Fairness is used to constrain models that might otherwise be unrealistic.

For example, a system may be the combination of n components that can evolve independently.

Models that Need Fairness

A system may be the combination of n components that can evolve independently.

The obvious description is:

$$R(v_0, \dots, v_n)(v'_0, \dots, v'_n) \equiv \\ \bigvee_i \left(R_i(v_i, v'_i) \wedge \bigwedge_{j \neq i} (v'_j = v_j) \right)$$

I.e., any v_i component may take a step at each stage, and all the other components stay the same.

Models that Need Fairness

A model with a description like

$$R(v_0, \dots, v_n)(v'_0, \dots, v'_n) \equiv \\ \bigvee_i \left(R_i(v_i, v'_i) \wedge \bigwedge_{j \neq i} (v'_j = v_j) \right)$$

is logically permitted to have only the v_0 component ever do anything.

Fairness can prevent this: require variables p_i that are only true in each component to be true infinitely often.

Fair Strongly Connected Components

A strongly connected component is **fair** with respect to F if it includes a world w that satisfies F .

So, in a **fair** strongly connected component, there is necessarily a fair path with respect to F .

- I.e., it is possible to repeatedly hit the satisfying w on an infinite path.
- (It is possible to have other paths in the component not hit the fair world w .)

Checking Fair EG

When labelling the model with EG ϕ :

- (recursively) find the states satisfying ϕ ;
- consider just the **fair** SCCs;
- extrapolate backwards from SCCs finding finite prefixes of paths

This procedure can also be used to find those states that are on fair paths:

- Check the state for EG T

Fair States and States on Fair Paths

Fair States: states that satisfy the fairness constraint F .

States on Fair Paths: states that are on paths that (have worlds on them that) satisfy the fairness constraint infinitely often.

(The two sets have a non-empty intersection, but neither is necessarily a subset of the other.)

Cache States of Fair Paths

We need to know which states are on fair paths.

Compute this information once, and record it in each state by using a fresh propositional variable *fair*.

This will allow us to use standard CTL algorithms for the rest of cases.

Fair Checking for EX

To check $EX \phi$:

- Check ϕ
- Check $EX (\phi \wedge \textit{fair})$
 - i.e., label world w with $EX \phi$ if it has a successor with label ϕ , and with propositional variable *fair*.

Fair Checking for $E[\phi_1 \cup \phi_2]$

To check $E[\phi_1 \cup \phi_2]$:

- Check ϕ_1 and ϕ_2 (using fair checking)
- Check $E[\phi_1 \cup (\phi_2 \wedge \textit{fair})]$ (using rest of standard algorithm for until)

Exercise: Argue that this is correct!

Summary

- The **very idea** of model checking: determining whether or not a system satisfies certain properties
- Explicit state model checking with **Java Pathfinder**, a powerful bug-finding system
- Checking **CTL properties** of explicit models
- Doing so **fairly**
- (One exercise!)