Program Verification: An overview of the series

Prakash Panangaden
School of Computer Science
McGill University

The lecturers

The lecturers

The lecturers

- Joel Ouaknine; University of Oxford

The lecturers

- Joel Ouaknine; University of Oxford
- James Worrell; University of Oxford

The lecturers

- Joel Ouaknine; University of Oxford
- James Worrell; University of Oxford
- Amy Felty; University of Ottawa

The lecturers

- Joel Ouaknine; University of Oxford
- James Worrell; University of Oxford
- Amy Felty; University of Ottawa
- Stephen Brookes; Carnegie-Mellon University

The topics

The topics

- Real-time systems (Ouaknine)

The topics

* Real-time systems (Ouaknine)
* Probabilistic systems (Worrell)

The topics

* Real-time systems (Ouaknine)
* Probabilistic systems (Worrell)
* Theorem proving techniques (Felty)

The topics

- Real-time systems (Ouaknine)
- Probabilistic systems (Worrell)
- Theorem proving techniques (Felty)
- Concurrent systems (Brookes)

Early history: formalisms

Early history: formalisms

- McCarthy in the early 1960s introduced a *Mathematical Theory of Computation*

Early history: formalisms

- McCarthy in the early 1960s introduced a *Mathematical Theory of Computation*
- Floyd in the mid 1960s introduced methods for reasoning on flowcharts: inductive assertions

Early history: formalisms

* McCarthy in the early 1960s introduced a *Mathematical Theory of Computation*
* Floyd in the mid 1960s introduced methods for reasoning on flowcharts: inductive assertions
* Scott, deBakker 1969: fixed-point induction

Early history: formalisms

- McCarthy in the early 1960s introduced a *Mathematical Theory of Computation*
- Floyd in the mid 1960s introduced methods for reasoning on flowcharts: inductive assertions
- Scott, deBakker 1969: fixed-point induction
- Hoare 1969: axiomatic semantics

Early history: formalisms

- McCarthy in the early 1960s introduced a *Mathematical Theory of Computation*
- Floyd in the mid 1960s introduced methods for reasoning on flowcharts: inductive assertions
- Scott, deBakker 1969: fixed-point induction
- Hoare 1969: axiomatic semantics
- Early 1970s: predicate transformers (Dijkstra)

The pace picks up

The pace picks up

- Early 1970s: LCF/ML (Milner), Boyer-Moore

The pace picks up

- Early 1970s: LCF/ML (Milner), Boyer-Moore
- Dynamic logics: Pratt, Kozen, Parikh, Harel, Constable, Clark,...

The pace picks up

- Early 1970s: LCF/ML (Milner), Boyer-Moore
- Dynamic logics: Pratt, Kozen, Parikh, Harel, Constable, Clark,...
- (linear) Temporal logic (Pnueli), CTL

The pace picks up

- Early 1970s: LCF/ML (Milner), Boyer-Moore
- Dynamic logics: Pratt, Kozen, Parikh, Harel, Constable, Clark,...
- (linear) Temporal logic (Pnueli), CTL
- Abstract interpretation (1976) Cousot and Cousot

The pace picks up

* Early 1970s: LCF/ML (Milner), Boyer-Moore
* Dynamic logics: Pratt, Kozen, Parikh, Harel, Constable, Clark,...
* (linear) Temporal logic (Pnueli), CTL
* Abstract interpretation (1976) Cousot and Cousot
* Model checking (Clarke, Emerson, Sifakis)

Disasters and Opportunities

Disasters and Opportunities

- The Ariane-5 disaster

Disasters and Opportunities

- The Ariane-5 disaster
- The Pentium bug

Disasters and Opportunities

* The Ariane-5 disaster
* The Pentium bug
* The Bang & Olufsen Audio/Video protocol

Disasters and Opportunities

- The Ariane-5 disaster
- The Pentium bug
- The Bang & Olufsen Audio/Video protocol
- The attack on the Needham-Schroder protocol

Some areas of current research

Some areas of current research

- Abstraction techniques

Some areas of current research

- Abstraction techniques
- Probabilistic verification

Some areas of current research

* Abstraction techniques
* Probabilistic verification
* Real-time systems

Some areas of current research

- Abstraction techniques
- Probabilistic verification
- Real-time systems
- Concurrency

Some areas of current research

- Abstraction techniques
- Probabilistic verification
- Real-time systems
- Concurrency
- Security

The Main Point

- Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning: Hoare 1969.

The Main Point

- Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning: Hoare 1969.

The Main Point

- Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning: Hoare 1969.

The Main Point

- Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning: Hoare 1969.

Semantics and Axioms

Semantics and Axioms

- A precise specification of the execution effect of a program.

Semantics and Axioms

- A precise specification of the execution effect of a program.
- Ideally it should be compositional.

Semantics and Axioms

- A precise specification of the execution effect of a program.
- Ideally it should be compositional.
- One should be able to extract the **relevant** aspects of the program through axioms that capture the semantics.

Hoare Logic

Hoare Logic

- Assertions describe properties of the state.

Hoare Logic

- Assertions describe properties of the state.
- {P} S {Q}: If P holds before execution of S then Q will hold after S terminates, if S does indeed terminate.

Hoare Logic

- Assertions describe properties of the state.
- {P} S {Q}: If P holds before execution of S then Q will hold after S terminates, if S does indeed terminate.
- Compositionality: From {P} S {R} and {R} S' {Q} deduce {P} S;S' {Q}.

Dynamic Logic

Dynamic Logic

* A modal logic with programs and formulas defined by mutual induction.

Dynamic Logic

- A modal logic with programs and formulas defined by mutual induction.
- Every program defines a modality.

Dynamic Logic

- A modal logic with programs and formulas defined by mutual induction.
- Every program defines a modality.
- Challenging from the point of view of basic theory: the canonical model construction does not work.

Recursion

Recursion

- How to make sense of recursion compositionally?

Recursion

* How to make sense of recursion compositionally?
* Fixed-point theory (Kleene).

Recursion

* How to make sense of recursion compositionally?
* Fixed-point theory (Kleene).
* rec f. F[f] is the solution of f = F[f].

Recursion

- How to make sense of recursion compositionally?
- Fixed-point theory (Kleene).
- rec f. F[f] is the solution of f = F[f].
- Fixed-point induction for programs: Scott and deBakker, Park.

Denotational Semantics

Denotational Semantics

Data types are *domains*: dcpos with $\perp$.

Denotational Semantics

Data types are *domains*: dcpos with $\perp$.

Programs define functions between data types:
these functions are (Scott) continuous and monotone.

Denotational Semantics

Data types are *domains*: dcpos with $\perp$.

Programs define functions between data types:
these functions are (Scott) continuous and monotone.

The function spaces are themselves data types.

Denotational Semantics

Data types are *domains*: dcpos with $\perp$.

Programs define functions between data types:
these functions are (Scott) continuous and monotone.

The function spaces are themselves data types.

Continuous functions from D to itself have *least fixed points.*

Denotational Semantics

Data types are *domains*: dcpos with $\perp$.

Programs define functions between data types:
these functions are (Scott) continuous and monotone.

The function spaces are themselves data types.

Continuous functions from D to itself have *least fixed points.*

The meaning of a recursively defined function from D to D is given by the least fixed point of a functional from $D \to D$ to $D \to D$.

Fixed-point Induction

Fixed-point Induction

Let D be a dcpo. A subset $S \subseteq D$ is called *chain-closed* if for all chains

$$d_0 \leq d_1 \leq d_2 \leq \ldots$$

in D, we have

$$\forall n. d_n \in S \Rightarrow \bigvee_n d_n \in S.$$

Fixed-point Induction

Let D be a dcpo. A subset $S \subseteq D$ is called *chain-closed* if for all chains

$$d_0 \leq d_1 \leq d_2 \leq \dots$$

in D, we have

$$\forall n. d_n \in S \Rightarrow \bigvee_n d_n \in S.$$

If S contains $\perp$ and is chain closed, we call it *admissible*.

Fixed-point Induction

Let D be a dcpo. A subset $S \subseteq D$ is called *chain-closed* if for all chains

$$d_0 \leq d_1 \leq d_2 \leq \dots$$

in D, we have

$$\forall n. d_n \in S \Rightarrow \bigvee_n d_n \in S.$$

If S contains $\perp$ and is chain closed, we call it *admissible*.

Similarly a property Φ, may be admissible.

Fixed-point Induction

Let D be a dcpo. A subset $S \subseteq D$ is called *chain-closed* if for all chains

$$d_0 \leq d_1 \leq d_2 \leq \dots$$

in D, we have

$$\forall n. d_n \in S \Rightarrow \bigvee_n d_n \in S.$$

If S contains $\perp$ and is chain closed, we call it *admissible*.

Similarly a property Φ, may be admissible.

If $f : D \rightarrow D$ is continuous, Φ is admissible and

$$\forall d \in S, f(d) \in S$$

then, $fix(f) \in S$.

Fixed-point Induction

Let D be a dcpo. A subset $S \subseteq D$ is called *chain-closed* if for all chains

$$d_0 \leq d_1 \leq d_2 \leq \dots$$

in D, we have

$$\forall n. d_n \in S \Rightarrow \bigvee_n d_n \in S.$$

If S contains $\perp$ and is chain closed, we call it *admissible*.

Similarly a property Φ, may be admissible.

If $f : D \to D$ is continuous, Φ is admissible and

$$\forall d \in S, f(d) \in S$$

then, $fix(f) \in S$.

With this many properties of recursively defined functions can be proved.

Abstract Interpretation

How can we use denotational semantics to prove properties without computing the detailed behaviour of the program?

Abstract Interpretation

How can we use denotational semantics to prove properties without computing the detailed behaviour of the program?

Use abstracted data types!

Abstract Interpretation

How can we use denotational semantics to prove properties without computing the detailed behaviour of the program?

Use abstracted data types!

$$\begin{tikzcd}
D \arrow[r, "f"] \arrow[d, bend right, "\alpha_1"'] & E \arrow[d, bend left, "\alpha_2"] \\
A \arrow[r, "g"] \arrow[u, "\gamma_1"'] & B \arrow[u, "\gamma_2"]
\end{tikzcd}$$

Model Checking

Model Checking

* Describe the system (program) as a transition system of some kind.

Model Checking

- Describe the system (program) as a transition system of some kind.
- Give the specification in a suitable (dynamic) logic.

Model Checking

- Describe the system (program) as a transition system of some kind.
- Give the specification in a suitable (dynamic) logic.
- Show **automatically** that the system is a **model** of the specification.

Theorem proving

Theorem proving

- Describe the (relevant parts of or behaviour of) the system using formulas. [Beh]

Theorem proving

- Describe the (relevant parts of or behaviour of) the system using formulas. [Beh]
- Define the specification as another formula. [Spec]

Theorem proving

- Describe the (relevant parts of or behaviour of) the system using formulas. [Beh]
- Define the specification as another formula. [Spec]
- Prove, using semi-automatic tools if possible, that Beh implies Spec.

Which is better?

Which is better?

- Theorem proving is good when one doesn't have a complete picture of the model and one can capture some of their properties using axioms.

Which is better?

* Theorem proving is good when one doesn't have a complete picture of the model and one can capture some of their properties using axioms.
* Model checking allows a different formalism for describing the model and writing the specification. This allows one to use a rather restricted language for the specifications which has a better chance of being decidable.

Peace, flowers and love

Peace, flowers and love

- Of course, the two approaches should co-exist.

Peace, flowers and love

- Of course, the two approaches should co-exist.
- Theorem proving can settle properties that would require induction proofs and is much more powerful.

Peace, flowers and love

- Of course, the two approaches should co-exist.
- Theorem proving can settle properties that would require induction proofs and is much more powerful.
- Model checking can be a powerful tactic within a theorem proving environment.

Peace, flowers and love

- Of course, the two approaches should co-exist.
- Theorem proving can settle properties that would require induction proofs and is much more powerful.
- Model checking can be a powerful tactic within a theorem proving environment.
- Abstraction is a vital tool in both cases.

Model Checking

Model Checking

The system is a transition system

S: States

P: Propositions

$\rightarrow \subset S \times S$: Transition relation

$L : S \rightarrow 2^P$: Labelling function.

Model Checking

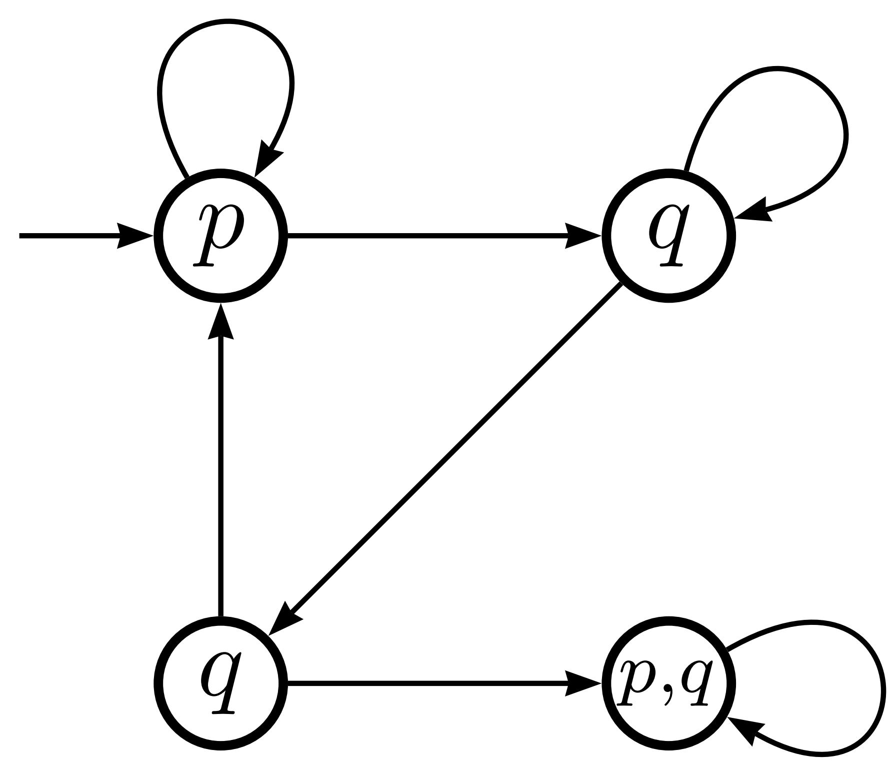
The system is a transition system

S: States

P: Propositions

$\rightarrow \subset S \times S$: Transition relation

$L : S \rightarrow 2^P$: Labelling function.



The Logic

$$\begin{array}{lll} \bigcirc & : & \text{Next} \\ \Diamond & : & \text{Eventually} \\ \Box & : & \text{Always} \\ \bigcup & : & \text{Until} \end{array}$$

The Logic

Usually temporal logic: Linear Temporal Logic (LTL) or Computation Tree Logic (CTL).

$\bigcirc$	:	Next
$\Diamond$	:	Eventually
$\Box$	:	Always
$\bigcup$	:	Until

The Logic

Usually temporal logic: Linear Temporal Logic (LTL) or Computation Tree Logic (CTL).

For transition systems of the type shown
CTL is more natural.

$$
\begin{array}{lcl}
\bigcirc & : & \text{Next} \\
\Diamond & : & \text{Eventually} \\
\Box & : & \text{Always} \\
\bigcup & : & \text{Until}
\end{array}
$$

The Logic

Usually temporal logic: Linear Temporal Logic (LTL) or Computation Tree Logic (CTL).

For transition systems of the type shown CTL is more natural.

State formulas:

$\phi ::== \text{true}|p|\phi_1 \wedge \phi_2|\neg\phi|\exists\psi|\forall\psi$

$\bigcirc$	:	Next
$\Diamond$	:	Eventually
$\Box$	:	Always
$\bigcup$	:	Until

The Logic

Usually temporal logic: Linear Temporal Logic (LTL) or Computation Tree Logic (CTL).

For transition systems of the type shown CTL is more natural.

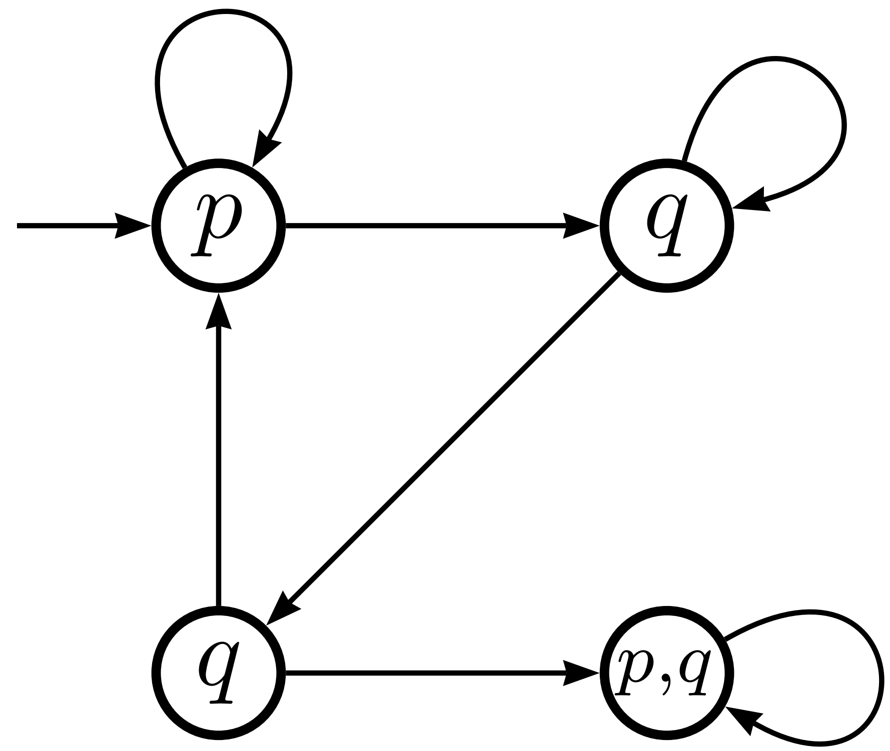
State formulas:

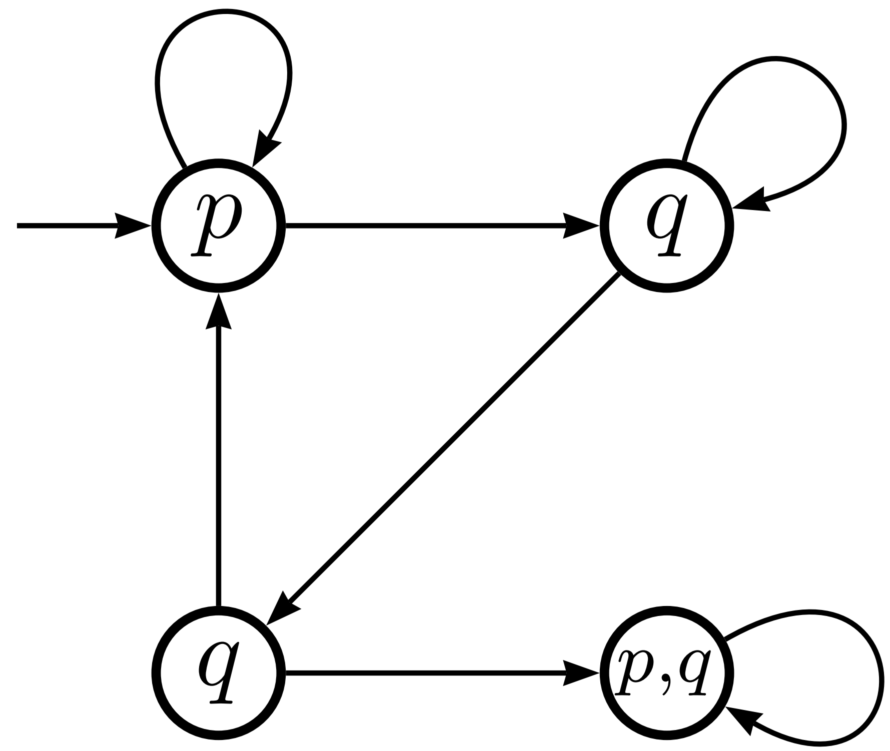
$\phi ::== \text{true}|p|\phi_1 \wedge \phi_2|\neg\phi|\exists\psi|\forall\psi$

Path formulas:

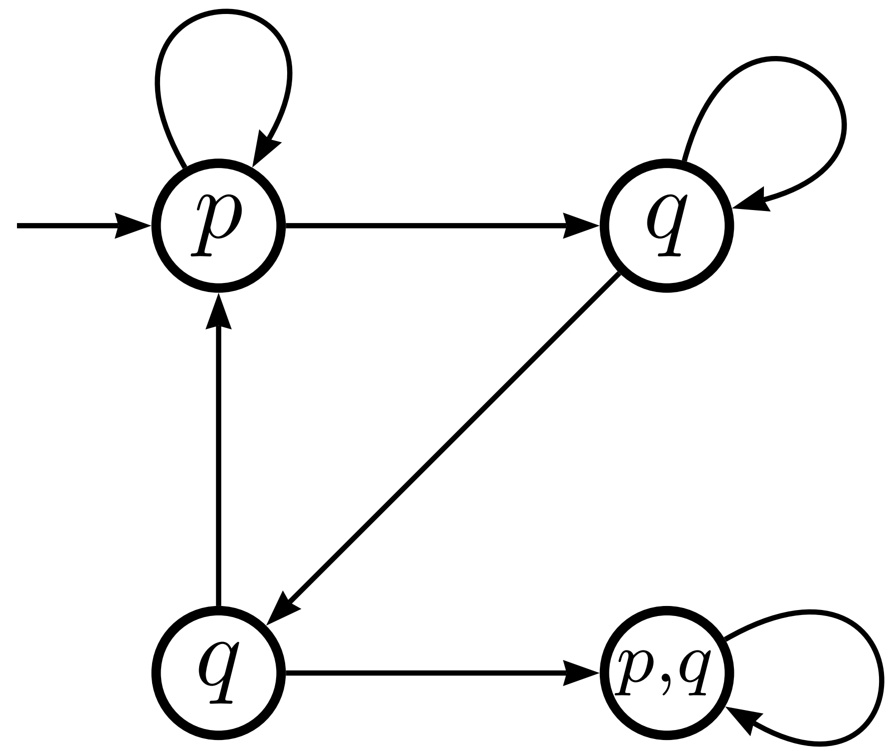
$\psi ::== \bigcirc\phi||\Diamond\phi|\Box\phi|\phi_1 \bigcup \phi_2$

$$\begin{array}{lcl} \bigcirc & : & \text{Next} \\ \Diamond & : & \text{Eventually} \\ \Box & : & \text{Always} \\ \bigcup & : & \text{Until} \end{array}$$


p
q
q
p,q

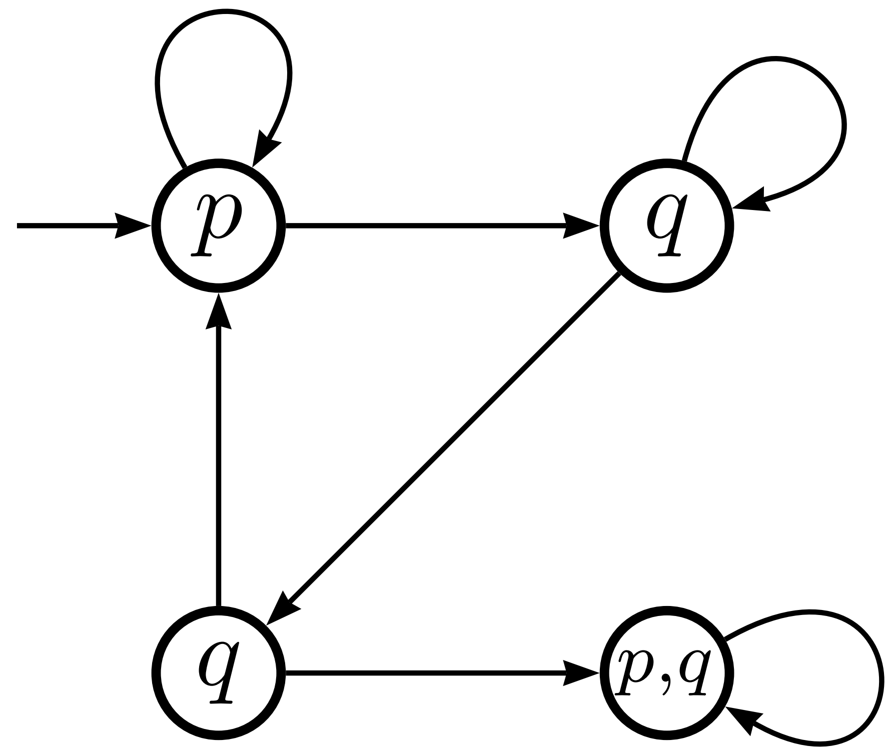


start state $\not\models \forall\Diamond q$



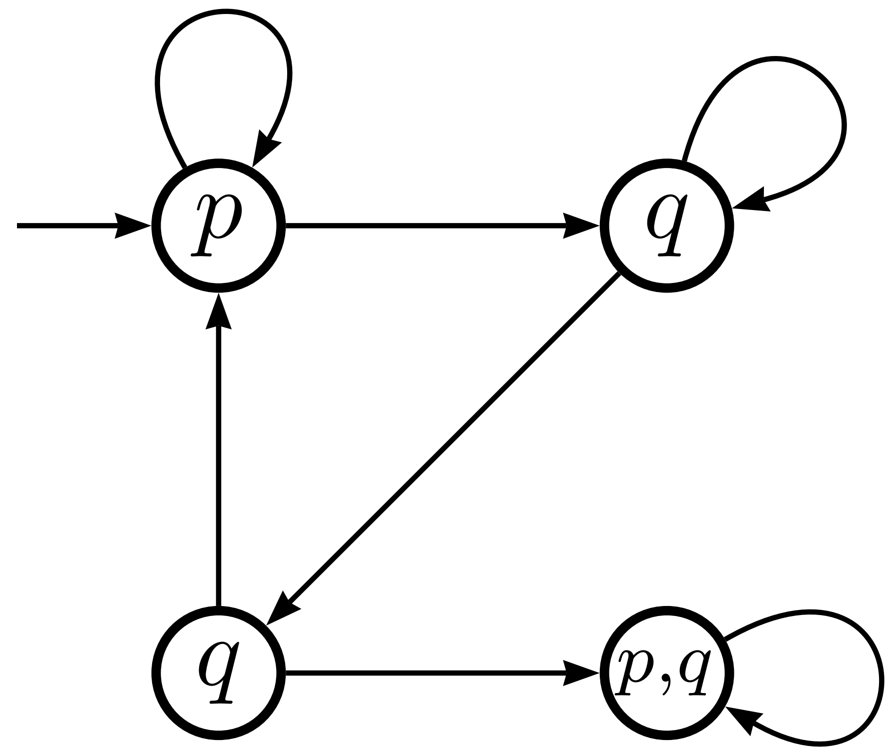
start state $\not\models \forall\Diamond q$

start state $\not\models \forall\Box\Diamond p$



start state $\not\models \forall\Diamond q$

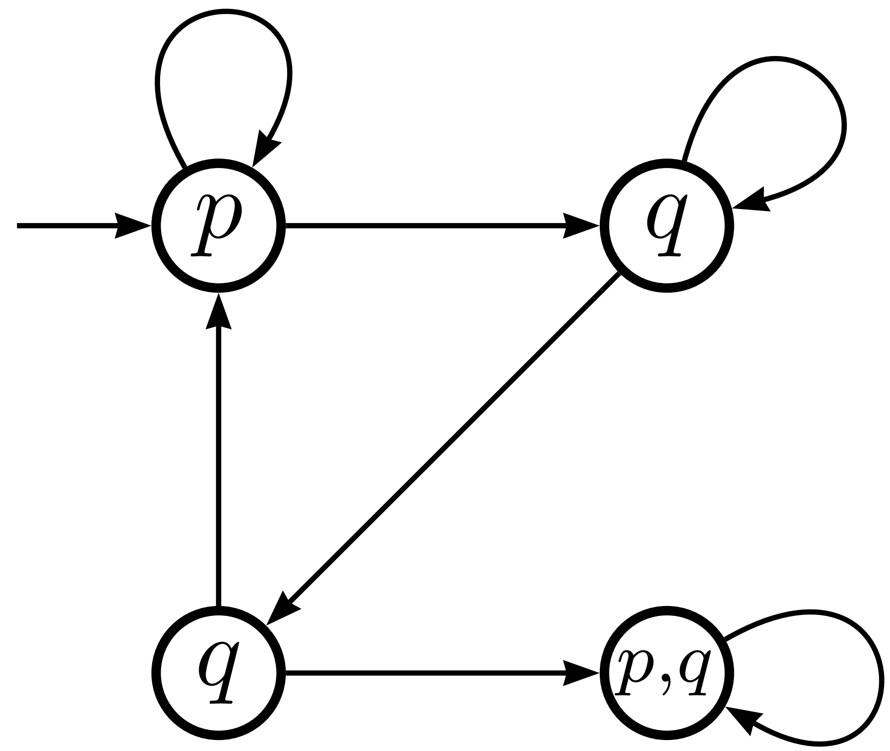
start state $\not\models \forall\Box\Diamond p$ This is LTL not CTL



start state $\not\models \forall\Diamond q$

start state $\not\models \forall\Box\Diamond p$ This is LTL not CTL

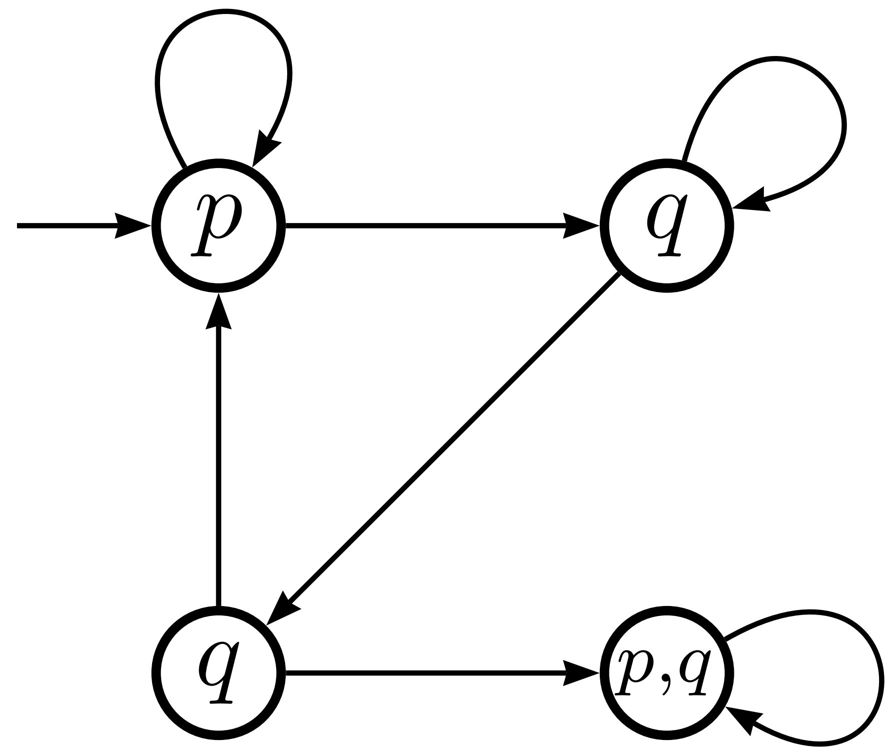
start state $\models \exists\Diamond\Box q$



start state $\not\models \forall\Diamond q$

start state $\not\models \forall\Box\Diamond p$ This is LTL not CTL

start state $\models \exists\Diamond\Box q$ This is CTL* not LTL

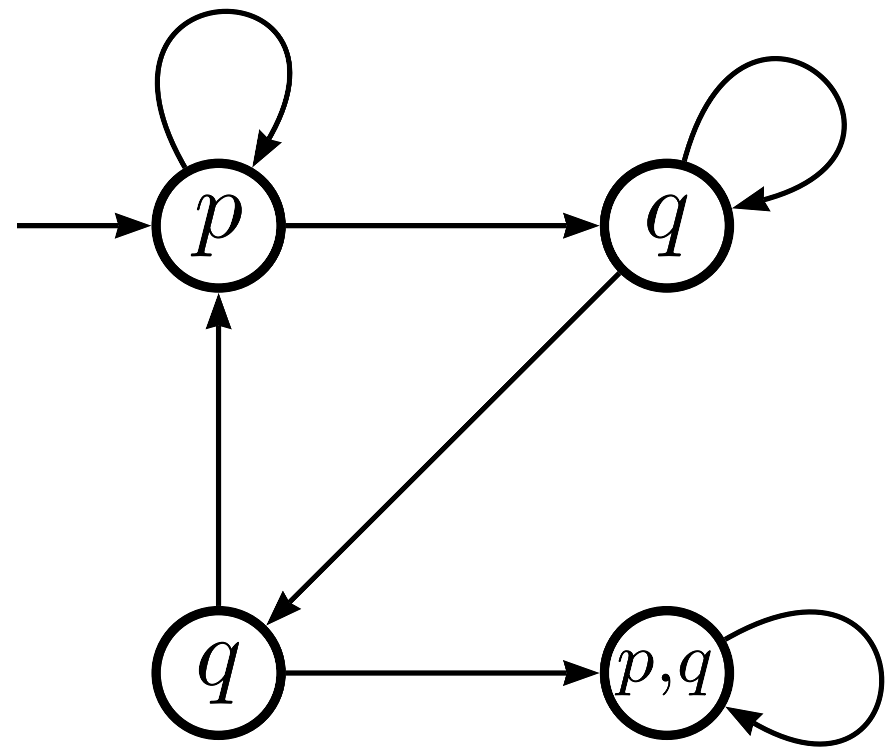


start state $\not\models \forall\Diamond q$

start state $\not\models \forall\Box\Diamond p$ This is LTL not CTL

start state $\models \exists\Diamond\Box q$ This is CTL* not LTL

start state $\models \exists$ "every second state satisfies q."



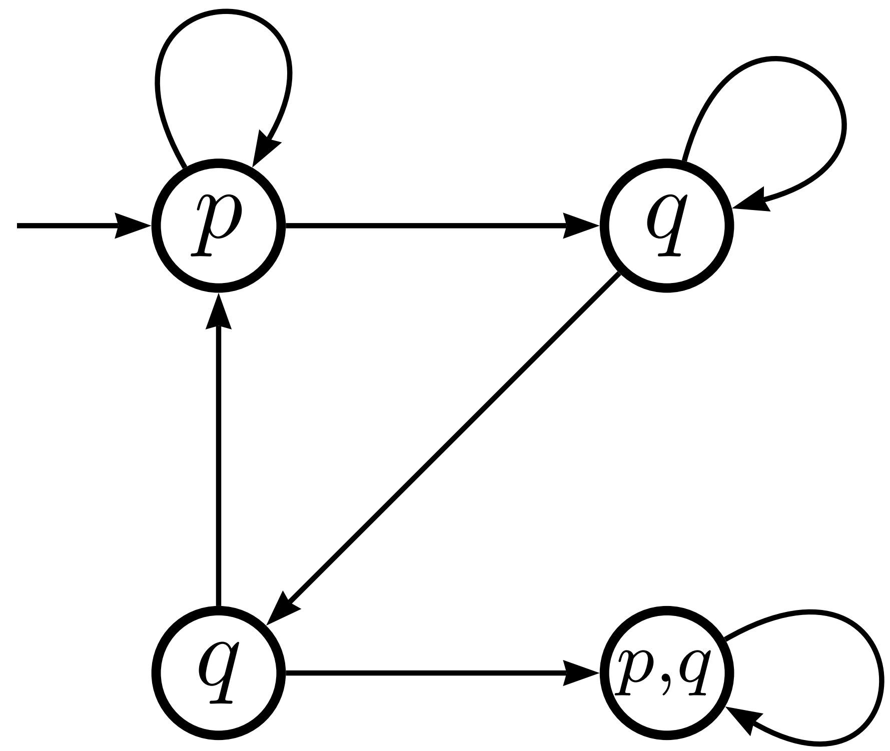
start state $\not\models \forall\Diamond q$

start state $\not\models \forall\Box\Diamond p$ This is LTL not CTL

start state $\models \exists\Diamond\Box q$ This is CTL* not LTL

start state $\models \exists$ "every second state satisfies q."

But "every second state satisfies q" cannot be expressed with these temporal formulas.



start state $\not\models \forall\Diamond q$

start state $\not\models \forall\Box\Diamond p$ This is LTL not CTL

start state $\models \exists\Diamond\Box q$ This is CTL* not LTL

start state $\models \exists$ "every second state satisfies q."

But "every second state satisfies q" cannot be expressed with these temporal formulas.

It can be expressed with *fixed-point* operators in the logic.

Semantics of the Logic

$s \models p$	iff $p \in L(s)$
$s \models \phi_1 \wedge \phi_2$	iff $s \models \phi_1$ and $s \models \phi_2$
$s \models \forall \psi$	iff $\forall$ paths $\pi = s s_1 s_2 \ldots$, $\pi \models \psi$
$s \models \exists \psi$	iff $\exists$ a path $\pi = s s_1 s_2 \ldots$, $\pi \models \psi$

A *path* is a sequence of states: $\pi = s_0 s_1 s_2 \ldots$

$\pi \models \bigcirc \phi$	iff $s_1 \models \phi$
$\pi \models \Diamond \phi$	iff $\exists j$ such that $s_j \models \phi$
$\pi \models \Box \phi$	iff $\forall j \;\; s_j \models \phi$
$\pi \models \phi_1 \bigcup \phi_2$	iff $\exists j$ such that $s_j \models \phi_2$ and $\forall i < j \; s_i \models \phi_1$

The Model-Checking Algorithm

The Model-Checking Algorithm

$$Sat(\phi) = \{s | s \models \phi\}$$

The Model-Checking Algorithm

$$Sat(\phi) = \{s | s \models \phi\}$$

$$Post(s) = \{s' | s \rightarrow s'\}, Pre(s) = \{s' | s' \rightarrow s\}$$

The Model-Checking Algorithm

$$Sat(\phi) = \{s | s \models \phi\}$$

$$Post(s) = \{s' | s \to s'\}, Pre(s) = \{s' | s' \to s\}$$

Input: TS with states S, CTL state formula Φ
Output: $T(\subset S) = \{s | s \models \Phi\} = Sat(\Phi)$.

The Model-Checking Algorithm

$$Sat(\phi) = \{s | s \models \phi\}$$

$$Post(s) = \{s' | s \rightarrow s'\}, Pre(s) = \{s' | s' \rightarrow s\}$$

Input: TS with states S, CTL state formula Φ
Output: $T(\subset S) = \{s | s \models \Phi\} = Sat(\Phi)$.

$$\begin{aligned}
p: &\quad T = \{s | p \in L(s)\} \\
\phi_1 \wedge \phi_2: &\quad T = Sat(\phi_1) \bigcap Sat(\phi_2) \\
\neg\phi: &\quad T = S \setminus Sat(\phi) \\
\exists \bigcirc \phi: &\quad T = \{s | Post(s) \bigcap Sat(\phi) \neq \emptyset\} \\
\forall \bigcirc \phi: &\quad T = \{s | Post(s) \subseteq Sat(\phi)\}
\end{aligned}$$

Suppose the formula is $\phi = \exists(\phi_1 \bigcup \phi_2))$.
Note that $\phi = \phi_2 \vee \exists \bigcirc \phi$; a fixed-point formula!

Suppose the formula is $\phi = \exists(\phi_1 \bigcup \phi_2))$.
Note that $\phi = \phi_2 \vee \exists \bigcirc \phi$; a fixed-point formula!

Iterative algorithm to compute this (least) fixed point:

Suppose the formula is $\phi = \exists(\phi_1 \bigcup \phi_2))$.
Note that $\phi = \phi_2 \vee \exists \bigcirc \phi$; a fixed-point formula!

Iterative algorithm to compute this (least) fixed point:

$$
\begin{array}{l}
T := Sat(\phi_2) \\
\textbf{for all} \\
\quad s \in Sat(\phi_1) \setminus T \\
\qquad \textbf{do} \\
\qquad\quad \textbf{if } Post(s) \bigcap T \neq \emptyset \\
\qquad\quad \textbf{then } T := T \bigcup \{s\}.
\end{array}
$$

Similarly,

$$\exists\Box\phi = \phi \wedge \exists \bigcirc \exists\Box\phi,$$

so we have a *greatest* fixed point.

Similarly,

$$\exists\Box\phi = \phi \wedge \exists \bigcirc \exists\Box\phi,$$

so we have a *greatest* fixed point.

An iterative algorithm for computing the greatest fixed point.

Similarly,

$$\exists\Box\phi = \phi \wedge \exists \bigcirc \exists\Box\phi,$$

so we have a *greatest* fixed point.

An iterative algorithm for computing the greatest fixed point.

$T := Sat(\phi)$
repeat
  choose $s \in T$;
    if $Post(s) \bigcap T = \emptyset$
    then $T := T \setminus \{s\}$
until
  $\forall s \in T, Post(s) \bigcap T \neq \emptyset.$

For a transition system with n states and t transitions and a CTL formula ϕ of size k, the model-checking problem can be solved in time

$$O((n + t).k).$$

Further directions

Further directions

- Model checking with fairness assumptions

Further directions

* Model checking with fairness assumptions
* Finding counterexamples and witnesses

Further directions

- Model checking with fairness assumptions
- Finding counterexamples and witnesses
- Symbolic model checking: dealing with large systems by working with sets of states symbolically

Further directions

- Model checking with fairness assumptions
- Finding counterexamples and witnesses
- Symbolic model checking: dealing with large systems by working with sets of states symbolically
- Using BDDs to represent sets and set operations efficiently

LTL, CTL*, mu-calculus

LTL, CTL*, mu-calculus

- LTL uses a single outermost universal path quantifier.

LTL, CTL*, mu-calculus

- LTL uses a single outermost universal path quantifier.
- Very good for dealing with systems specified as sets of possible runs.

LTL, CTL*, mu-calculus

- LTL uses a single outermost universal path quantifier.
- Very good for dealing with systems specified as sets of possible runs.
- LTL and CTL have different expressive power: neither subsumes the other.

LTL, CTL*, mu-calculus

- LTL uses a single outermost universal path quantifier.
- Very good for dealing with systems specified as sets of possible runs.
- LTL and CTL have different expressive power: neither subsumes the other.
- Both are fragments of CTL*

LTL, CTL*, mu-calculus

- LTL uses a single outermost universal path quantifier.
- Very good for dealing with systems specified as sets of possible runs.
- LTL and CTL have different expressive power: neither subsumes the other.
- Both are fragments of CTL*
- mu-calculus, allows general fixed-point operators.

LTL model checking

LTL model checking

- Based on automata-theoretic techniques.

LTL model checking

- Based on automata-theoretic techniques.
- PSPACE hard.

LTL model checking

* Based on automata-theoretic techniques.
* PSPACE hard.
* So what? Still very useful!

LTL model checking

- Based on automata-theoretic techniques.
- PSPACE hard.
- So what? Still very useful!
- Handles fairness nicely.

LTL model checking

- Based on automata-theoretic techniques.
- PSPACE hard.
- So what? Still very useful!
- Handles fairness nicely.
- CTL* not significantly harder.

Extensions

* Timed automata
* Probabilistic transition systems

THE END