

C01 – Intro

Program Verification

FMI · Denisa Diaconescu · Spring 2022

Overview

Why formal verification?

Formal verification - overview

Program analysis

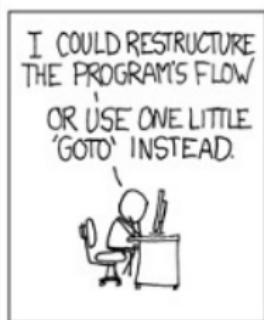
Formal semantics of programs

Why formal verification?

Computer scientists?

What is (or should be) the essential preoccupation of computer scientists?

The production of reliable software, its maintenance, and safe evolution year after year (up to 20 even 30 years).



Software bugs

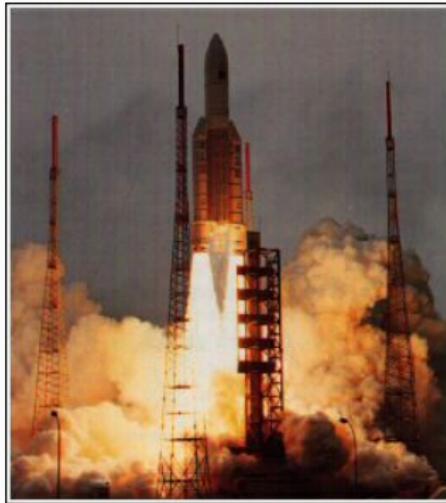
- Bugs are everywhere!
- Bugs can be very difficult to discover in huge software
- Bugs can have catastrophic consequences either very costly or inadmissible
- [Here](#) you can find a collection of "famous" bugs



The cost of software failure

- Patriot MIM-104 failure, 25 February 1991
 - death of 28 soldiers
 - An Iraqi Scud hit the Army barracks in Dhahran, Saudi Arabia. The Patriot defense system had failed to track and intercept the Scud.
 - R. Skeel. *Roundoff Error and the Patriot Missile*.
- Ariane 5 failure, 4 June 1996
 - cost estimated at more than 370 000 000\$
 - M. Dowson. *The Ariane 5 Software Failure*
- Toyota electronic throttle control system failure, 2005
 - at least 89 deaths
 - CBSNews. *Toyota "Unintended Acceleration" Has Killed 89.*
- Heartbleed bug in OpenSSL, April 2014
- The DAO attack on the Ethereum Blockchain in June 2016
- ...

Ariane 5



Maiden flight of the Ariane 5 Launcher, 4 June 1996

Ariane 5



40s after launch...

Cause: software error

- arithmetic overflow in unprotected data conversion
from 64-bit float to 16-bit integer types

```
P_M_DERIVE(T_ALG.E_BH) :=  
UC_16S_EN_16NS (TDB.T_ENTIER_16S  
((1.0/C_M_LSB_BH) * G_M_INFO_DERIVE(T_ALG.E_BH)));
```

- software exception not caught
⇒ computer switched off
- all backup computers run the same software
⇒ all computers switched off, no guidance
⇒ rocket self-destructs

The DAO Attack on the Ethereum Blockchain — 2016

Timeline

30th April The DAO is launched with a 28 day crowd-founding window

15th May More than 100 million US dollars were raised

12th June Stephan Tual, one of The DAO's creators:

- a “recursive call bug” has been found in the software
- ... but “no DAO funds [are] at risk”.

by 18th June More than 3.6m ether (\approx 50m US dollars) were drained from the DAO account using that bug

15th July Ethereum splits in two

ETH Rewrite the history to reverse effects of the attack

ETC Accept the aftermath of the attack

Who cares?

- No one is legally responsible for bugs:

This software is distributed WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

- Even more, one can even make money out of bugs!
(customers buy the next version to get around bugs in software)

How can we avoid such failures?

- Choose a common programming language.

C (low level) / Ada, Java (high level)

- Carefully design the software.

There exists many software development methods

- Test the software extensively.

How can we avoid such failures?

- Choose a common programming language.

C (low level) / Ada, Java (high level)

yet, Ariane 5 software was written in Ada

- Carefully design the software.

There exists many software development methods

yet, critical embedded software follow strict development processes

- Test the software extensively.

yet, the erroneous code was well tested. . . on Ariane 4

Not sufficient!

How can we avoid such failures?

- Choose a common programming language.

C (low level) / Ada, Java (high level)

yet, Ariane 5 software was written in Ada

- Carefully design the software.

There exists many software development methods

yet, critical embedded software follow strict development processes

- Test the software extensively.

yet, the erroneous code was well tested. . . on Ariane 4

Not sufficient!

We should use **Formal Methods** and **Formal Verification!**

(provide rigorous, mathematical insurance ♥)

Formal verification - overview



Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

(Edsger Dijkstra)

izquotes.com

Formal methods

Formal methods are a particular kind of mathematically based techniques for

- specification
- development
- verification

of software and hardware systems.

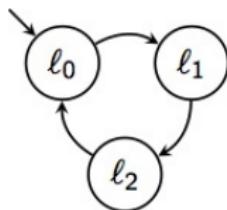


THE ONLY THING HARDER TO SELL THAN FORMAL METHODS

Formal program verification

Formal program verification is about proving properties of programs using logic and mathematics.

In particular, it is about proving they meet their specifications.



Program



Specification

Formal program verification

Rice's Theorem (1951):

The question $\text{Program} \models \text{Specification}$

is **undecidable!**

Automated software verification by formal methods is undecidable whence thought to be impossible.

Formal program verification

Rice's Theorem (1951):

The question $\text{Program} \models \text{Specification}$ 
is **undecidable!**

Automated software verification by formal methods is undecidable whence thought to be impossible.

There are powerful workarounds!

Current state of the art

We can check for the absence of large categories of bugs
(maybe not all of them but a significant portion of them).

Some bugs can be found completely automatically,
without any human intervention.

What gets verified?

- Hardware
- Compilers
- Programs
- Specifications

Current state of the art

Powerful tools/programming languages for **formal verification**:

- Infer
- Spark Pro
- Dafny
- KeY
- Alloy
- ASTRÉE
- Terminator
- Frama-C
- Model checkers
- SAT solvers
- SMT solvers
- VeriFast
- SAGE
- KLEE
- Spec #
- ...

Tools for static code analysis

Model checking tools

More tools

Why don't more people use formal verification?

- Time consuming
- Expensive

Why don't more people use formal verification?

- Time consuming
- Expensive

Formal or Informal?

- The question of whether to verify formally or not ultimately comes down to how disastrous occasional failure would be.

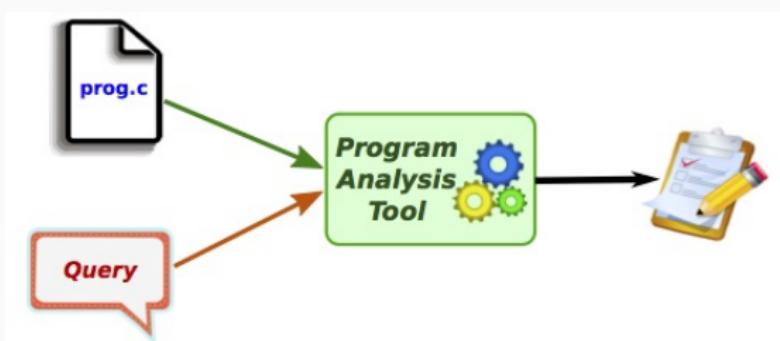
Formal verification methods

- Axiomatic semantics
- Deductive verification
 - SAT solvers
 - SMT solvers
 - Interactive theorem provers
 - Automatic theorem provers
- Model checking
- Abstract interpretation
- Type systems
- Lightweight formal methods
- Proof Assistants
- ...

Program analysis

What is Program analysis?

- Very broad topic, but generally speaking, automated analysis of program behaviour.
- Program analysis is about developing algorithms and tools that can analyze other programs.



source: Lecture Notes "A Gentle Introduction to Program Analysis" by Işıl Dilig

Applications of program analysis

- Bug finding

e.g., expose as many assertion failures as possible

- Security

e.g., does an app leak private user data?

- Verification

e.g., does the program analysis behave according to its specifications?

- Compiler optimizations

e.g., which variables should be kept in registers for fastest memory access?

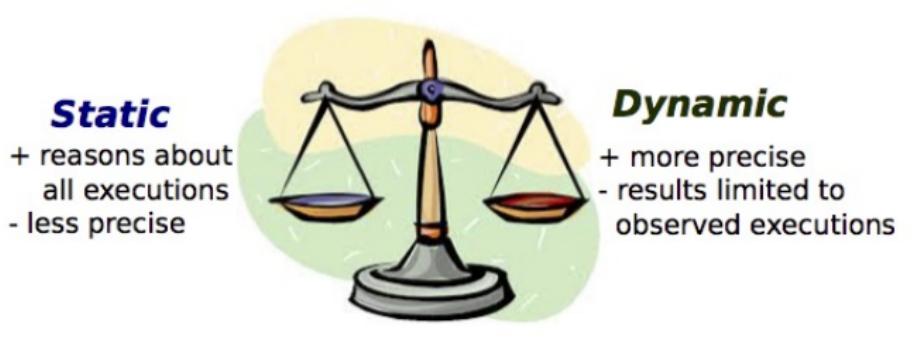
- Automatic parallelization

e.g., is it safe to execute different loop iterations on parallel?

Dynamic vs. Static Program Analysis

Two flavours of program analysis:

- **Dynamic analysis:** analyses programs while it is running
- **Static analysis:** analyses **source code** of the program



source: Lecture Notes "A Gentle Introduction to Program Analysis" by Işıl Dilig

Static Analysis

- Program analysis without running the program.
- It is nearly as old as programming
- It is a big field, with different approaches and applications
- [List of tools for static code analysis](#)
- Analysis paradigms:
 - Type systems
 - Dataflow analysis
 - Model checking

Type Systems

- Types are most widely used static analysis.
- A type is an example of an **abstract** value
 - Represents a set of concrete values
 - Every static analysis has abstract values
- A **type system** is a **tractable syntactic method** for proving the absence of certain program behaviours by classifying phrases according to the **kinds of values** they compute.
- Pierce.
- Unfortunately, we will not cover this topic in this course.

Formal semantics of programs

Formal semantics

To analyze/reason about programs, we must know what they mean.

Formal semantics

To analyze/reason about programs, we must know what they mean.

Formal semantics - three approaches:

Formal semantics

To analyze/reason about programs, we must know what they mean.

Formal semantics - three approaches:

- Operational semantics
 - Models program by its execution on an abstract machine
 - Useful for implementing compilers and interpreters

Formal semantics

To analyze/reason about programs, we must know what they mean.

Formal semantics - three approaches:

- Operational semantics
 - Models program by its execution on an abstract machine
 - Useful for implementing compilers and interpreters
- Denotational semantics
 - Models program as mathematical objects
 - Useful for theoretical foundations

Formal semantics

To analyze/reason about programs, we must know what they mean.

Formal semantics - three approaches:

- Operational semantics
 - Models program by its execution on an abstract machine
 - Useful for implementing compilers and interpreters
- Denotational semantics
 - Models program as mathematical objects
 - Useful for theoretical foundations
- Axiomatic semantics
 - Models program by the logical formulas it obeys
 - Useful for proving program correctness

Why just few languages have a formal semantics?

Too Hard?

- Modeling a real-world language is hard
- Notation can get very dense
- Sometimes requires developing new mathematics
- Not yet cost-effective for everyday use

Overly General?

- Explains the behaviour of a program on **every** input
- Most programmers are content knowing the behavior of their program on this input (or these inputs)

Who needs semantics?

Unambiguous description

- Anyone who wants to design a new feature
- Basis for most formal arguments
- Standard tool in Programming Languages research

Exhaustive reasoning

- Sometimes have to know behaviour on all inputs
- Compilers and interpreters
- Static analysis tools
- Program transformation tools
- Critical software

Operational semantics

- Gordon Plotkin in the 1980s
- Describe how a valid program is interpreted as sequences of computational steps. These sequences are the meaning of the program.
- Works well for sequential, object-oriented programs, parallel, distributed programs.



Operational semantics

- Describes **how** programs compute
- Relatively easy to define
- Close connection to implementation
- This is the most popular style of semantics

Operational semantics

- Evaluation is described as **transitions** in some (typically idealized) **abstract machine**. The state of the machine described by current expression.
- There are different styles of abstract machines.
- The **meaning** of a program can be
 - its fully reduced form (aka a **value**), for deterministic languages
 - all its possible executions and interactions, for nondeterministic/interactive languages
- A **small-step** semantics describes how such an execution proceeds in terms of successive reductions.

Example (Small-step semantics)

- Assume an abstract machine whose **configurations** have two components:
 - the **expression** e being evaluated
 - a **store** σ that records the values of variables
- Then, a small-step semantics describes the execution as a succession of one-step transitions of the form $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$

Example (Small-step semantics)

- Assume an abstract machine whose **configurations** have two components:
 - the **expression** e being evaluated
 - a **store** σ that records the values of variables
- Then, a small-step semantics describes the execution as a succession of one-step transitions of the form $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$

$\langle \text{int } x = 0; x = x + 1; , \emptyset \rangle$

Example (Small-step semantics)

- Assume an abstract machine whose **configurations** have two components:
 - the **expression** e being evaluated
 - a **store** σ that records the values of variables
- Then, a small-step semantics describes the execution as a succession of one-step transitions of the form $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$

$\langle \text{int } x = 0; x = x + 1; , \emptyset \rangle \rightarrow \langle x = x + 1; , x \mapsto 0 \rangle$

Example (Small-step semantics)

- Assume an abstract machine whose [configurations](#) have two components:
 - the [expression](#) e being evaluated
 - a [store](#) σ that records the values of variables
- Then, a small-step semantics describes the execution as a succession of one-step transitions of the form $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$

$$\begin{aligned} \langle \text{int } x = 0; x = x + 1; , \emptyset \rangle &\rightarrow \langle x = x + 1; , x \mapsto 0 \rangle \\ &\rightarrow \langle x = 0 + 1; , x \mapsto 0 \rangle \end{aligned}$$

Operational semantics

Example (Small-step semantics)

- Assume an abstract machine whose [configurations](#) have two components:
 - the [expression](#) e being evaluated
 - a [store](#) σ that records the values of variables
- Then, a small-step semantics describes the execution as a succession of one-step transitions of the form $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$

$\langle \text{int } x = 0; x = x + 1; , \emptyset \rangle \rightarrow \langle x = x + 1; , x \mapsto 0 \rangle$
 $\rightarrow \langle x = 0 + 1; , x \mapsto 0 \rangle$
 $\rightarrow \langle x = 1; , x \mapsto 0 \rangle$

Operational semantics

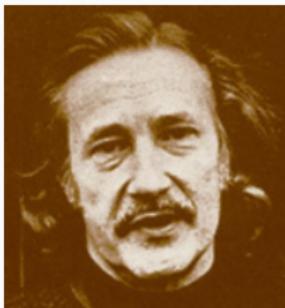
Example (Small-step semantics)

- Assume an abstract machine whose [configurations](#) have two components:
 - the [expression](#) e being evaluated
 - a [store](#) σ that records the values of variables
- Then, a small-step semantics describes the execution as a succession of one-step transitions of the form $\langle e, \sigma \rangle \rightarrow \langle e', \sigma' \rangle$

$$\begin{aligned} \langle \text{int } x = 0; x = x + 1; , \emptyset \rangle &\rightarrow \langle x = x + 1; , x \mapsto 0 \rangle \\ &\rightarrow \langle x = 0 + 1; , x \mapsto 0 \rangle \\ &\rightarrow \langle x = 1; , x \mapsto 0 \rangle \\ &\rightarrow \langle \{ \} , x \mapsto 1 \rangle \end{aligned}$$

Denotational semantics

- Christopher Strachey and Dana Scott published in the early 1970s



- Construct mathematical objects that describe the meaning of the blocks in the language
- Works well for sequential programs, but it gets a lot more complicated for parallel and distributed programs.

Axiomatic semantics

- Operational and denotational semantics let us reason about the meaning of a program.
- Axiomatic semantics define a program's meaning in terms of **what one can prove about it**.
- Useful for reasoning about correctness of programs

Quiz time!

<https://www.questionpro.com/t/AT4NiZrH1a>

See you next time!

C02 – Hoare Logic



Program Verification

FMI · Denisa Diaconescu · Spring 2022

Overview

What problem are we trying to solve?

Hoare Logic

**What problem are we trying to
solve?**

Verification for Imperative Languages

- Imperative languages are built around a **program state** (data stored in memory).
- Imperative programs are sequences of **commands that modify that state**.

Verification for Imperative Languages

- Imperative languages are built around a **program state** (data stored in memory).
- Imperative programs are sequences of **commands that modify that state**.
- To prove properties of imperative programs, we need
 1. A way of expressing assertions about program states.
 2. Rules for manipulating and proving those assertions.

Verification for Imperative Languages

- Imperative languages are built around a **program state** (data stored in memory).
- Imperative programs are sequences of **commands that modify that state**.
- To prove properties of imperative programs, we need
 1. A way of expressing assertions about program states.
 2. Rules for manipulating and proving those assertions.
- These will be provided by **Hoare Logic**.

Verification for programming languages

The formalisms we will see can be extended to verify

- Recursive Programs
- Object Oriented Programs
- Parallel Programs
- Distributed Programs

Tools

- [Dafny](#) – Microsoft
- [Infer](#) – Facebook
- [VeriFast](#)
- [SmallFoot](#)
- ...

Hoare Logic

C.A.R. (Tony) Hoare

Hoare Logic was introduced by Tony Hoare.

He also invented Quicksort algorithm in 1960 (when he was 26).



A simple imperative programming language

To prove things about programs, we first need to fix a programming language.
We will define a **little language** with four different kinds of statement.

A simple imperative programming language

To prove things about programs, we first need to fix a programming language.

We will define a **little language** with four different kinds of statement.

- **Assignment** – $x := E$
 - x is a variable and E is an expression built from variables and arithmetics that returns a number, e.g., $2+3$, $x*y+1$, ...

A simple imperative programming language

To prove things about programs, we first need to fix a programming language.

We will define a **little language** with four different kinds of statement.

- **Assignment** – $x := E$
 - x is a variable and E is an expression built from variables and arithmetics that returns a number, e.g., $2+3$, $x*y+1$, ...
- **Sequencing** – $C_1 ; C_2$

A simple imperative programming language

To prove things about programs, we first need to fix a programming language.

We will define a **little language** with four different kinds of statement.

- **Assignment** – $x := E$
 - x is a variable and E is an expression built from variables and arithmetics that returns a number, e.g., $2+3$, $x*y+1$, ...
- **Sequencing** – $C_1 ; C_2$
- **Conditional** – $\text{if } B \text{ then } C_1 \text{ else } C_2$
 - B is an expression built from variables, arithmetics, and logic that returns a boolean value, e.g., $y < 0$, $x \neq y \wedge z = 0$, ...

A simple imperative programming language

To prove things about programs, we first need to fix a programming language.

We will define a **little language** with four different kinds of statement.

- **Assignment** – $x := E$
 - x is a variable and E is an expression built from variables and arithmetics that returns a number, e.g., $2+3$, $x*y+1$, ...
- **Sequencing** – $C_1 ; C_2$
- **Conditional** – $\text{if } B \text{ then } C_1 \text{ else } C_2$
 - B is an expression built from variables, arithmetics, and logic that returns a boolean value, e.g., $y < 0$, $x \neq y \wedge z = 0$, ...
- **While** – $\text{while } B \text{ do } C$

Hoare triples

A Hoare triple $\{P\} \ C \ \{Q\}$ has three components:

P a precondition

C a code fragment

Q a postcondition

The precondition is an assertion saying something of interest about the state before the code is executed.

The postcondition is an assertion saying something of interest about the state after the code is executed.

The **precondition** and **postcondition** will be built from **program variables**, **numbers**, **basic arithmetics relations**, and use **propositional logic** to combine simple assertions.

Example

- $x = 3$
- $x \neq y$
- $x > 0$
- $x = 4 \wedge y = 2$
- $(x > y) \rightarrow (x = 2 * y)$
- \top
- \perp

Semantics

A **state** is determined by the values given to the program variables.

In our little language all our variables will store numbers only!

A **state** is determined by the values given to the program variables.

In our little language all our variables will store numbers only!

Hoare Triple: $\{P\} \text{ C } \{Q\}$

- **if** the pre-state satisfies P
- **and** the program C terminates
- **then** the post-state satisfies Q

Partial Correctness

Hoare logic expresses **partial correctness**!

- A program is **partially correct** if it gives the right answer whenever it terminates.
- It never gives a wrong answer, but it may give no answer at all.

Partial Correctness

Hoare logic expresses **partial correctness**!

- A program is **partially correct** if it gives the right answer whenever it terminates.
- It never gives a wrong answer, but it may give no answer at all.

Example

$$\{x = 1\} \text{ while } x = 1 \text{ do } y := 2 \{x = 3\}$$

is **true** in the Hoare logic semantics.

- if pre-state satisfies $x = 1$ **and** the while loop terminates then the post-state satisfies $x = 3$. **But the while loop does not terminate!**

Partial Correctness is OK

Why not insist on termination?

- It simplifies the logic.
- If necessary, we can prove termination separately.

We will come back to termination with the [Weakest Precondition Calculus](#).

Example

Hoare logic will allow us to make claims such as:

$$\{x > 0\} \quad y := 0 - x \quad \{y < 0 \wedge x \neq y\}$$

Example

Hoare logic will allow us to make claims such as:

$$\{x > 0\} \quad y := 0 - x \quad \{y < 0 \wedge x \neq y\}$$

If $(x > 0)$ is true **before** $y := 0 - x$ is executed then $(y < 0 \wedge x \neq y)$ is true **afterwards**.

Example

Hoare logic will allow us to make claims such as:

$$\{x > 0\} \quad y := 0 - x \quad \{y < 0 \wedge x \neq y\}$$

If $(x > 0)$ is true **before** $y := 0 - x$ is executed then $(y < 0 \wedge x \neq y)$ is true **afterwards**.

This particular reasoning is intuitively true. **How to prove this?**

Example

Hoare logic will allow us to make claims such as:

$$\{x > 0\} \quad y := 0 - x \quad \{y < 0 \wedge x \neq y\}$$

If $(x > 0)$ is true before $y := 0 - x$ is executed then $(y < 0 \wedge x \neq y)$ is true afterwards.

This particular reasoning is intuitively true. How to prove this?

We need a calculus – a collection of rules for (formally) manipulating the triples. We will have one rule for each of our four kinds of statement (plus two other rules).

The Assignment Axiom (Rule 1/6)

Assignments change the state so we expect Hoare triples for assignments to reflect that change.

The Assignment Axiom (Rule 1/6)

Assignments change the state so we expect Hoare triples for assignments to reflect that change.

The assignment axiom:

$$\{Q[x/\mathbb{E}]\} \ x := \mathbb{E} \ {Q}$$

(Q is an assertion involving a variable x and $Q[x/\mathbb{E}]$ indicates the same assertion with all occurrences of x replaced by the expression \mathbb{E})

The Assignment Axiom (Rule 1/6)

Assignments change the state so we expect Hoare triples for assignments to reflect that change.

The assignment axiom:

$$\{Q[x/\mathbb{E}]\} \ x := \mathbb{E} \ {Q}$$

(Q is an assertion involving a variable x and $Q[x/\mathbb{E}]$ indicates the same assertion with all occurrences of x replaced by the expression \mathbb{E})

If we want x to have some property Q after the assignment, then that property must hold for the value \mathbb{E} assigned to x before the assignment is executed.

The Assignment Axiom

Example

The **backwards** rule is false: $\{Q\} \ x := E \ \{Q[x/E]\}$

If we want to apply this wrong "axiom" to the precondition $x = 0$ and code fragment $x := 1$ we would get

$$\{x = 0\} \ x := 1 \ \{1 = 0\}$$

which says "if $x = 0$ initially and $x := 1$ terminates then $1 = 0$ finally".

Work from the Goal backwards

It may seem natural to start at the **precondition** and reason **towards the postcondition**, but this is not the best way to do Hoare logic.

Instead you **start with the goal (postcondition)** and go "backwards".

Work from the Goal backwards

It may seem natural to start at the **precondition** and reason **towards the postcondition**, but this is not the best way to do Hoare logic.

Instead you **start with the goal (postcondition)** and go "backwards".

Example

To apply the assignment axiom

$$\{Q[x/\mathbb{E}]\} \ x := \mathbb{E} \ \{Q\}$$

take the postcondition, copy it across the precondition and then replace all occurrences of x with \mathbb{E} .

Note that the postcondition may have no, one, or many occurrences of x .

All get replaced by \mathbb{E} in the precondition!

Proof rule for The Assignment

Assignment Axiom: $\{Q[x/\mathbb{E}]\} \ x := e \ {Q}$

Example

Suppose the code fragment is $x := 2$ and suppose the desired postcondition is $y = x$.

An instance of the assignment axiom:

Proof rule for The Assignment

Assignment Axiom: $\{Q[x/\mathbb{E}]\} \ x := e \ {Q}$

Example

Suppose the code fragment is $x := 2$ and suppose the desired postcondition is $y = x$.

An instance of the assignment axiom:

$$\{y = 2\} \ x := 2 \ {y = x}$$

Proof rule for The Assignment

You can always replace predicates by equivalent predicates; just label your proof step with "precondition equivalence" or "postcondition equivalence".

Proof rule for The Assignment

You can always replace predicates by equivalent predicates; just label your proof step with "precondition equivalence" or "postcondition equivalence".

Example

How should we prove

$$\{y > 0\} \ x := y+3 \ {x > 3}\text{?}$$

Proof rule for The Assignment

You can always replace predicates by equivalent predicates; just label your proof step with "precondition equivalence" or "postcondition equivalence".

Example

How should we prove

$$\{y > 0\} \ x := y+3 \ {x > 3}?$$

Start with the postcondition $x > 3$ and apply the assignment axiom:

$$\{y + 3 > 3\} \ x := y+3 \ {x > 3}$$

Proof rule for The Assignment

You can always replace predicates by equivalent predicates; just label your proof step with "precondition equivalence" or "postcondition equivalence".

Example

How should we prove

$$\{y > 0\} \ x := y+3 \ {x > 3}?$$

Start with the postcondition $x > 3$ and apply the assignment axiom:

$$\{y + 3 > 3\} \ x := y+3 \ {x > 3}$$

Then use the fact that $y + 3 > 3$ is equivalent with $y > 0$ to get the result.

Proof rule for The Assignment

Example

What if we want to prove

$$\{y = 2\} \ x := y \ \{x > 0\}?$$

Proof rule for The Assignment

Example

What if we want to prove

$$\{y = 2\} \ x := y \{x > 0\}?$$

This is clearly true. But the assignment axiom gives us:

$$\{y > 0\} \ x := y \{x > 0\}$$

We cannot just replace $y > 0$ with $y = 2$ – **they are not equivalent!**

Weak and strong predicates

A predicate P is **stronger** than Q if P implies Q .

If P is stronger than Q , then whenever P is true then Q is true as well.

Weak and strong predicates

A predicate P is **stronger** than Q if P implies Q .

If P is stronger than Q , then whenever P is true then Q is true as well.

Example

A politician's example:

- *I will keep unemployment below 3%* is **stronger** than
- *I will keep unemployment below 15%*

Weak and strong predicates

A predicate P is **stronger** than Q if P implies Q .

If P is stronger than Q , then whenever P is true then Q is true as well.

Example

A politician's example:

- *I will keep unemployment below 3%* is **stronger** than
- *I will keep unemployment below 15%*

The **strongest** possible assertion is \perp .

The **weakest** possible assertion is \top .

Strong postconditions

Example

The Hoare triple $\{x = 5\} \ x := x+1 \ \{x = 6\}$ says more about the code than does $\{x = 5\} \ x := x+1 \ \{x > 0\}$.

Strong postconditions

Example

The Hoare triple $\{x = 5\} \text{ } x := x+1 \text{ } \{x = 6\}$ says more about the code than does $\{x = 5\} \text{ } x := x+1 \text{ } \{x > 0\}$.

If a **postcondition** Q_1 is **stronger** than Q_2 , then
 $\{P\} \text{ } \mathbb{C} \text{ } \{Q_1\}$ is a **stronger** statement than $\{P\} \text{ } \mathbb{C} \text{ } \{Q_2\}$.

Strong postconditions

Example

The Hoare triple $\{x = 5\} \ x := x+1 \ \{x = 6\}$ says more about the code than does $\{x = 5\} \ x := x+1 \ \{x > 0\}$.

If a postcondition Q_1 is stronger than Q_2 , then $\{P\} \mathbb{C} \{Q_1\}$ is a stronger statement than $\{P\} \mathbb{C} \{Q_2\}$.

Example

Since the postcondition $x = 6$ is stronger than $x > 0$ (as $x = 6 \rightarrow x > 0$), then $\{x = 5\} \ x := x+1 \ \{x = 6\}$ is a stronger statement than $\{x = 5\} \ x := x+1 \ \{x > 0\}$.

Weak preconditions

Example

The Hoare triple $\{x > 0\} \ x := x+1 \ {x > 1}$ says more about the code than does $\{x = 5\} \ x := x+1 \ {x > 1}$.

If a precondition P_1 is weaker than P_2 , then

$\{P_1\} \ C \ {Q}$ is a stronger statement than $\{P_2\} \ C \ {Q}$.

Example

Since the precondition $x > 0$ is weaker than $x = 5$,

then $\{x > 0\} \ x := x+1 \ {x > 1}$ is a stronger statement than $\{x = 5\} \ x := x+1 \ {x > 1}$.

Proof rule for Strengthening Preconditions (Rule 2/6)

It is safe (sound) to make a **precondition** more *specific* (**stronger**).

Precondition Strengthening rule:

$$\frac{P_s \rightarrow P_w \quad \{P_w\} \subset \{Q\}}{\{P_s\} \subset \{Q\}}$$

Example

$$\frac{y = 2 \rightarrow y > 0 \quad \{y > 0\} \ x := y \ \{x > 0\}}{\{y = 2\} \ x := y \ \{x > 0\}}$$

Proof rule for Weakening Postconditions (Rule 3/6)

It is safe (sound) to make a **postcondition** less *specific* (**weaker**).

Postcondition Weakening rule:

$$\frac{\{P\} \subseteq \{Q_s\} \quad Q_s \rightarrow Q_w}{\{P\} \subseteq \{Q_w\}}$$

Example

$$\frac{\{x > 2\} \ x := x + 1 \ \{x > 3\} \quad x > 3 \rightarrow x > 1}{\{x > 2\} \ x := x + 1 \ \{x > 1\}}$$

Proof rule for Sequencing (Rule 4/6)

Imperative programs consist of a sequence of statements, affecting the state one after the other.

Sequencing rule:

$$\frac{\{P\}C_1\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}}$$

Proof rule for Sequencing (Rule 4/6)

Imperative programs consist of a sequence of statements, affecting the state one after the other.

Sequencing rule:

$$\frac{\{P\}C_1\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}}$$

Example

$$\frac{\{x > 2\}x := x + 1\{x > 3\} \quad \{x > 3\}x := x + 2\{x > 5\}}{\{x > 2\}x := x + 1 ; x := x + 2\{x > 5\}}$$

How do we get the intermediate condition?

In the rule

$$\frac{\{P\}C_1\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}}$$

our precondition P and postcondition R will be given to us, but how do we come up with Q ?

How do we get the intermediate condition?

In the rule

$$\frac{\{P\}C_1\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}C_1; C_2\{R\}}$$

our precondition P and postcondition R will be given to us, but how do we come up with Q ?

By starting with our goal R and [working backwards!](#)

$$\frac{\{x > 2\}x := x + 1\{Q\} \quad \{Q\}x := x + 2\{x > 5\}}{\{x > 2\}x := x + 1; x := x + 2\{x > 5\}}$$

Laying out a proof

Example

Suppose we want to prove

$$\{x = 3\} \text{ } x := x+1; \text{ } x := x+2 \{x > 5\}.$$

Laying out a proof

Example

Suppose we want to prove

$$\{x = 3\} \ x := x+1; \ x := x+2 \{x > 5\}.$$

Note the numbered proof steps and justifications!

1. $\{x + 2 > 5\} \ x := x+2 \{x > 5\}$ (Assignment)

Laying out a proof

Example

Suppose we want to prove

$$\{x = 3\} \ x := x+1; \ x := x+2 \{x > 5\}.$$

Note the numbered proof steps and justifications!

1. $\{x + 2 > 5\} \ x := x+2 \{x > 5\}$ (Assignment)
2. $\{x > 3\} \ x := x+2 \{x > 5\}$ (1, Precondition Equivalence)

Laying out a proof

Example

Suppose we want to prove

$$\{x = 3\} \ x := x+1; \ x := x+2 \{x > 5\}.$$

Note the numbered proof steps and justifications!

1. $\{x + 2 > 5\} \ x := x+2 \{x > 5\}$ (Assignment)
2. $\{x > 3\} \ x := x+2 \{x > 5\}$ (1, Precondition Equivalence)
3. $\{x + 1 > 3\} \ x := x+1 \{x > 3\}$ (Assignment)

Laying out a proof

Example

Suppose we want to prove

$$\{x = 3\} \ x := x+1; \ x := x+2 \{x > 5\}.$$

Note the numbered proof steps and justifications!

1. $\{x + 2 > 5\} \ x := x+2 \{x > 5\}$ (Assignment)
2. $\{x > 3\} \ x := x+2 \{x > 5\}$ (1, Precondition Equivalence)
3. $\{x + 1 > 3\} \ x := x+1 \{x > 3\}$ (Assignment)
4. $\{x > 2\} \ x := x+1 \{x > 3\}$ (3, Precondition Equivalence)

Laying out a proof

Example

Suppose we want to prove

$$\{x = 3\} \ x := x+1; \ x := x+2 \{x > 5\}.$$

Note the numbered proof steps and justifications!

1. $\{x + 2 > 5\} \ x := x+2 \{x > 5\}$ (Assignment)
2. $\{x > 3\} \ x := x+2 \{x > 5\}$ (1, Precondition Equivalence)
3. $\{x + 1 > 3\} \ x := x+1 \{x > 3\}$ (Assignment)
4. $\{x > 2\} \ x := x+1 \{x > 3\}$ (3, Precondition Equivalence)
5. $\{x > 2\} \ x := x+1; \ x := x+2 \{x > 5\}$ (2,4, Sequencing)

Laying out a proof

Example

Suppose we want to prove

$$\{x = 3\} \ x := x+1; \ x := x+2 \{x > 5\}.$$

Note the numbered proof steps and justifications!

1. $\{x + 2 > 5\} \ x := x+2 \{x > 5\}$ (Assignment)
2. $\{x > 3\} \ x := x+2 \{x > 5\}$ (1, Precondition Equivalence)
3. $\{x + 1 > 3\} \ x := x+1 \{x > 3\}$ (Assignment)
4. $\{x > 2\} \ x := x+1 \{x > 3\}$ (3, Precondition Equivalence)
5. $\{x > 2\} \ x := x+1; \ x := x+2 \{x > 5\}$ (2,4, Sequencing)
6. $x = 3 \rightarrow x > 2$ (Basic arithmetics)

Laying out a proof

Example

Suppose we want to prove

$$\{x = 3\} \ x := x+1; \ x := x+2 \{x > 5\}.$$

Note the numbered proof steps and justifications!

1. $\{x + 2 > 5\} \ x := x+2 \{x > 5\}$ (Assignment)
2. $\{x > 3\} \ x := x+2 \{x > 5\}$ (1, Precondition Equivalence)
3. $\{x + 1 > 3\} \ x := x+1 \{x > 3\}$ (Assignment)
4. $\{x > 2\} \ x := x+1 \{x > 3\}$ (3, Precondition Equivalence)
5. $\{x > 2\} \ x := x+1; \ x := x+2 \{x > 5\}$ (2,4, Sequencing)
6. $x = 3 \rightarrow x > 2$ (Basic arithmetics)
7. $\{x = 3\} \ x := x+1; \ x := x+2 \{x > 5\}.$ (5,6, Precondition Strength)

Proof rule for Conditionals (Rule 5/6)

Conditional rule:

$$\frac{\{P \wedge B\} C_1 \{Q\} \quad \{P \wedge \neg B\} C_2 \{Q\}}{\{P\} \text{ if } B \text{ then } C_1 \text{ else } C_2 \{Q\}}$$

Proof rule for Conditionals (Rule 5/6)

Conditional rule:

$$\frac{\{P \wedge \mathbb{B}\} C_1 \{Q\} \quad \{P \wedge \neg \mathbb{B}\} C_2 \{Q\}}{\{P\} \text{ if } \mathbb{B} \text{ then } C_1 \text{ else } C_2 \{Q\}}$$

- When a conditional is executed, either C_1 or C_2 is executed.
- Therefore, if the **conditional** is to establish Q , then **both** C_1 and C_2 must establish Q .
- Similarly, if the precondition for the **conditional** is P , then it must also be a precondition for the **both** branches C_1 and C_2 .
- The choice between C_1 and C_2 depends on evaluating b in the initial state, so we can also assume \mathbb{B} to be a precondition for C_1 and $\neg \mathbb{B}$ to be a precondition for C_2 .

Proof rule for Conditionals

Conditional rule:

$$\frac{\{P \wedge \mathbb{B}\} \mathbb{C}_1 \{Q\} \quad \{P \wedge \neg \mathbb{B}\} \mathbb{C}_2 \{Q\}}{\{P\} \text{ if } \mathbb{B} \text{ then } \mathbb{C}_1 \text{ else } \mathbb{C}_2 \{Q\}}$$

Example

Suppose we wish to prove

$$\{x > 2\} \text{ if } x > 2 \text{ then } y := 1 \text{ else } y := -1 \{y > 0\}$$

Proof rule for Conditionals

Conditional rule:

$$\boxed{\frac{\{P \wedge \mathbb{B}\} \mathbb{C}_1 \{Q\} \quad \{P \wedge \neg \mathbb{B}\} \mathbb{C}_2 \{Q\}}{\{P\} \text{ if } \mathbb{B} \text{ then } \mathbb{C}_1 \text{ else } \mathbb{C}_2 \{Q\}}}$$

Example

Suppose we wish to prove

$$\{x > 2\} \text{ if } x > 2 \text{ then } y := 1 \text{ else } y := -1 \{y > 0\}$$

The proof rule for conditionals suggests we prove:

$$\{(x > 2) \wedge (x > 2)\} y := 1 \{y > 0\} \quad \{(x > 2) \wedge \neg(x > 2)\} y := -1 \{y > 0\}$$

Simplifying the preconditions, we get

1. $\{x > 2\} y := 1 \{y > 0\}$
2. $\{\perp\} y := -1 \{y > 0\}$

Proof rule for Conditionals

Example (cont.)

For the subgoal 1. $\{x > 2\} \text{ y:=1 } \{y > 0\}$ we have the following proof

Proof rule for Conditionals

Example (cont.)

For the subgoal 1. $\{x > 2\} \text{ y:=1 } \{y > 0\}$ we have the following proof

3. $\{1 > 0\} \text{ y := 1 } \{y > 0\}$ (Assignment rule)

Proof rule for Conditionals

Example (cont.)

For the subgoal $1. \{x > 2\} \text{ y:=1 } \{y > 0\}$ we have the following proof

3. $\{1 > 0\} \text{ y := 1 } \{y > 0\}$ (Assignment rule)
4. $1 > 0 \leftrightarrow \top$ (Propositional logic)

Proof rule for Conditionals

Example (cont.)

For the subgoal $1. \{x > 2\} \textcolor{violet}{y := 1} \{y > 0\}$ we have the following proof

3. $\{1 > 0\} \textcolor{violet}{y := 1} \{y > 0\}$ (Assignment rule)
4. $1 > 0 \leftrightarrow \top$ (Propositional logic)
5. $\{\top\} \textcolor{violet}{y := 1} \{y > 0\}$ (Precondition equivalence)

Proof rule for Conditionals

Example (cont.)

For the subgoal $1. \{x > 2\} \textcolor{violet}{y := 1} \{y > 0\}$ we have the following proof

3. $\{1 > 0\} \textcolor{violet}{y := 1} \{y > 0\}$ (Assignment rule)
4. $1 > 0 \leftrightarrow \top$ (Propositional logic)
5. $\{\top\} \textcolor{violet}{y := 1} \{y > 0\}$ (Precondition equivalence)
6. $x > 2 \rightarrow \top$ (Propositional logic)

Proof rule for Conditionals

Example (cont.)

For the subgoal $1. \{x > 2\} \text{ y := 1 } \{y > 0\}$ we have the following proof

3. $\{1 > 0\} \text{ y := 1 } \{y > 0\}$ (Assignment rule)
4. $1 > 0 \leftrightarrow \top$ (Propositional logic)
5. $\{\top\} \text{ y := 1 } \{y > 0\}$ (Precondition equivalence)
6. $x > 2 \rightarrow \top$ (Propositional logic)
7. $\{x > 2\} \text{ y := 1 } \{y > 0\}$ (Precondition Strengthening)

Proof rule for Conditionals

Example (cont.)

For the subgoal 1. $\{x > 2\} \text{ y:=1 } \{y > 0\}$ we have the following proof

3. $\{1 > 0\} \text{ y := 1 } \{y > 0\}$ (Assignment rule)
4. $1 > 0 \leftrightarrow \top$ (Propositional logic)
5. $\{\top\} \text{ y := 1 } \{y > 0\}$ (Precondition equivalence)
6. $x > 2 \rightarrow \top$ (Propositional logic)
7. $\{x > 2\} \text{ y := 1 } \{y > 0\}$ (Precondition Strengthening)

For the subgoal 2. $\{\perp\} \text{ y:=-1 } \{y > 0\}$ we have the following proof

Proof rule for Conditionals

Example (cont.)

For the subgoal 1. $\{x > 2\} \text{ y:=1 } \{y > 0\}$ we have the following proof

3. $\{1 > 0\} \text{ y := 1 } \{y > 0\}$ (Assignment rule)
4. $1 > 0 \leftrightarrow \top$ (Propositional logic)
5. $\{\top\} \text{ y := 1 } \{y > 0\}$ (Precondition equivalence)
6. $x > 2 \rightarrow \top$ (Propositional logic)
7. $\{x > 2\} \text{ y := 1 } \{y > 0\}$ (Precondition Strengthening)

For the subgoal 2. $\{\perp\} \text{ y:=-1 } \{y > 0\}$ we have the following proof

8. $\{-1 > 0\} \text{ y := -1 } \{y > 0\}$ (Assignment rule)

Proof rule for Conditionals

Example (cont.)

For the subgoal 1. $\{x > 2\} \text{ y:=1 } \{y > 0\}$ we have the following proof

3. $\{1 > 0\} \text{ y := 1 } \{y > 0\}$ (Assignment rule)
4. $1 > 0 \leftrightarrow \top$ (Propositional logic)
5. $\{\top\} \text{ y := 1 } \{y > 0\}$ (Precondition equivalence)
6. $x > 2 \rightarrow \top$ (Propositional logic)
7. $\{x > 2\} \text{ y := 1 } \{y > 0\}$ (Precondition Strengthening)

For the subgoal 2. $\{\perp\} \text{ y:=-1 } \{y > 0\}$ we have the following proof

8. $\{-1 > 0\} \text{ y := -1 } \{y > 0\}$ (Assignment rule)
9. $-1 > 0 \leftrightarrow \perp$ (Propositional logic)

Proof rule for Conditionals

Example (cont.)

For the subgoal 1. $\{x > 2\} \text{ y := 1 } \{y > 0\}$ we have the following proof

3. $\{1 > 0\} \text{ y := 1 } \{y > 0\}$ (Assignment rule)
4. $1 > 0 \leftrightarrow \top$ (Propositional logic)
5. $\{\top\} \text{ y := 1 } \{y > 0\}$ (Precondition equivalence)
6. $x > 2 \rightarrow \top$ (Propositional logic)
7. $\{x > 2\} \text{ y := 1 } \{y > 0\}$ (Precondition Strengthening)

For the subgoal 2. $\{\perp\} \text{ y := -1 } \{y > 0\}$ we have the following proof

8. $\{-1 > 0\} \text{ y := -1 } \{y > 0\}$ (Assignment rule)
9. $-1 > 0 \leftrightarrow \perp$ (Propositional logic)
10. $\{\perp\} \text{ y := -1 } \{y > 0\}$ (Precondition equivalence)

Proof rule for Conditionals

Exercise:

How would you derive a rule for a conditional statement without `else`?

`if B then C`

Quiz time!

<https://www.questionpro.com/t/AT4NiZrRD9>

References

- Lecture Notes on "Formal Methods for Software Engineering", Australian National University, Rajeev Goré.
- Mike Gordon, "Specification and Verification I", chapters 1 and 2.
- Michael Huth, Mark Ryan, "Logic in Computer Science: Modeling and Reasoning about Systems", 2nd edition, Cambridge University Press, 2004.
- Krzysztof R. Apt, Frank S. de Boer, Ernst-Rüdiger Olderog, "Verification of Sequential and Concurrent Programs", 3rd edition, Springer.

C03 – Hoare Logic & Weakest Precondition calculus

Program Verification

FMI · Denisa Diaconescu · Spring 2022

Overview

Hoare Logic

Weakest Precondition calculus

Hoare Logic

Proof rules for Hoare logic

The assignment axiom:

$$\{Q(\mathbb{E})\} \ x := \mathbb{E} \ \{Q(x)\}$$

Precondition Strengthening rule:

$$\frac{P_s \rightarrow P_w \quad \{P_w\} \mathbb{C} \{Q\}}{\{P_s\} \mathbb{C} \{Q\}}$$

Postcondition Weakening rule:

$$\frac{\{P\} \mathbb{C} \{Q_s\} \quad Q_s \rightarrow Q_w}{\{P\} \mathbb{C} \{Q_w\}}$$

Sequencing rule:

$$\frac{\{P\} \mathbb{C}_1 \{Q\} \quad \{Q\} \mathbb{C}_2 \{R\}}{\{P\} \mathbb{C}_1; \mathbb{C}_2 \{R\}}$$

Conditional rule:

$$\frac{\{P \wedge \mathbb{B}\} \mathbb{C}_1 \{Q\} \quad \{P \wedge \neg \mathbb{B}\} \mathbb{C}_2 \{Q\}}{\{P\} \text{ if } \mathbb{B} \text{ then } \mathbb{C}_1 \text{ else } \mathbb{C}_2 \{Q\}}$$

Proof rule for While Loops

Suppose we want to prove

$$\{P\} \text{ while } B \text{ do } C \{Q\}$$

Proof rule for While Loops

Suppose we want to prove

$$\{P\} \text{ while } B \text{ do } C \{Q\}$$

While rule:

$$\frac{\{I \wedge B\} \vdash C \{I\}}{\{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\}}$$

Proof rule for While Loops

Suppose we want to prove

$$\{P\} \text{ while } B \text{ do } C \{Q\}$$

While rule:

$$\frac{\{I \wedge B\} \vdash C \{I\}}{\{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\}}$$

- I is called **loop invariant**
- I is true before we encounter the while statement, and remains true after each iteration of the loop (although not necessarily midway during execution of the loop body).
- If the loop terminates the loop condition must be false, so $\neg B$ appears in the postcondition.
- For the body of the loop C to execute, B needs to be true, so it appears in the precondition.

Proof rule for While Loops

Suppose we want to prove

$$\{P\} \text{ while } B \text{ do } C \{Q\}$$

While rule:

$$\frac{\{I \wedge B\} \vdash C \{I\}}{\{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\}}$$

- I is called **loop invariant**
- I is true before we encounter the while statement, and remains true after each iteration of the loop (although not necessarily midway during execution of the loop body).
- If the loop terminates the loop condition must be false, so $\neg B$ appears in the postcondition.
- For the body of the loop C to execute, B needs to be true, so it appears in the precondition.
- **The most difficult part** is to come up with the **invariant**.
- This requires **intuition**. There is **no algorithm** that will find the invariant.

Applying the while rule

How does the while rule helps to solve our problem?

$$\{P\} \text{ while } B \text{ do } C \{Q\}$$

$$\frac{\{I \wedge B\} \subseteq \{I\}}{\{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\}}$$

Applying the while rule

How does the while rule helps to solve our problem?

$$\{P\} \text{ while } B \text{ do } C \{Q\}$$

$$\frac{\{I \wedge B\} \subset \{I\}}{\{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\}}$$

- The postcondition we get after applying our rule has the form $I \wedge \neg B$. This might not be the same as the postcondition Q we want!

Applying the while rule

How does the while rule helps to solve our problem?

$$\{P\} \text{ while } B \text{ do } C \{Q\}$$

$$\frac{\{I \wedge B\} \subseteq \{I\}}{\{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\}}$$

- The postcondition we get after applying our rule has the form $I \wedge \neg B$. This might not be the same as the postcondition Q we want!
- If $(I \wedge \neg B) \leftrightarrow Q$, we can replace $I \wedge \neg B$ with Q .
- If $(I \wedge \neg B) \rightarrow Q$ we can use the Postcondition weakening rule:

$$\frac{\begin{array}{c} \{I \wedge B\} \subseteq \{I\} \\ \hline \{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\} \quad I \wedge \neg B \rightarrow Q \end{array}}{\{I\} \text{ while } B \text{ do } C \{Q\}}$$

Applying the while rule

How does the while rule helps to solve our problem?

$\{P\} \text{ while } B \text{ do } C \{Q\}$

$$\frac{\{I \wedge B\} \subseteq \{I\}}{\{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\}}$$

- Similarly, P and I might be different formulas.

Applying the while rule

How does the while rule helps to solve our problem?

$\{P\} \text{ while } B \text{ do } C \{Q\}$

$$\frac{\begin{array}{c} \{I \wedge B\} \subset \{I\} \\ \{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\} \end{array}}{\{P\} \text{ while } B \text{ do } C \{Q\}}$$

- Similarly, P and I might be different formulas.
- If $I \leftrightarrow P$, we can replace I with P to complete our proof.
- If $P \rightarrow I$ we can use the **Precondition strengthening rule**:

$$\frac{P \rightarrow I \quad \{I\} \text{ while } B \text{ do } C \{Q\}}{\{P\} \text{ while } B \text{ do } C \{Q\}}$$

Proof rule for While Loops

Example

Suppose we want to find a precondition P such that

$$\{P\} \text{ while } (n > 0) \text{ do } n := n-1 \{n = 0\}$$

Proof rule for While Loops

Example

Suppose we want to find a precondition P such that

$$\{P\} \text{ while } (n > 0) \text{ do } n := n-1 \{n = 0\}$$

We want a loop invariant I such that

- if I is true initially
- I remains true each time around the loop
- $I \wedge \neg(n > 0) \rightarrow (n = 0)$

Proof rule for While Loops

Example

Suppose we want to find a precondition P such that

$$\{P\} \text{ while } (n > 0) \text{ do } n := n-1 \{n = 0\}$$

We want a loop invariant I such that

- if I is true initially
- I remains true each time around the loop
- $I \wedge \neg(n > 0) \rightarrow (n = 0)$

$I \equiv n \geq 0$ looks like a reasonable loop invariant.

The premise of the while rule then follows from the assignment axiom.

Proof rule for While Loops

While rule:

$$\frac{\{I \wedge B\} \subseteq \{I\}}{\{I\} \text{ while } B \text{ do } C \subseteq \{I \wedge \neg B\}}$$

Example (cont.)

Suppose we want to find a precondition P such that

$$\{P\} \text{ while } (n > 0) \text{ do } n := n-1 \{n = 0\}$$

1. $\{n - 1 \geq 0\} \text{ n } := \text{n-1 } \{n \geq 0\}$ (Assignment rule)

Proof rule for While Loops

While rule:

$$\frac{\{I \wedge B\} \subseteq \{I\}}{\{I\} \text{ while } B \text{ do } C \subseteq \{I \wedge \neg B\}}$$

Example (cont.)

Suppose we want to find a precondition P such that

$$\{P\} \text{ while } (n > 0) \text{ do } n := n-1 \{n = 0\}$$

1. $\{n - 1 \geq 0\} \text{ n := n-1 } \{n \geq 0\}$ (Assignment rule)
2. $\{n \geq 0 \wedge n > 0\} \text{ n := n-1 } \{n \geq 0\}$ (1, Precond. Equiv.)

Proof rule for While Loops

While rule:

$$\frac{\{I \wedge B\} \subseteq \{I\}}{\{I\} \text{ while } B \text{ do } C \subseteq \{I \wedge \neg B\}}$$

Example (cont.)

Suppose we want to find a precondition P such that

$$\{P\} \text{ while } (n > 0) \text{ do } n := n-1 \{n = 0\}$$

1. $\{n - 1 \geq 0\} \text{ n := n-1 } \{n \geq 0\}$ (Assignment rule)
2. $\{n \geq 0 \wedge n > 0\} \text{ n := n-1 } \{n \geq 0\}$ (1, Precond. Equiv.)
3. $\{n \geq 0\} \text{ while } (n > 0) \text{ do } n := n-1 \{n \geq 0 \wedge \neg(n > 0)\}$ (2, While rule)

Proof rule for While Loops

While rule:

$$\frac{\{I \wedge B\} \subseteq \{I\}}{\{I\} \text{ while } B \text{ do } C \subseteq \{I \wedge \neg B\}}$$

Example (cont.)

Suppose we want to find a precondition P such that

$$\{P\} \text{ while } (n > 0) \text{ do } n := n-1 \{n = 0\}$$

1. $\{n - 1 \geq 0\} \text{ n := n-1 } \{n \geq 0\}$ (Assignment rule)
2. $\{n \geq 0 \wedge n > 0\} \text{ n := n-1 } \{n \geq 0\}$ (1, Precond. Equiv.)
3. $\{n \geq 0\} \text{ while } (n > 0) \text{ do } n := n-1 \{n \geq 0 \wedge \neg(n > 0)\}$ (2, While rule)
4. $\{n \geq 0\} \text{ while } (n > 0) \text{ do } n := n-1 \{n = 0\}$ (3, Postcond. Equiv.)

Proof rules for Hoare logic

The assignment axiom:

$$\{Q[x/\mathbb{E}]\} \ x := \mathbb{E} \ {Q}$$

Precondition Strengthening rule:

$$\frac{P_s \rightarrow P_w \quad \{P_w\} \subset \{Q\}}{\{P_s\} \subset \{Q\}}$$

Postcondition Weakening rule:

$$\frac{\{P\} \subset \{Q_s\} \quad Q_s \rightarrow Q_w}{\{P\} \subset \{Q_w\}}$$

Sequencing rule:

$$\frac{\{P\} C_1 \{Q\} \quad \{Q\} C_2 \{R\}}{\{P\} C_1; C_2 \{R\}}$$

Conditional rule:

$$\frac{\{P \wedge \mathbb{B}\} C_1 \{Q\} \quad \{P \wedge \neg \mathbb{B}\} C_2 \{Q\}}{\{P\} \text{ if } \mathbb{B} \text{ then } C_1 \text{ else } C_2 \{Q\}}$$

While rule:

$$\frac{\begin{array}{c} \{I \wedge \mathbb{B}\} \subset \{I\} \\ \{I\} \text{ while } \mathbb{B} \text{ do } C \{I \wedge \neg \mathbb{B}\} \end{array}}{\{I \wedge \mathbb{B}\} \subset \{I\}}$$

A simple program

Example

The sum of the first n odd numbers is n^2 .

Program with specification:

$$\{\top\}$$

i := 0;

s := 0;

while (i \neq n) do

i := i+1;

s := s+(2*i-1)

$$\{s = n^2\}$$

Goal: prove $\{\top\}$ Program $\{s = n^2\}$

A simple program

Example (cont.)

Let us check some examples:

- $1 = 1 = 1^2$
- $1 + 3 = 4 = 2^2$
- $1 + 3 + 5 = 9 = 3^2$
- $1 + 3 + 5 + 7 = 16 = 4^2$

It looks OK. Let us see if we can prove it!

Goal: prove $\{\top\}$ Program $\{s = n^2\}$

A simple program

Example (cont.)

First we need a loop invariant I .

$$\frac{\{I \wedge B\} \subset \{I\}}{\{I\} \text{ while } B \text{ do } C \{I \wedge \neg B\}}$$

```
while (i ≠ n) do
    i := i+1;
    s := s+(2*i-1)
{s = n2}
```

From the while rule, we want $I \wedge (i = n) \rightarrow (s = n^2)$ in order to be able to apply Postcond. Weak.

In the loop body, each time, i increments and s moves on the next square number.

A simple program

Example (cont.)

First we need a loop invariant I .

$$\frac{\{I \wedge \mathbb{B}\} \subset \{I\}}{\{I\} \text{ while } \mathbb{B} \text{ do } \mathbb{C} \{I \wedge \neg \mathbb{B}\}}$$

```
while (i ≠ n) do
    i := i+1;
    s := s+(2*i-1)
{s = n2}
```

From the while rule, we want $I \wedge (i = n) \rightarrow (s = n^2)$ in order to be able to apply Postcond. Weak.

In the loop body, each time, i increments and s moves on the next square number.

Loop invariant $I \equiv (s = i^2)$ seems plausible.

A simple program

Example (cont.)

Check $I \equiv (s = i^2)$ is an invariant: prove $\{I \wedge (i \neq n)\} \mathbb{C} \{I\}$

$$\frac{\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1 \{Q\} \quad \{Q\} \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}}{\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1; \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}}$$

A simple program

Example (cont.)

Check $I \equiv (s = i^2)$ is an invariant: prove $\{I \wedge (i \neq n)\} \mathbb{C} \{I\}$

$$\frac{\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1 \{Q\} \quad \{Q\} \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}}{\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1; \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}}$$

1. $\{Q\} \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}$
- 2.
3. $\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1 \{Q\}$
4. $\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1; \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}$ (1,3, Seq.)

A simple program

Example (cont.)

Check $I \equiv (s = i^2)$ is an invariant: prove $\{I \wedge (i \neq n)\} \mathbb{C} \{I\}$

$$\frac{\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i + 1} \{Q\} \quad \{Q\} \mathbf{s} := \mathbf{s + (2 * i - 1)} \{s = i^2\}}{\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i + 1; s := s + (2 * i - 1)} \{s = i^2\}}$$

Q is $\{s + (2 * i - 1) = i^2\}$

1. $\{s + (2 * i - 1) = i^2\} \mathbf{s := s + (2 * i - 1)} \{s = i^2\}$ (Assignment)
- 2.
3. $\{s = i^2 \wedge i \neq n\} \mathbf{i := i + 1} \{Q\}$
4. $\{s = i^2 \wedge i \neq n\} \mathbf{i := i + 1; s := s + (2 * i - 1)} \{s = i^2\}$ (1,3, Seq.)

A simple program

Example (cont.)

Check $I \equiv (s = i^2)$ is an invariant: prove $\{I \wedge (i \neq n)\} \mathbb{C} \{I\}$

$$\frac{\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i + 1} \{Q\} \quad \{Q\} \mathbf{s} := \mathbf{s + (2 * i - 1)} \{s = i^2\}}{\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i + 1; s := s + (2 * i - 1)} \{s = i^2\}}$$

Q is $\{s + (2 * i - 1) = i^2\}$

1. $\{s + (2 * i - 1) = i^2\} \mathbf{s := s + (2 * i - 1)} \{s = i^2\}$ (Assignment)
- 2.
3. $\{s = i^2 \wedge i \neq n\} \mathbf{i := i + 1} \{s + (2 * i - 1) = i^2\}$
4. $\{s = i^2 \wedge i \neq n\} \mathbf{i := i + 1; s := s + (2 * i - 1)} \{s = i^2\}$ (1,3, Seq.)

A simple program

Example (cont.)

Check $I \equiv (s = i^2)$ is an invariant: prove $\{I \wedge (i \neq n)\} \mathbb{C} \{I\}$

$$\frac{\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1 \{Q\} \quad \{Q\} \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}}{\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1; \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}}$$

Q is $\{s + (2 * i - 1) = i^2\}$

1. $\{s + (2 * i - 1) = i^2\} \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}$ (Assignment)
2. $\{s + (2 * (i + 1) - 1) = (i + 1)^2\} \mathbf{i} := \mathbf{i} + 1 \{s + (2 * i - 1) = i^2\}$ (Assignment)
3. $\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1 \{s + (2 * i - 1) = i^2\}$
4. $\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1; \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}$ (1,3, Seq.)

A simple program

Example (cont.)

Check $I \equiv (s = i^2)$ is an invariant: prove $\{I \wedge (i \neq n)\} \mathbb{C} \{I\}$

$$\frac{\{s = i^2 \wedge i \neq n\} \mathbf{i := i + 1} \{Q\} \quad \{Q\} \mathbf{s := s + (2 * i - 1)} \{s = i^2\}}{\{s = i^2 \wedge i \neq n\} \mathbf{i := i + 1; s := s + (2 * i - 1)} \{s = i^2\}}$$

Q is $\{s + (2 * i - 1) = i^2\}$

1. $\{s + (2 * i - 1) = i^2\} \mathbf{s := s + (2 * i - 1)} \{s = i^2\}$ (Assignment)
2. $\{s + (2 * (i + 1) - 1) = (i + 1)^2\} \mathbf{i := i + 1} \{s + (2 * i - 1) = i^2\}$ (Assignment)
3. $\{s = i^2 \wedge i \neq n\} \mathbf{i := i + 1} \{s + (2 * i - 1) = i^2\}$ (2, Strength. Precond.)
4. $\{s = i^2 \wedge i \neq n\} \mathbf{i := i + 1; s := s + (2 * i - 1)} \{s = i^2\}$ (1,3, Seq.)

A simple program

Example (cont.)

Check $I \equiv (s = i^2)$ is an invariant: prove $\{I \wedge (i \neq n)\} \mathbb{C} \{I\}$

$$\frac{\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1 \{Q\} \quad \{Q\} \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}}{\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1; \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}}$$

Q is $\{s + (2 * i - 1) = i^2\}$

1. $\{s + (2 * i - 1) = i^2\} \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}$ (Assignment)
2. $\{s + (2 * (i + 1) - 1) = (i + 1)^2\} \mathbf{i} := \mathbf{i} + 1 \{s + (2 * i - 1) = i^2\}$ (Assignment)
3. $\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1 \{s + (2 * i - 1) = i^2\}$ (2, Strength. Precond.)
4. $\{s = i^2 \wedge i \neq n\} \mathbf{i} := \mathbf{i} + 1; \mathbf{s} := \mathbf{s} + (2 * \mathbf{i} - 1) \{s = i^2\}$ (1,3, Seq.)

So far, so good.

A simple program

Example (cont.)

Completing the proof of $\{\top\}$ Program $\{s = n^2\}$

1. We have

$$\{(s = i^2) \wedge (i \neq n)\} \text{ i := i+1; s := s+(2*i-1)} \{s = i^2\}$$

A simple program

Example (cont.)

Completing the proof of $\{\top\}$ Program $\{s = n^2\}$

1. We have

$$\{(s = i^2) \wedge (i \neq n)\} \text{ i := i+1; s := s+(2*i-1)} \{s = i^2\}$$

2. Apply the While rule and postcond. equiv. $(s = i^2) \wedge (i = n) \leftrightarrow s = n^2$

$$\{s = i^2\} \text{ while } \dots \text{ s:=s+(2*i-1)} \{s = n^2\}$$

A simple program

Example (cont.)

Completing the proof of $\{\top\}$ Program $\{s = n^2\}$

1. We have

$$\{(s = i^2) \wedge (i \neq n)\} \text{ i } := \text{i+1; s } := \text{s+(2*i-1)} \{s = i^2\}$$

2. Apply the While rule and postcond. equiv. $(s = i^2) \wedge (i = n) \leftrightarrow s = n^2$

$$\{s = i^2\} \text{ while } \dots \text{ s:=s+(2*i-1)} \{s = n^2\}$$

3. Check that the initialization establishes the invariant:

$$\frac{\{0 = 0^2\} \text{i} := 0 \{0 = i^2\} \quad \{0 = i^2\} \text{s} := 0 \{s = i^2\}}{\{0 = 0^2\} \text{i} := 0; \text{s} := 0 \{s = i^2\}}$$

A simple program

Example (cont.)

Completing the proof of $\{\top\}$ Program $\{s = n^2\}$

1. We have

$$\{(s = i^2) \wedge (i \neq n)\} \quad i := i+1; \quad s := s + (2*i - 1) \quad \{s = i^2\}$$

2. Apply the While rule and postcond. equiv. $(s = i^2) \wedge (i = n) \leftrightarrow s = n^2$

$$\{s = i^2\} \text{ while } \dots \quad s := s + (2*i - 1) \quad \{s = n^2\}$$

3. Check that the initialization establishes the invariant:

$$\frac{\{0 = 0^2\} i := 0 \quad \{0 = i^2\} \quad \{0 = i^2\} s := 0 \quad \{s = i^2\}}{\{0 = 0^2\} i := 0; s := 0 \quad \{s = i^2\}}$$

4. $(0 = 0^2) \leftrightarrow \top$, so putting it all together with Sequencing we have

$$\{\top\} \quad i := 0; \quad s := 0; \quad \text{while } (i \neq n) \text{ do } S \quad \{s = n^2\}$$

Weakest Precondition calculus

Weakest precondition calculus

- Edsger W. Dijkstra 1975: introduced another technique for proving properties of imperative programs.
- Weakest Precondition calculus (WP)



Hoare logic presents logic problems:

- Given a precondition P , some code \mathbb{C} , and postcondition Q , is the Hoare triple $\{P\} \mathbb{C} \{Q\}$ true?

Weakest precondition calculus

- Edsger W. Dijkstra 1975: introduced another technique for proving properties of imperative programs.
- Weakest Precondition calculus (WP)



Hoare logic presents **logic** problems:

- Given a precondition P , some code \mathbb{C} , and postcondition Q , is the Hoare triple $\{P\} \mathbb{C} \{Q\}$ true?

WP is about evaluating a **function**:

- Given some code \mathbb{C} and postcondition Q , find the **unique** P which is the weakest precondition such that Q holds after \mathbb{C} .

Weakest precondition calculus

If C is a code fragment and Q is an assertion about states, then the weakest precondition for C with respect to Q is an assertion that is true for precisely those initial states from which:

- C must terminate, and
- executing C must produce a state satisfying Q .

Weakest precondition calculus

If \mathbb{C} is a code fragment and Q is an assertion about states, then the weakest precondition for \mathbb{C} with respect to Q is an assertion that is true for precisely those initial states from which:

- \mathbb{C} must terminate, and
- executing \mathbb{C} must produce a state satisfying Q .

The weakest precondition P is a function of \mathbb{C} and Q :

$$P = wp(\mathbb{C}, Q)$$

- The function wp is sometimes called predicate transformer.
- The calculus WP is sometimes called Predicate Transformer Semantics.

Relationship with Hoare Logic

Hoare Logic is **relational**:

- For each Q , there are **many** P such that $\{P\} \mathbb{C} \{Q\}$.
- For each P , there are **many** Q such that $\{P\} \mathbb{C} \{Q\}$.

Relationship with Hoare Logic

Hoare Logic is **relational**:

- For each Q , there are **many** P such that $\{P\} \mathbb{C} \{Q\}$.
- For each P , there are **many** Q such that $\{P\} \mathbb{C} \{Q\}$.

WP is **functional**:

- For each Q , there is **exactly one** assertion $wp(\mathbb{C}, Q)$.

WP **respects** Hoare logic: $\{wp(\mathbb{C}, Q)\} \mathbb{C} \{Q\}$ is true.

Relationship with Hoare Logic

Hoare Logic is **relational**:

- For each Q , there are **many** P such that $\{P\} \mathbb{C} \{Q\}$.
- For each P , there are **many** Q such that $\{P\} \mathbb{C} \{Q\}$.

WP is **functional**:

- For each Q , there is **exactly one** assertion $wp(\mathbb{C}, Q)$.

WP **respects** Hoare logic: $\{wp(\mathbb{C}, Q)\} \mathbb{C} \{Q\}$ is true.

Hoare logic is about **partial correctness** (we don't care about termination).

Relationship with Hoare Logic

Hoare Logic is **relational**:

- For each Q , there are **many** P such that $\{P\} \mathbb{C} \{Q\}$.
- For each P , there are **many** Q such that $\{P\} \mathbb{C} \{Q\}$.

WP is **functional**:

- For each Q , there is **exactly one** assertion $wp(\mathbb{C}, Q)$.

WP **respects** Hoare logic: $\{wp(\mathbb{C}, Q)\} \mathbb{C} \{Q\}$ is true.

Hoare logic is about **partial correctness** (we don't care about termination).

WP is about **total correctness** (we do care about termination).

$$\text{Total correctness} = \text{Termination} + \text{Partial correctness}$$

Example

Consider the code `x := x+1` and postcondition $(x > 0)$.

Example

Consider the code `x := x+1` and postcondition $(x > 0)$.

- One valid precondition is $(x > 0)$, so in Hoare logic the following is true

$$\{x > 0\} \ x := x+1 \ {x > 0}$$

Example

Consider the code `x := x+1` and postcondition $(x > 0)$.

- One valid precondition is $(x > 0)$, so in Hoare logic the following is true

$$\{x > 0\} \ x := x+1 \ {x > 0}$$

- Another valid precondition is $(x > -1)$, so

$$\{x > -1\} \ x := x+1 \ {x > 0}$$

Example

Consider the code `x := x+1` and postcondition $(x > 0)$.

- One valid precondition is $(x > 0)$, so in Hoare logic the following is true

$$\{x > 0\} \ x \ := \ x+1 \ \{x > 0\}$$

- Another valid precondition is $(x > -1)$, so

$$\{x > -1\} \ x \ := \ x+1 \ \{x > 0\}$$

- $(x > -1)$ is **weaker** than $(x > 0)$ (since $(x > 0) \rightarrow (x > -1)$)

Example

Consider the code `x := x+1` and postcondition $(x > 0)$.

- One valid precondition is $(x > 0)$, so in Hoare logic the following is true

$$\{x > 0\} \ x := x+1 \ {x > 0}$$

- Another valid precondition is $(x > -1)$, so

$$\{x > -1\} \ x := x+1 \ {x > 0}$$

- $(x > -1)$ is **weaker** than $(x > 0)$ (since $(x > 0) \rightarrow (x > -1)$)
- In fact $(x > -1)$ is the **weakest precondition**

$$wp(x := x+1, x > 0) \equiv (x > -1)$$

Weakest precondition for Assignment (Rule 1/4)

The Assignment axiom of Hoare Logic is designed to give the "best" (i.e., the weakest) precondition:

$$\{Q[x/\mathbb{E}]\} \ x := \mathbb{E} \ {Q}$$

Weakest precondition for Assignment (Rule 1/4)

The Assignment axiom of Hoare Logic is designed to give the "best" (i.e., the weakest) precondition:

$$\{Q[x/\mathbb{E}]\} \ x := \mathbb{E} \ {Q}$$

Therefore **the rule for Assignment** in the weakest precondition calculus corresponds closely:

$$wp(x := \mathbb{E}, Q) \equiv Q[x/\mathbb{E}]$$

(Q is an assertion involving a variable x and $Q[x/\mathbb{E}]$ indicates the same assertion with all occurrences of x replaced by the expression \mathbb{E})

Weakest precondition for Assignment

The rule for Assignment:

$$wp(x := E, Q) \equiv Q[x/E]$$

Example

$$wp(x := y+3, x > 3)$$

Weakest precondition for Assignment

The rule for Assignment:

$$wp(x := E, Q) \equiv Q[x/E]$$

Example

$$\begin{aligned} wp(x := y+3, x > 3) &\equiv y + 3 > 3 && \text{(substitute } y + 3 \text{ for } x\text{)} \\ &\equiv y > 0 && \text{(simplify)} \end{aligned}$$

Weakest precondition for Assignment

The rule for Assignment:

$$wp(x := \mathbb{E}, Q) \equiv Q[x/\mathbb{E}]$$

Example

$$\begin{aligned} wp(x := y+3, x > 3) &\equiv y + 3 > 3 && \text{(substitute } y + 3 \text{ for } x\text{)} \\ &\equiv y > 0 && \text{(simplify)} \end{aligned}$$

$$wp(n := n+1, n > 5)$$

Weakest precondition for Assignment

The rule for Assignment:

$$wp(x := \mathbb{E}, Q) \equiv Q[x/\mathbb{E}]$$

Example

$$\begin{aligned} wp(x := y+3, x > 3) &\equiv y + 3 > 3 && (\text{substitute } y + 3 \text{ for } x) \\ &\equiv y > 0 && (\text{simplify}) \end{aligned}$$

$$\begin{aligned} wp(n := n+1, n > 5) &\equiv n + 1 > 5 && (\text{substitute } n + 1 \text{ for } n) \\ &\equiv n > 4 && (\text{simplify}) \end{aligned}$$

Weakest precondition for Sequences (Rule 2/4)

The rule for sequencing compose the effect of the consecutive statements:

$$wp(\mathbb{C}_1; \mathbb{C}_2, Q) \equiv wp(\mathbb{C}_1, wp(\mathbb{C}_2, Q))$$

Weakest precondition for Sequences (Rule 2/4)

The rule for sequencing compose the effect of the consecutive statements:

$$wp(\mathbb{C}_1; \mathbb{C}_2, Q) \equiv wp(\mathbb{C}_1, wp(\mathbb{C}_2, Q))$$

Example

$$wp(x := x+2; y := y-2, x + y = 0)$$

\equiv

Weakest precondition for Sequences (Rule 2/4)

The rule for sequencing compose the effect of the consecutive statements:

$$wp(C_1; C_2, Q) \equiv wp(C_1, wp(C_2, Q))$$

Example

$$\begin{aligned} & wp(x := x+2; y := y-2, x + y = 0) \\ \equiv & wp(x := x+2, wp(y := y-2, x + y = 0)) \\ \equiv & wp(x := x+2, x + (y - 2) = 0) \\ \equiv & (x + 2) + (y - 2) = 0 \\ \equiv & x + y = 0 \end{aligned}$$

Weakest precondition for Conditionals (Rule 3a/4)

$$wp(\text{if } \mathbb{B} \text{ then } \mathbb{C}_1 \text{ else } \mathbb{C}_2, Q) \equiv (\mathbb{B} \rightarrow wp(\mathbb{C}_1, Q)) \wedge (\neg\mathbb{B} \rightarrow wp(\mathbb{C}_2, Q))$$

Weakest precondition for Conditionals (Rule 3a/4)

$$wp(\text{if } \mathbb{B} \text{ then } \mathbb{C}_1 \text{ else } \mathbb{C}_2, Q) \equiv (\mathbb{B} \rightarrow wp(\mathbb{C}_1, Q)) \wedge (\neg\mathbb{B} \rightarrow wp(\mathbb{C}_2, Q))$$

Example

$$wp(\text{if } x > 2 \text{ then } y := 1 \text{ else } y := -1, y > 0)$$

\equiv

Weakest precondition for Conditionals (Rule 3a/4)

$$wp(\text{if } \mathbb{B} \text{ then } \mathbb{C}_1 \text{ else } \mathbb{C}_2, Q) \equiv (\mathbb{B} \rightarrow wp(\mathbb{C}_1, Q)) \wedge (\neg \mathbb{B} \rightarrow wp(\mathbb{C}_2, Q))$$

Example

$$\begin{aligned} & wp(\text{if } x > 2 \text{ then } y := 1 \text{ else } y := -1, y > 0) \\ \equiv & ((x > 2) \rightarrow wp(y := 1, y > 0)) \wedge (\neg(x > 2) \rightarrow wp(y := -1, y > 0)) \\ \equiv & ((x > 2) \rightarrow (1 > 0)) \wedge (\neg(x > 2) \rightarrow (-1 > 0)) \\ \equiv & ((x > 2) \rightarrow \top) \wedge (\neg(x > 2) \rightarrow \perp) \\ \equiv & x > 2 \end{aligned}$$

Alternative rule for Conditionals (Rule 3b/4)

It is often easier to deal with disjunctions and conjunctions than implications, so the following **equivalent** rule for conditionals is usually more convenient.

$$wp(\text{if } \mathbb{B} \text{ then } \mathbb{C}_1 \text{ else } \mathbb{C}_2, Q) \equiv (\mathbb{B} \wedge wp(\mathbb{C}_1, Q)) \vee (\neg \mathbb{B} \wedge wp(\mathbb{C}_2, Q))$$

Alternative rule for Conditionals (Rule 3b/4)

It is often easier to deal with disjunctions and conjunctions than implications, so the following **equivalent** rule for conditionals is usually more convenient.

$$wp(\text{if } \mathbb{B} \text{ then } \mathbb{C}_1 \text{ else } \mathbb{C}_2, Q) \equiv (\mathbb{B} \wedge wp(\mathbb{C}_1, Q)) \vee (\neg \mathbb{B} \wedge wp(\mathbb{C}_2, Q))$$

Example

$$\begin{aligned} & wp(\text{if } x > 2 \text{ then } y := 1 \text{ else } y := -1, y > 0) \\ \equiv & ((x > 2) \wedge wp(y := 1, y > 0)) \vee (\neg(x > 2) \wedge wp(y := -1, y > 0)) \\ \equiv & ((x > 2) \wedge (1 > 0)) \vee (\neg(x > 2) \wedge (-1 > 0)) \\ \equiv & ((x > 2) \wedge \top) \vee (\neg(x > 2) \wedge \perp) \\ \equiv & (x > 2) \vee \perp \\ \equiv & (x > 2) \end{aligned}$$

Proof rule for Conditionals

Exercise:

How would you derive a rule for a conditional statement without `else`?

`if B then C`

Quiz time!

<https://www.questionpro.com/t/AT4NiZrXs0>

References

- Lecture Notes on "Formal Methods for Software Engineering", Australian National University, Rajeev Goré.
- Mike Gordon, "Specification and Verification I", chapters 1 and 2.
- Michael Huth, Mark Ryan, "Logic in Computer Science: Modeling and Reasoning about Systems", 2nd edition, Cambridge University Press, 2004.
- Krzysztof R. Apt, Frank S. de Boer, Ernst-Rüdiger Olderog, "Verification of Sequential and Concurrent Programs", 3rd edition, Springer.

C04 – Weakest Precondition calculus & Separation Logic

Program Verification

FMI · Denisa Diaconescu · Spring 2022

Overview

Weakest Precondition calculus (cont.)

Separation Logic

Weakest Precondition calculus (cont.)

Weakest precondition calculus (WP)

WP is about evaluating a **function**:

- Given some code C and postcondition Q , find the **unique** P which is the weakest precondition such that Q holds after C .
- $P = wp(C, Q)$
- WP **respects** Hoare logic: $\{wp(C, Q)\} C \{Q\}$ is true in Hoare logic.

WP is about **total correctness**.

$$\text{Total correctness} = \text{Termination} + \text{Partial correctness}$$

Weakest precondition rules

Rule for Assignment $\boxed{wp(x := E, Q) \equiv Q[x/E]}$

(Q is an assertion involving a variable x and $Q[x/E]$ indicates the same assertion with all occurrences of x replaced by the expression E)

Rule for sequencing $\boxed{wp(C_1; C_2, Q) \equiv wp(C_1, wp(C_2, Q))}$

Rules for conditionals (equivalent)

$\boxed{wp(\text{if } B \text{ then } C_1 \text{ else } C_2, Q) \equiv (B \rightarrow wp(C_1, Q)) \wedge (\neg B \rightarrow wp(C_2, Q))}$

$\boxed{wp(\text{if } B \text{ then } C_1 \text{ else } C_2, Q) \equiv (B \wedge wp(C_1, Q)) \vee (\neg B \wedge wp(C_2, Q))}$

Loops

Suppose we have a while loop and some postcondition Q .

The precondition P that we seek is the weakest that:

- establishes Q
- guarantees termination

Loops

Suppose we have a while loop and some postcondition Q .

The precondition P that we seek is the weakest that:

- establishes Q
- guarantees termination

We can take hints for the corresponding rule for Hoare Logic. That is, think in terms of **loop invariants**.

Loops

Suppose we have a while loop and some postcondition Q .

The precondition P that we seek is the weakest that:

- establishes Q
- guarantees termination

We can take hints for the corresponding rule for Hoare Logic. That is, think in terms of **loop invariants**.

But **termination is a bigger problem!**

An undecidable problem

Determining if a program terminates or not on a given input is an **undecidable problem!**

An undecidable problem

Determining if a program terminates or not on a given input is an **undecidable problem!**

So there's no algorithm to compute $wp(\text{while } \mathbb{B} \text{ do } \mathbb{C}, Q)$ in all cases.

But that doesn't mean there are no techniques to tackle this problem that at least work some of the time!

Guaranteeing termination

The precondition P we seek is the weakest that establishes Q and guarantees termination.

How can a loop terminate?

Guaranteeing termination

The precondition P we seek is the weakest that establishes Q and guarantees termination.

How can a loop terminate?

- If the loop body is never entered, then the postcondition Q must already be true and the loop condition \mathbb{B} false.
 - We will call this precondition P_0 .
 - $P_0 \equiv \neg\mathbb{B} \wedge Q$ i.e, $\{\neg\mathbb{B} \wedge Q\}$ do nothing $\{Q\}$

Guaranteeing termination

The precondition P we seek is the weakest that establishes Q and guarantees termination.

How can a loop terminate?

- If the loop body is never entered, then the postcondition Q must already be true and the loop condition \mathbb{B} false.
 - We will call this precondition P_0 .
 - $P_0 \equiv \neg \mathbb{B} \wedge Q$ i.e., $\{\neg \mathbb{B} \wedge Q\}$ do nothing $\{Q\}$
- Suppose the loop body executes exactly once. In this case:
 - \mathbb{B} must be true initially
 - after the first time through the loop body, P_0 must become true (so that the loop terminates next time through).
 - $P_1 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_0)$ i.e., $\{\mathbb{B} \wedge wp(\mathbb{C}, P_0)\} \subseteq \{P_0\}$

Guaranteeing termination

$$P_0 \equiv \neg \mathbb{B} \wedge Q$$

i.e., $\{\neg \mathbb{B} \wedge Q\}$ do nothing $\{Q\}$

$$P_1 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_0)$$

i.e., $\{\mathbb{B} \wedge wp(\mathbb{C}, P_0)\} \subseteq \{P_0\}$

Guaranteeing termination

$$P_0 \equiv \neg \mathbb{B} \wedge Q \quad \text{i.e., } \{\neg \mathbb{B} \wedge Q\} \text{ do nothing } \{Q\}$$

$$P_1 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_0) \quad \text{i.e., } \{\mathbb{B} \wedge wp(\mathbb{C}, P_0)\} \subseteq \{P_0\}$$

$$P_2 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_1) \quad \text{i.e., } \{\mathbb{B} \wedge wp(\mathbb{C}, P_1)\} \subseteq \{P_1\}$$

Guaranteeing termination

$$P_0 \equiv \neg \mathbb{B} \wedge Q \quad \text{i.e., } \{\neg \mathbb{B} \wedge Q\} \text{ do nothing } \{Q\}$$

$$P_1 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_0) \quad \text{i.e., } \{\mathbb{B} \wedge wp(\mathbb{C}, P_0)\} \subseteq \{P_0\}$$

$$P_2 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_1) \quad \text{i.e., } \{\mathbb{B} \wedge wp(\mathbb{C}, P_1)\} \subseteq \{P_1\}$$

$$P_3 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_2) \quad \text{i.e., } \{\mathbb{B} \wedge wp(\mathbb{C}, P_2)\} \subseteq \{P_2\}$$

...

Guaranteeing termination

$$P_0 \equiv \neg \mathbb{B} \wedge Q \quad \text{i.e., } \{\neg \mathbb{B} \wedge Q\} \text{ do nothing } \{Q\}$$

$$P_1 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_0) \quad \text{i.e., } \{\mathbb{B} \wedge wp(\mathbb{C}, P_0)\} \subseteq \{P_0\}$$

$$P_2 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_1) \quad \text{i.e., } \{\mathbb{B} \wedge wp(\mathbb{C}, P_1)\} \subseteq \{P_1\}$$

$$P_3 \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_2) \quad \text{i.e., } \{\mathbb{B} \wedge wp(\mathbb{C}, P_2)\} \subseteq \{P_2\}$$

...

P_k – the weakest precondition under which the loop terminates with postcondition Q after exactly k iterations.

We can capture the definition of P_k with an inductive definition.

An inductive definition

$$P_0 \equiv \neg \mathbb{B} \wedge Q$$

$$P_{k+1} \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k)$$

An inductive definition

$$P_0 \equiv \neg B \wedge Q$$

$$P_{k+1} \equiv B \wedge wp(C, P_k)$$

If any of the P_k is true in the initial state, then we are guaranteed that the loop will terminate and establish the postcondition Q ,

i.e. $\{P_0 \vee P_1 \vee \dots\}$ while B do C $\{Q\}$ is true.

The weakest precondition for while loops (rule 4/4)

$$wp(\text{while } \mathbb{B} \text{ do } \mathbb{C}, Q) \equiv \exists k \ (k \geq 0 \wedge P_k)$$

where P_k is defined inductively:

$$P_0 \equiv \neg \mathbb{B} \wedge Q$$

$$P_{k+1} \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k)$$

The weakest precondition for while loops (rule 4/4)

$$wp(\text{while } \mathbb{B} \text{ do } \mathbb{C}, Q) \equiv \exists k (k \geq 0 \wedge P_k)$$

where P_k is defined inductively:

$$P_0 \equiv \neg \mathbb{B} \wedge Q$$

$$P_{k+1} \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k)$$

Interpretation:

- P_k is the weakest precondition that ensures that the body \mathbb{C} executes exactly k times and terminates in a state in which the postcondition Q holds.
- If our loop is to terminate with postcondition Q , some P_k must hold before we enter the loop
i.e. $\{P_0 \vee P_1 \vee \dots\} \text{while } \mathbb{B} \text{ do } \mathbb{C} \{Q\}$ is true.

The weakest precondition for while loops

Applying the wp function to a while loop and postcondition will produce an assertion of the form

$$\exists k \ (k \geq 0 \wedge P_k)$$

P_k may be different for each k , so wp may produce an **infinitely long** assertion! Such an assertion is unsuitable for further manipulation.

The weakest precondition for while loops

Applying the wp function to a while loop and postcondition will produce an assertion of the form

$$\exists k \ (k \geq 0 \wedge P_k)$$

P_k may be different for each k , so wp may produce an **infinitely long** assertion! Such an assertion is unsuitable for further manipulation.

We can simplify matters by expressing P_k as a **single, finite formula** that is parameterised by k .

The weakest precondition for while loops

Applying the wp function to a while loop and postcondition will produce an assertion of the form

$$\exists k \ (k \geq 0 \wedge P_k)$$

P_k may be different for each k , so wp may produce an **infinitely long** assertion! Such an assertion is unsuitable for further manipulation.

We can simplify matters by expressing P_k as a **single, finite formula** that is parameterised by k .

Example

If $P_0 \equiv (n = 0)$, $P_1 \equiv (n = 1)$, $P_2 \equiv (n = 2)$, ..., then $P_k \equiv (n = k)$.

The weakest precondition for while loops

Applying the wp function to a while loop and postcondition will produce an assertion of the form

$$\exists k \ (k \geq 0 \wedge P_k)$$

P_k may be different for each k , so wp may produce an **infinitely long** assertion! Such an assertion is unsuitable for further manipulation.

We can simplify matters by expressing P_k as a **single, finite formula** that is parameterised by k .

Example

If $P_0 \equiv (n = 0)$, $P_1 \equiv (n = 1)$, $P_2 \equiv (n = 2)$, ..., then $P_k \equiv (n = k)$.

We must prove correctness of P_k by induction!

The weakest precondition for while loops

$$wp(\text{while } \mathbb{B} \text{ do } \mathbb{C}, Q) \equiv \exists k (k \geq 0 \wedge P_k)$$

$$P_0 \equiv \neg \mathbb{B} \wedge Q$$

$$P_{k+1} \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k)$$

Example

Suppose we want to find:

$$wp(\text{while } n > 0 \text{ do } n := n-1, n = 0)$$

The weakest precondition for while loops

$$wp(\text{while } \mathbb{B} \text{ do } \mathbb{C}, Q) \equiv \exists k (k \geq 0 \wedge P_k)$$

$$P_0 \equiv \neg \mathbb{B} \wedge Q$$

$$P_{k+1} \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k)$$

Example

Suppose we want to find:

$$wp(\text{while } n > 0 \text{ do } n := n-1, n = 0)$$

We start by generating some of the P_k sequence:

The weakest precondition for while loops

$$wp(\text{while } \mathbb{B} \text{ do } \mathbb{C}, Q) \equiv \exists k (k \geq 0 \wedge P_k)$$

$$P_0 \equiv \neg \mathbb{B} \wedge Q$$

$$P_{k+1} \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k)$$

Example

Suppose we want to find:

$$wp(\text{while } n > 0 \text{ do } n := n-1, n = 0)$$

We start by generating some of the P_k sequence:

- $P_0 \equiv \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$ i.e., $\neg \mathbb{B} \wedge Q$

The weakest precondition for while loops

$$wp(\text{while } \mathbb{B} \text{ do } \mathbb{C}, Q) \equiv \exists k \ (k \geq 0 \wedge P_k)$$

$$P_0 \equiv \neg \mathbb{B} \wedge Q$$

$$P_{k+1} \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k)$$

Example

Suppose we want to find:

$$wp(\text{while } n > 0 \text{ do } n := n - 1, n = 0)$$

We start by generating some of the P_k sequence:

- $P_0 \equiv \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$ i.e., $\neg \mathbb{B} \wedge Q$
- $P_1 \equiv (n > 0) \wedge wp(n := n - 1, n = 0) \equiv (n = 1)$ i.e., $\mathbb{B} \wedge wp(\mathbb{C}, P_0)$

The weakest precondition for while loops

$$wp(\text{while } \mathbb{B} \text{ do } \mathbb{C}, Q) \equiv \exists k (k \geq 0 \wedge P_k)$$

$$P_0 \equiv \neg \mathbb{B} \wedge Q$$

$$P_{k+1} \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k)$$

Example

Suppose we want to find:

$$wp(\text{while } n > 0 \text{ do } n := n - 1, n = 0)$$

We start by generating some of the P_k sequence:

- $P_0 \equiv \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$ i.e., $\neg \mathbb{B} \wedge Q$
- $P_1 \equiv (n > 0) \wedge wp(n := n - 1, n = 0) \equiv (n = 1)$ i.e., $\mathbb{B} \wedge wp(\mathbb{C}, P_0)$
- $P_2 \equiv (n > 0) \wedge wp(n := n - 1, n = 1) \equiv (n = 2)$ i.e., $\mathbb{B} \wedge wp(\mathbb{C}, P_1)$

The weakest precondition for while loops

$$wp(\text{while } \mathbb{B} \text{ do } \mathbb{C}, Q) \equiv \exists k (k \geq 0 \wedge P_k)$$

$$P_0 \equiv \neg \mathbb{B} \wedge Q$$

$$P_{k+1} \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_k)$$

Example

Suppose we want to find:

$$wp(\text{while } n > 0 \text{ do } n := n - 1, n = 0)$$

We start by generating some of the P_k sequence:

- $P_0 \equiv \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$ i.e., $\neg \mathbb{B} \wedge Q$
- $P_1 \equiv (n > 0) \wedge wp(n := n - 1, n = 0) \equiv (n = 1)$ i.e., $\mathbb{B} \wedge wp(\mathbb{C}, P_0)$
- $P_2 \equiv (n > 0) \wedge wp(n := n - 1, n = 1) \equiv (n = 2)$ i.e., $\mathbb{B} \wedge wp(\mathbb{C}, P_1)$
- ...

so it looks pretty likely that $P_k \equiv (n = k)$

The weakest precondition for while loops

Example

Suppose we want to find:

$$wp(\text{while } n > 0 \text{ do } n := n-1, n = 0)$$

We prove by induction that $P_k \equiv (n = k)$:

The weakest precondition for while loops

Example

Suppose we want to find:

$$wp(\text{while } n > 0 \text{ do } n := n-1, n = 0)$$

We prove by induction that $P_k \equiv (n = k)$:

- We already checked the base case:

$$P_0 \equiv \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$$

The weakest precondition for while loops

Example

Suppose we want to find:

$$wp(\text{while } n > 0 \text{ do } n := n-1, n = 0)$$

We prove by induction that $P_k \equiv (n = k)$:

- We already checked the **base case**:

$$P_0 \equiv \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$$

- Now for our **induction step**:

We assume $P_i \equiv (n = i)$ for some $i \geq 0$.

Recall that $P_{i+1} \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_i)$.

The weakest precondition for while loops

Example

Suppose we want to find:

$$wp(\text{while } n > 0 \text{ do } n := n-1, n = 0)$$

We prove by induction that $P_k \equiv (n = k)$:

- We already checked the **base case**:

$$P_0 \equiv \neg(n > 0) \wedge (n = 0) \equiv (n = 0)$$

- Now for our **induction step**:

We assume $P_i \equiv (n = i)$ for some $i \geq 0$.

Recall that $P_{i+1} \equiv \mathbb{B} \wedge wp(\mathbb{C}, P_i)$.

$$\begin{aligned} P_{i+1} &\equiv (n > 0) \wedge wp(n := n - 1, n = i) \\ &\equiv (n > 0) \wedge (n - 1 = i) \\ &\equiv (n > 0) \wedge (n = i + 1) \\ &\equiv (n = i + 1) \end{aligned}$$

The weakest precondition for while loops

Example

Therefore we have

$$wp(\text{while } n > 0 \text{ do } n := n-1, n = 0) \equiv \exists k (k \geq 0 \wedge n = k)$$

The weakest precondition for while loops

Example

Therefore we have

$$wp(\text{while } n > 0 \text{ do } n := n-1, n = 0) \equiv \exists k (k \geq 0 \wedge n = k)$$

We can still simplify it further!

Useful trick: $\exists k ((k \geq 0) \wedge P_k) \equiv P_0 \vee P_1 \vee P_2 \vee \dots$

In this example we have $(n = 0) \vee (n = 1) \vee (n = 2) \vee \dots$

The weakest precondition for while loops

Example

Therefore we have

$$wp(\text{while } n > 0 \text{ do } n := n-1, n = 0) \equiv \exists k (k \geq 0 \wedge n = k)$$

We can still simplify it further!

Useful trick: $\exists k ((k \geq 0) \wedge P_k) \equiv P_0 \vee P_1 \vee P_2 \vee \dots$

In this example we have $(n = 0) \vee (n = 1) \vee (n = 2) \vee \dots$

We can compress this infinite disjunction into a finite final result:

$$wp(\text{while } n > 0 \text{ do } n := n-1, n = 0) \equiv (n \geq 0)$$

Total correctness

Example

We want to find

$$wp(\text{while } n \neq 0 \text{ do } n := n-1, n = 0)$$

Total correctness

Example

We want to find

$$wp(\text{while } n \neq 0 \text{ do } n := n-1, n = 0)$$

Step 1 - finding the P_k :

- $P_0 \equiv \neg(n \neq 0) \wedge (n = 0) \equiv (n = 0)$ i.e., $\neg B \wedge Q$
- $P_1 \equiv (n \neq 0) \wedge wp(n := n - 1, n = 0) \equiv (n = 1)$ i.e., $B \wedge wp(C, P_0)$
- ...
- $P_k \equiv (n = k)$ (induction omitted)

Total correctness

Example

Step 2 - finding the weakest precondition:

$$\begin{aligned}\exists k ((k \geq 0) \wedge P_k) &\equiv \exists k ((k \geq 0 \wedge (n = k))) \\ &\equiv (n \geq 0)\end{aligned}$$

Thus,

$$wp(\text{while } n \neq 0 \text{ do } n := n-1, n = 0) \equiv (n \geq 0)$$

Total correctness

Example

Step 2 - finding the weakest precondition:

$$\begin{aligned}\exists k ((k \geq 0) \wedge P_k) &\equiv \exists k ((k \geq 0 \wedge (n = k))) \\ &\equiv (n \geq 0)\end{aligned}$$

Thus,

$$wp(\text{while } n \neq 0 \text{ do } n := n-1, n = 0) \equiv (n \geq 0)$$

This is not really any different than the previous example.

But what is the trap in this while-loop?

Total correctness

Example

Step 2 - finding the weakest precondition:

$$\begin{aligned}\exists k ((k \geq 0) \wedge P_k) &\equiv \exists k ((k \geq 0 \wedge (n = k))) \\ &\equiv (n \geq 0)\end{aligned}$$

Thus,

$$wp(\text{while } n \neq 0 \text{ do } n := n-1, n = 0) \equiv (n \geq 0)$$

This is not really any different than the previous example.

But what is the trap in this while-loop?

We have automatically found that the while-loop will not terminate for initial values of n less than 0.

- Rule for Assignment: $wp(x := E, Q) \equiv Q[x/E]$
(Q is an assertion involving a variable x and $Q[x/E]$ indicates the same assertion with all occurrences of x replaced by the expression E)
- Rule for Sequencing: $wp(C_1; C_2, Q) \equiv wp(C_1, wp(C_2, Q))$
- Rule for Conditionals:
 $wp(\text{if } B \text{ then } C_1 \text{ else } C_2, Q) \equiv (B \rightarrow wp(C_1, Q)) \wedge (\neg B \rightarrow wp(C_2, Q))$
- There is no algorithm to compute $wp(\text{while } B \text{ do } C, Q)$ in all cases!
 - But that doesn't mean there are no techniques to tackle this problem that at least work some of the time!
 - Inductive definition.

Separation Logic

Adding the heap

We extend our programming language with:

- Memory reads: $x := [E]$ *(dereferencing)*
- Memory writes: $[E_1] := E_2$ *(update heap)*
- Memory allocation: $x := \text{cons}(E_1, \dots E_n)$
- Memory deallocation: $\text{dispose } E$

Adding the heap

We extend our programming language with:

- Memory reads: $x := [\mathbb{E}]$ *(dereferencing)*
- Memory writes: $[\mathbb{E}_1] := \mathbb{E}_2$ *(update heap)*
- Memory allocation: $x := \text{cons}(\mathbb{E}_1, \dots, \mathbb{E}_n)$
- Memory deallocation: $\text{dispose } \mathbb{E}$

The **state** is now represented by a pair of type $\text{Store} \times \text{Heap}$, denoted (σ, h) , where

$\sigma \in \text{Store}$, where $\text{Store} \triangleq \text{Var} \rightarrow \text{Val}$

$h \in \text{Heap}$, where $\text{Heap} \triangleq \text{Loc} \rightarrow \text{Val}$

where $\text{Loc} \subseteq \text{Val}$.

Adding the heap

Memory reads: $x := [E]$

- evaluate expression E to get location l
- **fault** if location l is not in the current heap
- otherwise variable x is assigned the content of location l

Adding the heap

Memory reads: $x := [\mathbb{E}]$

- evaluate expression \mathbb{E} to get location l
- **fault** if location l is not in the current heap
- otherwise variable x is assigned the content of location l

Example ($x := [y+1]$)

σ
y 0xAB

h	
0xAB	1
0xAC	2

$x := [y+1]$

σ
y 0xAB
x 2

h	
0xAB	1
0xAC	2

Adding the heap

Memory writes: $[\mathbb{E}_1] := \mathbb{E}_2$

- evaluate expression \mathbb{E}_1 to get location $/$
- **fault** if location $/$ is not in the current heap
- otherwise make the content of location $/$ the value of expression \mathbb{E}_2

Adding the heap

Memory writes: $[\mathbb{E}_1] := \mathbb{E}_2$

- evaluate expression \mathbb{E}_1 to get location $/$
- **fault** if location $/$ is not in the current heap
- otherwise make the content of location $/$ the value of expression \mathbb{E}_2

Example ($[y+1] := 5$)

σ	
y	0xAB

h	
0xAB	1
0xAC	2

$[y+1] := 5$

σ	
y	0xAB

h	
0xAB	1
0xAC	5

Adding the heap

Memory allocation: $x := \text{cons}(\mathbb{E}_1, \dots, \mathbb{E}_n)$

- extend the heap with n consecutive new locations $l, l + 1, \dots, l + n - 1$
- put values of $\mathbb{E}_1, \dots, \mathbb{E}_n$ into locations $l, l + 1, \dots, l + n - 1$ respectively
- extend the stack by assigning x the value l
- never fault

Adding the heap

Memory allocation: $x := \text{cons}(E_1, \dots, E_n)$

- extend the heap with n consecutive new locations $l, l + 1, \dots, l + n - 1$
- put values of E_1, \dots, E_n into locations $l, l + 1, \dots, l + n - 1$ respectively
- extend the stack by assigning x the value l
- never fault

Example ($p := \text{cons}(3, 7)$)

σ	h	$p := \text{cons}(3, 7)$	σ	h										
	<table border="1"><tr><td>:</td><td>:</td></tr></table>	:	:		<table border="1"><tr><td>p</td><td>0xAB</td></tr></table>	p	0xAB	<table border="1"><tr><td>:</td><td>:</td></tr><tr><td>0xAB</td><td>3</td></tr><tr><td>0xAC</td><td>7</td></tr></table>	:	:	0xAB	3	0xAC	7
:	:													
p	0xAB													
:	:													
0xAB	3													
0xAC	7													

Adding the heap

Memory deallocation: `dispose E`

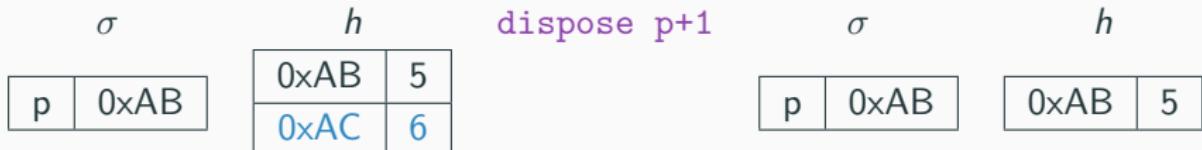
- evaluate expression E to get location $/$
- **fault** if location $/$ is not in the current heap
- otherwise remove location $/$ from the heap

Adding the heap

Memory deallocation: dispose \mathbb{E}

- evaluate expression \mathbb{E} to get location $/$
- **fault** if location $/$ is not in the current heap
- otherwise remove location $/$ from the heap

Example (dispose $p+1$)



Example

x := cons(3,3)

σ

x	0xAB
---	------

h

0xAB	3
0xAC	3

Example

```
x := cons(3,3); y := cons(4,4);
```

σ

x	0xAB
y	0xDD

h

0xAB	3
0xAC	3
0xDD	4
0xDE	4

Example

```
x := cons(3,3) ; y := cons(4,4); [x+1] := y;
```

σ

x	0xAB
y	0xDD

h

0xAB	3
0xAC	0xDD
0xDD	4
0xDE	4

Example

```
x := cons(3,3) ; y := cons(4,4); [x+1] := y; [y+1] := x;
```

σ

x	0xAB
y	0xDD

h

0xAB	3
0xAC	0xDD
0xDD	4
0xDE	0xAB

Example

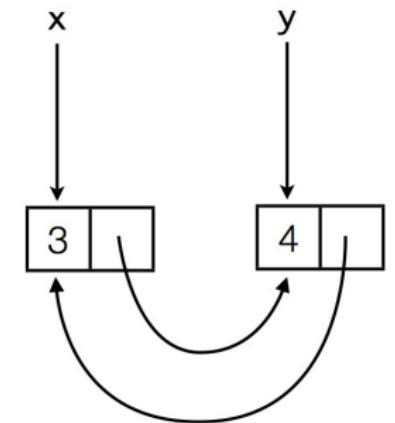
```
x := cons(3,3) ; y := cons(4,4); [x+1] := y; [y+1] := x;
```

σ

x	0xAB
y	0xDD

h

0xAB	3
0xAC	0xDD
0xDD	4
0xDE	0xAB



Why separation logic?

Can you suggest a precondition such that this triple holds?

{???

[y] := 4; [z] := 5;

{($\exists y, z$) $(y \mapsto y \wedge z \mapsto z \wedge y \neq z)$ }

Why separation logic?

Can you suggest a precondition such that this triple holds?

$$\begin{array}{l} \{y \neq z \wedge y \mapsto _ \wedge z \mapsto _ \} \\ \quad [y] := 4; \quad [z] := 5; \\ \{(\exists y, z)(y \mapsto y \wedge z \mapsto z \wedge y \neq z)\} \end{array}$$

We need to assume that the locations pointed by y and z are different (aliasing).

Note that, for example, y is used to denote program variables, while y is used to denote logical variables.

Why separation logic?

And now?

```
[y] := 4; [z] := 5;  
{( $\exists y, z$ ) $(y \mapsto y \wedge z \mapsto z \wedge y \neq z \wedge x \mapsto 3)$ }
```

We need to assume that the locations pointed by y and z are different
(aliasing).

We also need to know when things stay the same.

Why separation logic?

And now?

$$\{y \neq z \wedge x \neq y \wedge x \neq z \wedge y \mapsto _ \wedge z \mapsto _ \wedge x \mapsto 3\}$$

$[y] := 4; [z] := 5;$

$$\{(\exists y, z)(y \mapsto y \wedge z \mapsto z \wedge y \neq z \wedge x \mapsto 3)\}$$

We need to assume that the locations pointed by y and z are different (aliasing).

We also need to know when things stay the same.

Framing

We want a general concept of things not being affected.

$$\frac{\{P\} \textcolor{violet}{C} \{Q\}}{\{x \mapsto 3 \wedge P\} \textcolor{violet}{C} \{Q \wedge x \mapsto 3\}}$$

What are the conditions on $\textcolor{violet}{C}$ and $x \mapsto 3$?

These are very hard to define if reasoning about a heap and aliasing.

Framing

We want a general concept of things not being affected.

$$\frac{\{P\} \textcolor{violet}{C} \{Q\}}{\{x \mapsto 3 \wedge P\} \textcolor{violet}{C} \{Q \wedge x \mapsto 3\}}$$

What are the conditions on C and $x \mapsto 3$?

These are very hard to define if reasoning about a heap and aliasing.

This is where separation logic comes in:

$$\frac{\{P\} \textcolor{violet}{C} \{Q\}}{\{R * P\} \textcolor{violet}{C} \{Q * R\}}$$

The new connective $*$ ("sep" operator) is used to separate the heap.

Reasoning about pointers

For Hoare logic, we assumed no aliasing of variables!

In most real languages we can have multiple names for the same piece of memory.

How is aliasing a problem?

From Hoare logic to separation logic

- Robert W. Floyd 1967: gave some rules to reason about programs.
- Sometimes, our Hoare Logic is called Floyd-Hoare Logic in recognition.
- Many attempts made to extend Floyd-Hoare Logic to handle pointers.
- Only really solved around 2000 by Reynolds, O'Hearn and Yang using a connective $*$ called separating conjunction.
- To make the presentation less scary, we need to first extend Hoare Logic with an axiom due to Floyd.



Hoare Logic

Syntax	Semantics	Calculus
$\neg \wedge \vee \rightarrow \forall \exists$	FOL	N/A
$= + - \geq \leq \dots$	Arithmetics	N/A
<code>:= ; while</code> <code>if then else</code>	State maps variables to values (no pointers)	N/A
$\{P\} \ C \ \{Q\}$	If initial state satisfies P and C terminates then final state satisfies Q	6 Inference Rules

Store assignment axiom of Floyd

Hoare axiom: $\{Q[x/\mathbb{E}]\} \ x := \mathbb{E} \ {Q}$ (backward driven)

Store assignment axiom of Floyd

Hoare axiom: $\{Q[x/\mathbb{E}]\} \ x := \mathbb{E} \ {Q}$ (backward driven)

Floyd axiom: $\{x = v\} \ x := \mathbb{E} \ {x = \mathbb{E}[x/v]}\}$ (forward driven)

- equivalent to Hoare axiom
- v is an auxiliary variable which does not occur in \mathbb{E}
- $\mathbb{E}[x/v]$ means replace all occurrences of x in \mathbb{E} by v

Store assignment axiom of Floyd

Hoare axiom: $\{Q[x/\mathbb{E}]\} \ x := \mathbb{E} \ {Q}$ (backward driven)

Floyd axiom: $\{x = v\} \ x := \mathbb{E} \ {x = \mathbb{E}[x/v]}$ (forward driven)

- equivalent to Hoare axiom
- v is an auxiliary variable which does not occur in \mathbb{E}
- $\mathbb{E}[x/v]$ means replace all occurrences of x in \mathbb{E} by v

Example

Hoare instance: $\{x + 1 = 5\} \ x := x+1 \ {x = 5}$

Store assignment axiom of Floyd

Hoare axiom: $\{Q[x/\mathbb{E}]\} \ x := \mathbb{E} \ {Q}$ (backward driven)

Floyd axiom: $\{x = v\} \ x := \mathbb{E} \ {x = \mathbb{E}[x/v]}$ (forward driven)

- equivalent to Hoare axiom
- v is an auxiliary variable which does not occur in \mathbb{E}
- $\mathbb{E}[x/v]$ means replace all occurrences of x in \mathbb{E} by v

Example

Hoare instance: $\{x + 1 = 5\} \ x := x+1 \ {x = 5}$

Floyd instance: $\{x = v\} \ x := x+1 \ {x = v + 1}$

Store assignment axiom of Floyd

Hoare axiom: $\{Q[x/\mathbb{E}]\} \ x := \mathbb{E} \ \{Q\}$ (backward driven)

Floyd axiom: $\{x = v\} \ x := \mathbb{E} \ \{x = \mathbb{E}[x/v]\}$ (forward driven)

- equivalent to Hoare axiom
- v is an auxiliary variable which does not occur in \mathbb{E}
- $\mathbb{E}[x/v]$ means replace all occurrences of x in \mathbb{E} by v

Example

Hoare instance: $\{x + 1 = 5\} \ x := x+1 \ \{x = 5\}$

Floyd instance: $\{x = v\} \ x := x+1 \ \{x = v + 1\}$

- If we want the postcondition $x = 5$ then instantiate v to be 4
 $\{x = 4\} \ x := x+1 \ \{x = 5\}$

Note: does not solve the problem with pointers!

Quiz time!

<https://www.questionpro.com/t/AT4NiZrf20>

References

- Lecture Notes on "Formal Methods for Software Engineering", Australian National University, Rajeev Goré.
- Mike Gordon, "Specification and Verification I", chapters 1 and 2.
- Michael Huth, Mark Ryan, "Logic in Computer Science: Modeling and Reasoning about Systems", 2nd edition, Cambridge University Press, 2004.
- Krzysztof R. Apt, Frank S. de Boer, Ernst-Rüdiger Olderog, "Verification of Sequential and Concurrent Programs", 3rd edition, Springer.

C05 – Separation Logic & SAT solvers

Program Verification

FMI · Denisa Diaconescu · Spring 2021

Separation Logic

Adding the heap

We extend our programming language with:

- **Heap reads:** $x := [\mathbb{E}]$ *(dereferencing)*
- **Heap writes:** $[\mathbb{E}_1] := \mathbb{E}_2$ *(update heap)*
- **Heap allocation:** $x := \text{cons}(\mathbb{E}_1, \dots, \mathbb{E}_n)$
- **Heap deallocation:** $\text{dispose } \mathbb{E}$

The **state** is now represented by a pair of type $\text{Store} \times \text{Heap}$, denoted (σ, h) , where

$\sigma \in \text{Store}$, where $\text{Store} \triangleq \text{Var} \rightarrow \text{Val}$

$h \in \text{Heap}$, where $\text{Heap} \triangleq \text{Loc} \rightarrow \text{Val}$

where $\text{Loc} \subseteq \text{Val}$.

Note that we consider $\text{dom}(h)$ to always be finite. By this, we ensure that cons commands will never fail.

Evaluating expressions in the store of a state

Strictly speaking, the store gives values to variables only.

But we need a way to say "value of an expression in a store" so we will abuse notation and use $\sigma(\mathbb{E})$ for this as below:

- $\sigma(n) = n$ where n is a number is just its usual value
- $\sigma(x + n) = \sigma(x) + \sigma(n)$ where n is a number and x is a variable

Extra connectives in separation logic

emp empty heap

$E_1 \mapsto E_2$ points to

$P * Q$ separating conjunction

Semantics of separation logic

$$\sigma \triangleq \text{Var} \rightarrow \text{Val}$$

$$h \triangleq \text{Loc} \rightarrow \text{Val}$$

$$(\sigma, h) \models \text{emp} \text{ if } \text{dom}(h) = \emptyset$$

- `emp` is an atomic formula for checking if the heap is empty
- a state (σ, h) makes the formula `emp` true if the heap is empty

Semantics of separation logic

$$\sigma \triangleq \text{Var} \rightarrow \text{Val}$$

$$h \triangleq \text{Loc} \rightarrow \text{Val}$$

$(\sigma, h) \models \mathbb{E}_1 \mapsto \mathbb{E}_2$ if $\text{dom}(h) = \{\sigma(\mathbb{E}_1)\}$ and $h(\sigma(\mathbb{E}_1)) = \sigma(\mathbb{E}_2)$

- a state (σ, h) makes the formula $\mathbb{E}_1 \mapsto \mathbb{E}_2$ true if the heap is a singleton and maps the location $\sigma(\mathbb{E}_1)$ to the value $\sigma(\mathbb{E}_2)$
- $\sigma(\mathbb{E})$ is the value of an expression in a store as explained before

Semantics of separation logic

$$\sigma \triangleq \text{Var} \rightarrow \text{Val}$$

$$h \triangleq \text{Loc} \rightarrow \text{Val}$$

$(\sigma, h) \models P * Q$ if h can be partitioned into two disjoint heaps h_1 and h_2 ,

and $(\sigma, h_1) \models P$ and $(\sigma, h_2) \models Q$

Note that two heaps are disjoint if the intersection of their domains is empty.

Semantics of separation logic

$$\sigma \triangleq \text{Var} \rightarrow \text{Val}$$

$$h \triangleq \text{Loc} \rightarrow \text{Val}$$

$(\sigma, h) \models P * Q$ if h can be partitioned into two disjoint heaps h_1 and h_2 ,

and $(\sigma, h_1) \models P$ and $(\sigma, h_2) \models Q$

Note that two heaps are disjoint if the intersection of their domains is empty.

$(\sigma, h) \models P_1 * P_2 * \dots * P_n$ if h can be partitioned into n disjoint heaps

h_1, h_2, \dots, h_n and $(\sigma, h_i) \models P_i$ for any $i \in \{1, \dots, n\}$

Semantics of separation logic

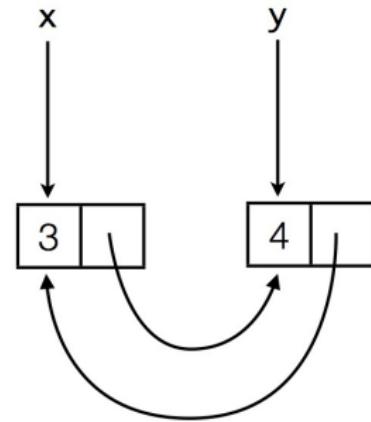
Example

σ

x	0xAB
y	0xDD

h

0xAB	3
0xAC	0xDD
0xDD	4
0xDE	0xAB



Semantics of separation logic

Example

σ

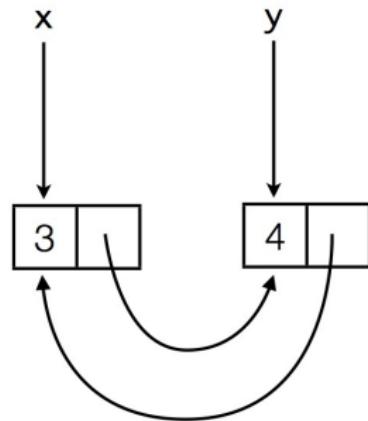
x	0xAB
y	0xDD

h

0xAB	3
0xAC	0xDD
0xDD	4
0xDE	0xAB

Satisfies the statement:

$$(x \mapsto 3) * (x + 1 \mapsto y) * (y \mapsto 4) * (y + 1 \mapsto x)$$



Semantics of separation logic

$(\sigma, h) \models P_1 * P_2 * \dots * P_n$ if h can be partitioned into n distinct heaps h_1, h_2, \dots, h_n
and $(\sigma, h_i) \models P_i$ for any $i \in \{1, \dots, n\}$

Example

σ	h
x 0xAB	3
y 0xDD	0xDD

0xAB	3
0xAC	0xDD
0xDD	4
0xDE	0xAB

We want to show that

$(\sigma, h) \models (x \mapsto 3) * (x + 1 \mapsto y) * (y \mapsto 4) * (y + 1 \mapsto x)$

Semantics of separation logic

$(\sigma, h) \models P_1 * P_2 * \dots * P_n$ if h can be partitioned into n distinct heaps h_1, h_2, \dots, h_n
and $(\sigma, h_i) \models P_i$ for any $i \in \{1, \dots, n\}$

Example

σ

x	0xAB
y	0xDD

h

0xAB	3
0xAC	0xDD
0xDD	4
0xDE	0xAB

We want to show that

$$(\sigma, h) \models (x \mapsto 3) * (x + 1 \mapsto y) * (y \mapsto 4) * (y + 1 \mapsto x)$$

We can partition h into 4 distinct heaps:

σ

x	0xAB
y	0xDD

h_1

0xAB	3
------	---

h_2

0xAC	0xDD
------	------

h_3

0xDD	4
------	---

h_4

0xDE	0xAB
------	------

Semantics of separation logic

$(\sigma, h) \models P_1 * P_2 * \dots * P_n$ if h can be partitioned into n distinct heaps h_1, h_2, \dots, h_n
and $(\sigma, h_i) \models P_i$ for any $i \in \{1, \dots, n\}$

Example

σ

x	0xAB
y	0xDD

h

0xAB	3
0xAC	0xDD
0xDD	4
0xDE	0xAB

We want to show that

$$(\sigma, h) \models (x \mapsto 3) * (x + 1 \mapsto y) * (y \mapsto 4) * (y + 1 \mapsto x)$$

We can partition h into 4 distinct heaps:

σ

x	0xAB
y	0xDD

h_1

0xAB	3
------	---

h_2

0xAC	0xDD
------	------

h_3

0xDD	4
------	---

h_4

0xDE	0xAB
------	------

We must show that

$$(\sigma, h_1) \models x \mapsto 3$$

$$(\sigma, h_2) \models x + 1 \mapsto y$$

$$(\sigma, h_3) \models y \mapsto 4$$

$$(\sigma, h_4) \models y + 1 \mapsto x$$

Semantics of separation logic

$(\sigma, h) \models E_1 \mapsto E_2$ if $\text{dom}(h) = \{\sigma(E_1)\}$ and $h(\sigma(E_1)) = \sigma(E_2)$

Example

σ	h_1	h_2	h_3	h_4												
<table border="1"><tr><td>x</td><td>0xAB</td></tr><tr><td>y</td><td>0xDD</td></tr></table>	x	0xAB	y	0xDD	<table border="1"><tr><td>0xAB</td><td>3</td></tr></table>	0xAB	3	<table border="1"><tr><td>0xAC</td><td>0xDD</td></tr></table>	0xAC	0xDD	<table border="1"><tr><td>0xDD</td><td>4</td></tr></table>	0xDD	4	<table border="1"><tr><td>0xDE</td><td>0xAB</td></tr></table>	0xDE	0xAB
x	0xAB															
y	0xDD															
0xAB	3															
0xAC	0xDD															
0xDD	4															
0xDE	0xAB															

Semantics of separation logic

$(\sigma, h) \models E_1 \mapsto E_2$ if $\text{dom}(h) = \{\sigma(E_1)\}$ and $h(\sigma(E_1)) = \sigma(E_2)$

Example

σ	h_1	h_2	h_3	h_4												
<table border="1"><tr><td>x</td><td>0xAB</td></tr><tr><td>y</td><td>0xDD</td></tr></table>	x	0xAB	y	0xDD	<table border="1"><tr><td>0xAB</td><td>3</td></tr></table>	0xAB	3	<table border="1"><tr><td>0xAC</td><td>0xDD</td></tr></table>	0xAC	0xDD	<table border="1"><tr><td>0xDD</td><td>4</td></tr></table>	0xDD	4	<table border="1"><tr><td>0xDE</td><td>0xAB</td></tr></table>	0xDE	0xAB
x	0xAB															
y	0xDD															
0xAB	3															
0xAC	0xDD															
0xDD	4															
0xDE	0xAB															

$(\sigma, h_1) \models x \mapsto 3$

- $\text{dom}(h_1) = 0xAB = \sigma(x)$
- $h_1(0xAB) = 3$

Semantics of separation logic

$(\sigma, h) \models E_1 \mapsto E_2$ if $\text{dom}(h) = \{\sigma(E_1)\}$ and $h(\sigma(E_1)) = \sigma(E_2)$

Example

σ	h_1	h_2	h_3	h_4												
<table border="1"><tr><td>x</td><td>0xAB</td></tr><tr><td>y</td><td>0xDD</td></tr></table>	x	0xAB	y	0xDD	<table border="1"><tr><td>0xAB</td><td>3</td></tr></table>	0xAB	3	<table border="1"><tr><td>0xAC</td><td>0xDD</td></tr></table>	0xAC	0xDD	<table border="1"><tr><td>0xDD</td><td>4</td></tr></table>	0xDD	4	<table border="1"><tr><td>0xDE</td><td>0xAB</td></tr></table>	0xDE	0xAB
x	0xAB															
y	0xDD															
0xAB	3															
0xAC	0xDD															
0xDD	4															
0xDE	0xAB															

$(\sigma, h_1) \models x \mapsto 3$

- $\text{dom}(h_1) = 0xAB = \sigma(x)$
- $h_1(0xAB) = 3$

$(\sigma, h_2) \models x + 1 \mapsto y$

- $\text{dom}(h_2) = 0xAC = \sigma(x + 1)$
- $h_2(0xAC) = 0xDD = \sigma(y)$

Semantics of separation logic

$(\sigma, h) \models E_1 \mapsto E_2$ if $\text{dom}(h) = \{\sigma(E_1)\}$ and $h(\sigma(E_1)) = \sigma(E_2)$

Example

σ	h_1	h_2	h_3	h_4												
<table border="1"><tr><td>x</td><td>0xAB</td></tr><tr><td>y</td><td>0xDD</td></tr></table>	x	0xAB	y	0xDD	<table border="1"><tr><td>0xAB</td><td>3</td></tr></table>	0xAB	3	<table border="1"><tr><td>0xAC</td><td>0xDD</td></tr></table>	0xAC	0xDD	<table border="1"><tr><td>0xDD</td><td>4</td></tr></table>	0xDD	4	<table border="1"><tr><td>0xDE</td><td>0xAB</td></tr></table>	0xDE	0xAB
x	0xAB															
y	0xDD															
0xAB	3															
0xAC	0xDD															
0xDD	4															
0xDE	0xAB															

$(\sigma, h_1) \models x \mapsto 3$

- $\text{dom}(h_1) = 0xAB = \sigma(x)$
- $h_1(0xAB) = 3$

$(\sigma, h_3) \models y \mapsto 4$

- $\text{dom}(h_3) = 0xDD = \sigma(y)$
- $h_3(0xDD) = 4$

$(\sigma, h_2) \models x + 1 \mapsto y$

- $\text{dom}(h_2) = 0xAC = \sigma(x + 1)$
- $h_2(0xAC) = 0xDD = \sigma(y)$

Semantics of separation logic

$(\sigma, h) \models E_1 \mapsto E_2$ if $\text{dom}(h) = \{\sigma(E_1)\}$ and $h(\sigma(E_1)) = \sigma(E_2)$

Example

σ	h_1	h_2	h_3	h_4
x 0xAB y 0xDD	0xAB 3	0xAC 0xDD	0xDD 4	0xDE 0xAB

$(\sigma, h_1) \models x \mapsto 3$

- $\text{dom}(h_1) = 0xAB = \sigma(x)$
- $h_1(0xAB) = 3$

$(\sigma, h_3) \models y \mapsto 4$

- $\text{dom}(h_3) = 0xDD = \sigma(y)$
- $h_3(0xDD) = 4$

$(\sigma, h_2) \models x + 1 \mapsto y$

- $\text{dom}(h_2) = 0xAC = \sigma(x + 1)$
- $h_2(0xAC) = 0xDD = \sigma(y)$

$(\sigma, h_4) \models y + 1 \mapsto x$

- $\text{dom}(h_4) = 0xDE = \sigma(y + 1)$
- $h_4(0xDE) = 0xAB = \sigma(x)$

Semantics of separation logic

Example

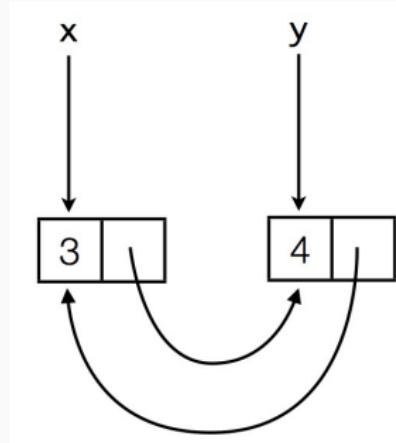
σ

x	0xAB
y	0xDD

h

0xAB	3
0xAC	0xDD
0xDD	4
0xDE	0xAB

Does not satisfy the statement $x \mapsto 3$



Semantics of separation logic

Example

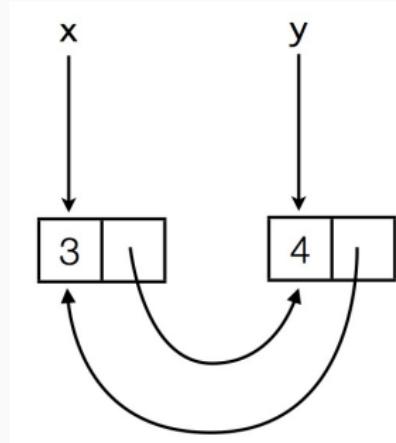
σ

x	0xAB
y	0xDD

h

0xAB	3
0xAC	0xDD
0xDD	4
0xDE	0xAB

Doest not satisfy the statement $x \mapsto 3$



- $(\sigma, h) \models \mathbb{E}_1 \mapsto \mathbb{E}_2$ if $dom(h) = \{\sigma(\mathbb{E}_1)\}$ and $h(\sigma(\mathbb{E}_1)) = \sigma(\mathbb{E}_2)$
- $dom(h) = \{0xAB, 0xAC, 0xDD, 0xDE\}$
- $\sigma(x) = 0xAB$
- $h(\sigma(x)) = 3$

Store assignment axiom for separation logic

Hoare axiom: $\{Q[x/\mathbb{E}]\} \ x := \mathbb{E} \ \{Q\}$

Floyd axiom: $\{x = v\} \ x := \mathbb{E} \ \{x = \mathbb{E}[x/v]\}$

where v is an auxiliary variable which does not occur in \mathbb{E} .

Store assignment axiom for separation logic

Hoare axiom: $\{Q[x/\mathbb{E}]\} \ x := \mathbb{E} \ \{Q\}$

Floyd axiom: $\{x = v\} \ x := \mathbb{E} \ \{x = \mathbb{E}[x/v]\}$

where v is an auxiliary variable which does not occur in \mathbb{E} .

Store assignment axiom for Separation logic:

$\{x = v \wedge \text{emp}\} \ x := \mathbb{E} \ \{x = \mathbb{E}[x/v] \wedge \text{emp}\}$

where v is an auxiliary variable which does not occur in \mathbb{E}

New:

- atomic formula `emp` to say that the "heap is empty"
- we want to track the smallest amount of heap information

Store assignment axiom for separation logic

Store assignment axiom for Separation logic:

$$\{x = v \wedge \text{emp}\} \ x := E \ \{x = E[x/v] \wedge \text{emp}\}$$

where v is an auxiliary variable which does not occur in E

Example

$$\{x = v \wedge \text{emp}\} \ x := 1 \ \{x = 1 \wedge \text{emp}\}$$

If we want the precondition $1 = 1$ (i.e. \top) then instantiate v to x

$$\{x = x \wedge \text{emp}\} \ x := 1 \ \{x = 1 \wedge \text{emp}\}$$

Heap reads axiom

Heap reads axiom:

$$\{x = v_1 \wedge E \mapsto v_2\} \ x := [E] \ \{x = v_2 \wedge E[x/v_1] \mapsto v_2\}$$

where v_1 and v_2 are auxiliary variables which do not occur in E

Heap reads axiom

Heap reads axiom:

$$\{x = v_1 \wedge E \mapsto v_2\} \ x := [E] \ \{x = v_2 \wedge E[x/v_1] \mapsto v_2\}$$

where v_1 and v_2 are auxiliary variables which do not occur in E

Example ($x := [y]$)

σ
y
x

h	
0xAB	1

$x := [y]$

σ
y
x

h	
0xAB	1

Heap reads axiom

Heap reads axiom:

$$\{x = v_1 \wedge E \mapsto v_2\} \ x := [E] \ \{x = v_2 \wedge E[x/v_1] \mapsto v_2\}$$

where v_1 and v_2 are auxiliary variables which do not occur in E

Example ($x := [y]$)

σ	
y	0xAB
x	2

h	
0xAB	1

$x := [y]$

σ	
y	0xAB
x	1

h	
0xAB	1

Heap read axiom instance:

$$\{x = 2 \wedge y \mapsto 1\} \ x := [y] \ \{x = 1 \wedge y \mapsto 1\}$$

Heap writes axiom

Heap writes axiom:

$$\{\mathbb{E}_1 \mapsto -\} [\mathbb{E}_1] := \mathbb{E}_2 \quad \{\mathbb{E}_1 \mapsto \mathbb{E}_2\}$$

where $(\mathbb{E}_1 \mapsto -)$ abbreviates $(\exists z. \mathbb{E}_1 \mapsto z)$ and z does not occur in \mathbb{E}_1

Heap writes axiom

Heap writes axiom:

$$\{\mathbb{E}_1 \mapsto -\} [\mathbb{E}_1] := \mathbb{E}_2 \quad \{\mathbb{E}_1 \mapsto \mathbb{E}_2\}$$

where $(\mathbb{E}_1 \mapsto -)$ abbreviates $(\exists z. \mathbb{E}_1 \mapsto z)$ and z does not occur in \mathbb{E}_1

Heap assignment semantics:

- evaluate expression \mathbb{E}_1 to get location $/$
- **fault** if location $/$ is not in the current heap
- otherwise make the contents of location $/$ the value of expression \mathbb{E}_2

Heap writes axiom

Heap writes axiom:

$$\{\mathbb{E}_1 \mapsto -\} [\mathbb{E}_1] := \mathbb{E}_2 \quad \{\mathbb{E}_1 \mapsto \mathbb{E}_2\}$$

where $(\mathbb{E}_1 \mapsto -)$ abbreviates $(\exists z. \mathbb{E}_1 \mapsto z)$ and z does not occur in \mathbb{E}_1

Heap assignment semantics:

- evaluate expression \mathbb{E}_1 to get location $/$
- **fault** if location $/$ is not in the current heap
- otherwise make the contents of location $/$ the value of expression \mathbb{E}_2

Example

$$\{y \mapsto -\} [y] := 5 \quad \{y \mapsto 5\}$$

σ	h	$[y] := 5$	σ	h
y 0xAB	0xAB -		y 0xAB	0xAB 5

Heap allocation axiom

Heap allocation axiom:

$$\{x = v \wedge \text{emp}\} \quad x := \text{cons}(\mathbb{E}_1, \dots, \mathbb{E}_n) \quad \{x \mapsto \mathbb{E}_1[x/v], \dots, \mathbb{E}_n[x/v]\}$$

where v is a variable diff. from x and not appearing in $\mathbb{E}_1, \dots, \mathbb{E}_n$

Heap allocation assignment axiom means: if $\sigma(x) = v$ and the heap is empty then executing $x := \text{cons}(\mathbb{E}_1, \dots, \mathbb{E}_n)$ gives a heap consisting of n new consecutive locations, where location $\sigma(x) + i$ contains $\sigma(\mathbb{E}_{i+1}[x/v])$

Heap allocation axiom

Heap allocation axiom:

$$\{x = v \wedge \text{emp}\} \quad x := \text{cons}(E_1, \dots, E_n) \quad \{x \mapsto E_1[x/v], \dots, E_n[x/v]\}$$

where v is a variable diff. from x and not appearing in E_1, \dots, E_n

Heap allocation assignment axiom means: if $\sigma(x) = v$ and the heap is empty then executing $x := \text{cons}(E_1, \dots, E_n)$ gives a heap consisting of n new consecutive locations, where location $\sigma(x) + i$ contains $\sigma(E_{i+1}[x/v])$

σ	h
x	-
y	7

$x := \text{cons}(5, y + 1)$

σ	h
x	0xAB
y	7

Heap allocation axiom

Heap allocation axiom:

$$\{x = v \wedge \text{emp}\} \quad x := \text{cons}(\mathbb{E}_1, \dots, \mathbb{E}_n) \quad \{x \mapsto \mathbb{E}_1[x/v], \dots, \mathbb{E}_n[x/v]\}$$

where v is a variable diff. from x and not appearing in $\mathbb{E}_1, \dots, \mathbb{E}_n$

Heap allocation assignment axiom means: if $\sigma(x) = v$ and the heap is empty then executing $x := \text{cons}(\mathbb{E}_1, \dots, \mathbb{E}_n)$ gives a heap consisting of n new consecutive locations, where location $\sigma(x) + i$ contains $\sigma(\mathbb{E}_{i+1}[x/v])$

σ	h
x	-
y	7

$$x := \text{cons}(5, y + 1)$$

σ	h
x	0xAB
y	7

$x \mapsto \mathbb{E}_1[x/v], \dots, \mathbb{E}_n[x/v]$ abbreviates

$x \mapsto \mathbb{E}_1[x/v] * (x + 1) \mapsto \mathbb{E}_2[x/v] * \dots * (x + n - 1) \mapsto \mathbb{E}_n[x/v]$

Heap deallocation axiom

Heap deallocation axiom: $\{\mathbb{E} \mapsto -\} \text{ dispose } \mathbb{E} \{\text{emp}\}$

where $(\mathbb{E} \mapsto -)$ abbreviates $(\exists z. \mathbb{E} \mapsto z)$ and z does not occur in \mathbb{E}

Heap deallocation axiom

Heap deallocation axiom: $\{\mathbb{E} \mapsto -\} \text{ dispose } \mathbb{E} \{ \text{emp} \}$

where $(\mathbb{E} \mapsto -)$ abbreviates $(\exists z. \mathbb{E} \mapsto z)$ and z does no occur in \mathbb{E}

Heap deallocation: **dispose** \mathbb{E}

- evaluate \mathbb{E} to get location $/$
- **fault** if location $/$ is not in the current heap
- otherwise remove location $/$ from the heap

Heap deallocation axiom means: if the heap is a singleton with domain $\sigma(\mathbb{E})$ then executing **dispose** \mathbb{E} results in the empty heap.

Separation logic axioms - recap

Store assignment axiom:

$$\{x = v \wedge \text{emp}\} x := E \{x = E[x/v] \wedge \text{emp}\}$$

where v is an auxiliary variable which does not occur in E

Heap reads axiom:

$$\{x = v_1 \wedge E \mapsto v_2\} x := [E] \{x = v_2 \wedge E[x/v_1] \mapsto v_2\}$$

where v_1 and v_2 are auxiliary variables which do not occur in E

Heap writes axiom: $\{E_1 \mapsto -\}[E_1] := E_2 \{E_1 \mapsto E_2\}$

where $(E_1 \mapsto -)$ abbreviates $(\exists z. E_1 \mapsto z)$ and z does not occur in E_1

Heap allocation axiom:

$$\{x = v \wedge \text{emp}\} x := \text{cons}(E_1, \dots, E_n) \{x \mapsto E_1[x/v], \dots, E_n[x/v]\}$$

where v is a variable diff. from x and not appearing in E_1, \dots, E_n

Heap deallocation axiom: $\{E \mapsto -\} \text{ dispose } E \{\text{emp}\}$

where $(E \mapsto -)$ abbreviates $(\exists z. E \mapsto z)$ and z does not occur in E

The frame rule

Frame rule:

$$\frac{\{P\} \textcolor{violet}{C} \{Q\}}{\{P * R\} \textcolor{violet}{C} \{Q * R\}}$$

where no variables modified by $\textcolor{violet}{C}$ appears free in R .

The Frame rule means that $\{P\} \textcolor{violet}{C} \{Q\}$ is restricted to the variables and parts of the heap that are actually used by $\textcolor{violet}{C}$.

The frame rule

Frame rule:

$$\frac{\{P\} \text{ C } \{Q\}}{\{P * R\} \text{ C } \{Q * R\}}$$

where no variables modified by C appears free in R.

Example

Is the following instance a legal instance of the Frame rule?

If so, why and if not, why not?

$$\frac{\{\text{emp}\} \text{ x := cons(1) } \{x \mapsto 1\}}{\{\text{emp} * x \mapsto 1\} \text{ x := cons(1) } \{x \mapsto 1 * x \mapsto 1\}}$$

The frame rule

Frame rule:

$$\frac{\{P\} \text{ C } \{Q\}}{\{P * R\} \text{ C } \{Q * R\}}$$

where no variables modified by C appears free in R.

Example

Is the following instance a legal instance of the Frame rule?

If so, why and if not, why not?

$$\frac{\{\text{emp}\} \text{ x := cons(1) } \{x \mapsto 1\}}{\{\text{emp} * x \mapsto 1\} \text{ x := cons(1) } \{x \mapsto 1 * x \mapsto 1\}}$$

No, the command modifies x and R contains a free occurrence of x.

SAT solvers

Propositional Logic

Formulas are defined by

$$\varphi ::= p \mid \top \mid \perp \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi$$

starting from propositional variables (atoms).

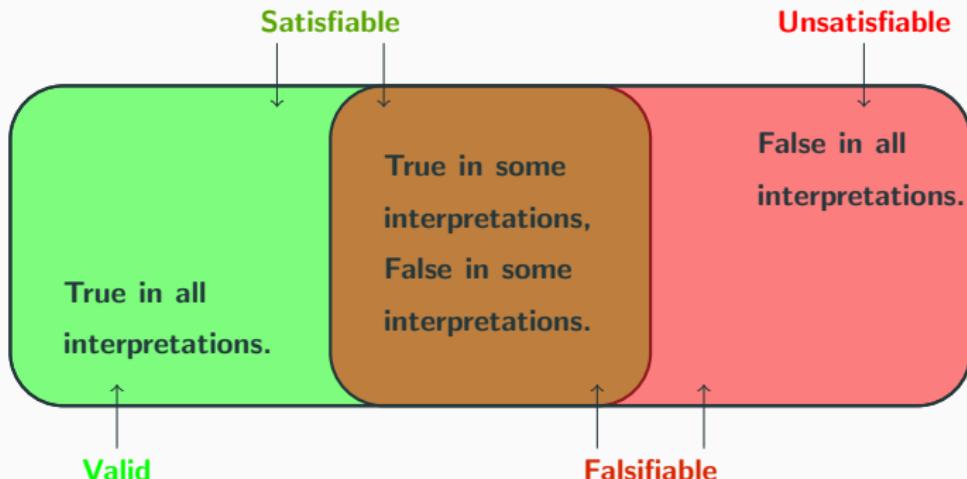
Interpretations assign truth values to propositional variables (true/false).

Further, we can compute the truth value of any formula (e.g., using truth tables).

A formula is satisfiable if there exists an interpretation which makes the formula true.

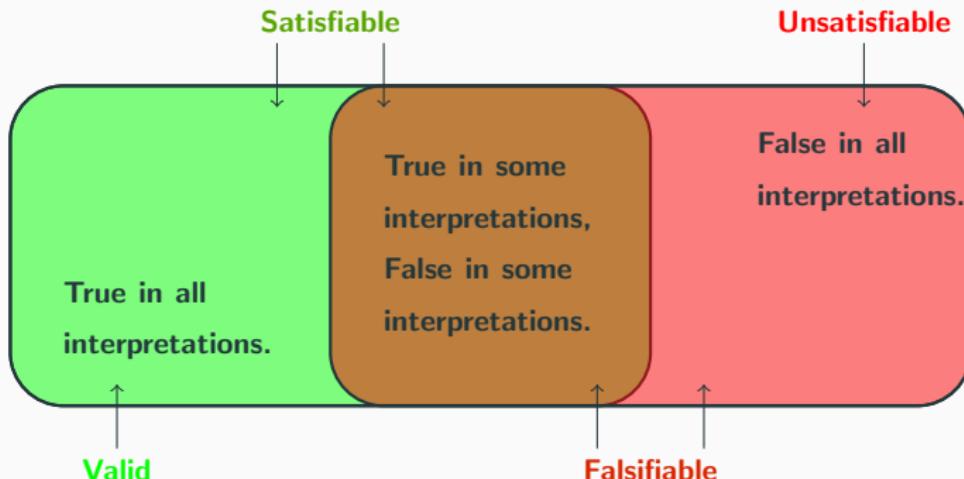
A formula is valid if it is true under all interpretations.

Satisfiability and Validity of formulas



valid = not falsifiable
satisfiable = not unsatisfiable

Satisfiability and Validity of formulas



valid = not falsifiable

satisfiable = not unsatisfiable

φ satisfiable iff $\neg\varphi$ is falsifiable iff $\neg\varphi$ is not valid

φ valid iff $\neg\varphi$ is unsatisfiable

The SAT problem

The SAT problem:

*Given a propositional formula with n variables,
can we find an interpretation to make the formula true?*

Is the formula **satisfiable**? If so, **how**?

The SAT problem

The SAT problem:

*Given a propositional formula with n variables,
can we find an interpretation to make the formula true?*

Is the formula satisfiable? If so, how?

The first known NP-complete problem, as proved by Stephen Cook in 1971.

The SAT problem

The SAT problem:

*Given a propositional formula with n variables,
can we find an interpretation to make the formula true?*

Is the formula satisfiable? If so, how?

The first known NP-complete problem, as proved by Stephen Cook in 1971.

Since SAT is NP-complete, is there hope?

SAT solvers

A **SAT solver** is a program that **automatically decides** whether a propositional formula is satisfiable (i.e, answers the SAT problem).

If it is satisfiable, a SAT solver will produce an example of an interpretation that satisfies the formula.

SAT solvers

A SAT solver is a program that automatically decides whether a propositional formula is satisfiable (i.e, answers the SAT problem).

If it is satisfiable, a SAT solver will produce an example of an interpretation that satisfies the formula.

Naive algorithm: enumerate all assignments to the n variables in the formula
(2^n assignments!)

Worst case complexity is exponential (for all known algorithms).

SAT solvers

A SAT solver is a program that automatically decides whether a propositional formula is satisfiable (i.e, answers the SAT problem).

If it is satisfiable, a SAT solver will produce an example of an interpretation that satisfies the formula.

Naive algorithm: enumerate all assignments to the n variables in the formula (2^n assignments!)

Worst case complexity is exponential (for all known algorithms).

Perhaps surprisingly, many efficient SAT solvers exist!

- average cases encountered in practice can be handled (much) faster
- real problem instances will not be random: exploit implicit structure
- some variables will be tightly correlated with other variables
- some variables will be irrelevant for the difficult parts of the search

SAT solvers

- There are plenty of SAT solvers:
 - MiniSAT, PicoSAT
 - RelSAT
 - GRASP
 - ...
- There are also online SAT solvers:
 - Logictools
 - ...

SAT solvers

- There are plenty of SAT solvers:
 - MiniSAT, PicoSAT
 - RelSAT
 - GRASP
 - ...
- There are also online SAT solvers:
 - Logictools
 - ...
- **SAT competition**
 - The First International SAT Competition in 1992, followed by 1993, 1996, since 2002 every year, affiliated with the SAT conference

SAT solvers

- There are plenty of SAT solvers:
 - MiniSAT, PicoSAT
 - RelSAT
 - GRASP
 - ...
- There are also online SAT solvers:
 - Logictools
 - ...
- SAT competition
 - The First International SAT Competition in 1992, followed by 1993, 1996, since 2002 every year, affiliated with the SAT conference
- Early 90's: 100 variables, 200 clauses
- Today: 1,000,000 variables and 5,000,000 clauses.

Applications

Where can we find SAT technology today?

- Formal methods
 - Hardware model checking
 - Software model checking
 - Termination analysis of term-rewrite systems
 - Test pattern generation (testing of software & hardware)
 - ...
- Artificial intelligence
 - Planning
 - Knowledge representation
 - Games (n-queens, *sudoku*, ...)

Applications

Where can we find SAT technology today?

- Bioinformatics
 - Haplotype inference
 - Pedigree checking
 - ...
- Design automation
 - Equivalence checking
 - Fault diagnosis
 - Noise analysis
 - ...
- Security
 - Cryptanalysis
 - Inversion attacks on hash functions
 - ...

Applications

Where can we find SAT technology today?

- Computationally hard problems
 - Graph coloring
 - Traveling salesperson
 - ...
- Mathematical problems
 - van der Waerden numbers
 - Quasigroup open problems
 - ...
- Core engine for other solvers
- Integrated into theorem provers
 - HOL
 - Isabelle
 - ...

An example - Pythagorean Triples

Is it possible to assign to each integer $1, 2, \dots, n$ one of two colors such that if $a^2 + b^2 = c^2$ then a, b , and c do not all have the same color?

An example - Pythagorean Triples

Is it possible to assign to each integer $1, 2, \dots, n$ one of two colors such that if $a^2 + b^2 = c^2$ then a, b , and c do not all have the same color?

- Solution: nope
- for $n = 7825$ it is not possible
- the proof obtained by a SAT solver has 200 Terrabytes
- the largest Math proof ever ([see the article](#))

An example - Pythagorean Triples

Is it possible to assign to each integer $1, 2, \dots, n$ one of two colors such that if $a^2 + b^2 = c^2$ then a, b , and c do not all have the same color?

- Solution: nope
- for $n = 7825$ it is not possible
- the proof obtained by a SAT solver has 200 Terrabytes
- the largest Math proof ever ([see the article](#))

How to encode this problem?

- for each integer i we have a Boolean variable x_i
- $x_i = 1$ if the color of i is 1 and $x_i = 0$ otherwise
- for each a, b, c such that $a^2 + b^2 = c^2$ we have two clauses:

$$(x_a \vee x_b \vee x_c) \sim \neg(\neg x_a \wedge \neg x_b \wedge \neg x_c)$$

$$(\neg x_a \vee \neg x_b \vee \neg x_c) \sim \neg(x_a \wedge x_b \wedge x_c)$$

CNF - Conjunctive normal form

All current fast SAT solvers work on CNF (or slightly generalized CNF).

CNF - Conjunctive normal form

All current fast SAT solvers work on CNF (or slightly generalized CNF).

- A **literal** is a propositional variable or its negation
 - example: $p, \neg q$
 - For a literal $/$ we write $\sim /$ for the negation of $/$ cancelling double negations

CNF - Conjunctive normal form

All current fast SAT solvers work on CNF (or slightly generalized CNF).

- A **literal** is a propositional variable or its negation
 - example: $p, \neg q$
 - For a literal $/$ we write $\sim /$ for the negation of $/$ cancelling double negations
- A **clause** is a **disjunction of literals**
 - example: $p \vee \neg q \vee r$
 - Since \vee is associative, we can represent clauses as **lists of literals**.
 - The **empty clause** (0 disjuncts) is defined to be \perp
 - A **unit clause** is a clause consisting of exactly one literal.

CNF - Conjunctive normal form

All current fast SAT solvers work on CNF (or slightly generalized CNF).

- A **literal** is a propositional variable or its negation
 - example: $p, \neg q$
 - For a literal l we write $\sim l$ for the negation of l cancelling double negations
- A **clause** is a **disjunction of literals**
 - example: $p \vee \neg q \vee r$
 - Since \vee is associative, we can represent clauses as **lists of literals**.
 - The **empty clause** (0 disjuncts) is defined to be \perp
 - A **unit clause** is a clause consisting of exactly one literal.
- A formula is in **CNF** if it is a **conjunction of clauses**
 - example: $(p \vee \neg q \vee r) \wedge (\neg p \vee s \vee t \vee \neg u)$
 - Since \wedge is associative, we can represent formulas in CNF as **lists of clauses**.
 - The **empty conjunction** is defined to be \top

We can represent CNF formulas as **vectors of vectors of literals**, which are often integers.

If "5" means x_5 , then "-5" means $\neg x_5$.

Example

$[[2, -1], [3, -2, 1]]$

is the representation of the CNF formula

$$(x_2 \vee \neg x_1) \wedge (x_3 \vee \neg x_2 \vee x_1)$$

- the most common input format for SAT solvers
- a way to encode CNF formulas

Example

The input

```
c This is a comment
c This is another comment
p cnf 6 3
1 -2 3 0
2 4 5 0
4 6 0
```

represents the CNF formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee x_4 \vee x_5) \wedge (x_4 \vee x_6)$

DIMACS Format

- At the beginning there can exist one or more comment line.
- Comment lines start with a **c**
- The following lines are information about the expression itself
- the Problem line starts with a **p**:

p FORMAT VARIABLES CLAUSES

- FORMAT should always be **cnf**
- VARIABLES is the number of variables in the expression
- CLAUSES is the number of clauses in the expression

Example

p cnf 6 3 expresses that there are 6 variables and 3 clauses

- The next CLAUSES lines are for the clauses themselves
- Variables are enumerated from 1 to VARIABLES
- A **negation** is represented by **-**
- Each variable information is separated by a blank space
- A **0** is added at the end to mark the end of the clause

Example

1 -2 3 0 expresses the clause $(x_1 \vee \neg x_2 \vee x_3)$

A planning problem

Example

Scheduling a meeting considering the following constraints:

- Adam can only meet on Monday and Wednesday
- Bridget cannot meet on Wednesday
- Charles cannot meet on Friday
- Darren can only meet on Thursday or Friday

A planning problem

Example

Scheduling a meeting considering the following constraints:

- Adam can only meet on Monday and Wednesday
- Bridget cannot meet on Wednesday
- Charles cannot meet on Friday
- Darren can only meet on Thursday or Friday

We represent week day *Monday*, *Tuesday*, ... as variables x_1, x_2, \dots

A planning problem

Example

Scheduling a meeting considering the following constraints:

- Adam can only meet on Monday and Wednesday
- Bridget cannot meet on Wednesday
- Charles cannot meet on Friday
- Darren can only meet on Thursday or Friday

We represent week day *Monday*, *Tuesday*, ... as variables x_1, x_2, \dots

We obtain the following formula in CNF:

$$\varphi = (x_1 \vee x_3) \wedge (\neg x_3) \wedge (\neg x_5) \wedge (x_4 \vee x_5) \wedge \text{AtMostOne}$$

A planning problem

Example (cont.)

AtMostOne =

$$\begin{aligned} & (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_5) \wedge \\ & (\neg x_2 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_5) \wedge \\ & (\neg x_3 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_5) \wedge \\ & (\neg x_4 \vee \neg x_5) \end{aligned}$$

A planning problem

Example (cont.)

$$\varphi = (x_1 \vee x_3) \wedge (\neg x_3) \wedge (\neg x_5) \wedge (x_4 \vee x_5) \wedge \text{AtMostOne}$$

$$\begin{aligned}\text{AtMostOne} = & (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_5) \wedge (\neg x_2 \vee \neg x_3) \wedge \\ & (\neg x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_3 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_5)\end{aligned}$$

DIMACS Format c Planning a meeting

p cnf 5 14

1 3 0

-3 0

-5 0

4 5 0

-1 -2 0

-1 -3 0

-1 -4 0

-1 -5 0

-2 -3 0

-2 -4 0

-2 -5 0

-3 -4 0

-3 -5 0

-4 -5 0

Quiz time!

<https://www.questionpro.com/t/AT4NiZrmaJ>

References – Separation logic

- Lecture Notes on "Formal Methods for Software Engineering", Australian National University, Rajeev Goré.
- John Reynolds, "Introduction to Separation Logic (Preliminary Draft)", parts 1-4.
- Peter W. O'Hearn, "Resources, Concurrency and Local Reasoning", Theoretical Computer Science, Volume 375, Issue 1-3, Pages 271-307, 2007.

C06 – SAT solvers

Program Verification

FMI · Denisa Diaconescu · Spring 2022

CNF - Conjunctive normal form

The SAT problem

The SAT problem:

*Given a propositional formula with n variables,
can we find an interpretation to make the formula true?*

A SAT solver is a program that automatically decides whether a propositional formula is satisfiable (i.e, answers the SAT problem).

If it is satisfiable, a SAT solver will produce an example of an interpretation that satisfies the formula.

CNF - Conjunctive normal form

All current fast SAT solvers work on CNF (or slightly generalized CNF).

- A **literal** is a propositional variable or its negation
 - example: $p, \neg q$
 - For a literal l we write $\sim l$ for the negation of l cancelling double negations
- A **clause** is a **disjunction of literals**
 - example: $p \vee \neg q \vee r$
 - Since \vee is associative, we can represent clauses as **lists of literals**.
 - The **empty clause** (0 disjuncts) is defined to be \perp
 - A **unit clause** is a clause consisting of exactly one literal.
- A formula is in **CNF** if it is a **conjunction of clauses**
 - example: $(p \vee \neg q \vee r) \wedge (\neg p \vee s \vee t \vee \neg u)$
 - Since \wedge is associative, we can represent formulas in CNF as **lists of clauses**.
 - The **empty conjunction** is defined to be \top

Conversion to CNF

Any propositional formula can be transformed into an **equivalent** formula in CNF (**need not be unique!**).

Two formulas are **equivalent** if they are satisfied by the same interpretations.

Conversion to CNF

Any propositional formula can be transformed into an **equivalent** formula in CNF (**need not be unique!**).

Two formulas are **equivalent** if they are satisfied by the same interpretations.

Example

The formula p is equivalent with the following formulas in CNF:

- p
- $p \wedge (p \vee q)$

Conversion to CNF

We can rewrite the formula directly via the following equivalences:

- Remove implications: rewrite $A \rightarrow B$ to $\neg A \vee B$
- Push all negations inwards:
 - rewrite $\neg(A \vee B)$ to $\neg A \wedge \neg B$
 - rewrite $\neg(A \wedge B)$ to $\neg A \vee \neg B$
- Remove double negations: rewrite $\neg\neg A$ to A
- Eliminate \top and \perp :
 - rewrite $A \vee \perp$ to A
 - remove clauses containing \top
- Distribute disjunctions over conjunctions: rewrite $A \vee (B \wedge C)$ to $(A \vee B) \wedge (A \vee C)$

Conversion to CNF

Example

Applying the above rules to the formula

$$\neg p \wedge q \rightarrow p \wedge (r \rightarrow q)$$

we obtain the equivalent formula in CNF:

Conversion to CNF

Example

Applying the above rules to the formula

$$\neg p \wedge q \rightarrow p \wedge (r \rightarrow q)$$

we obtain the equivalent formula in CNF:

$$(p \vee \neg q \vee p) \wedge (p \vee \neg q \vee \neg r \vee q).$$

Conversion to CNF

Example

Applying the above rules to the formula

$$\neg p \wedge q \rightarrow p \wedge (r \rightarrow q)$$

we obtain the equivalent formula in CNF:

$$(p \vee \neg q \vee p) \wedge (p \vee \neg q \vee \neg r \vee q).$$

We can further **simplify**:

- Remove duplicate clauses, duplicate literals from clauses
- Remove clauses in which a literal is both positive and negative
- In fact, each **variable** need only occur in each clause **at most once!**

Example

If we simplify the CNF formula from the above example we get

$$(p \vee \neg q).$$

CNF and Validity

Theorem

A clause $L_1 \vee \dots \vee L_m$ is valid iff there are $1 \leq i, j \leq m$ such that L_i is $\neg L_j$.

CNF and Validity

Theorem

A clause $L_1 \vee \dots \vee L_m$ is valid iff there are $1 \leq i, j \leq m$ such that L_i is $\neg L_j$.

Checking validity for formulas in CNF is very easy! For each clause of the formula, check if it contains a literal and its negation.

Example

- $(\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q$ is not valid
- $(\neg q \vee p \vee q) \wedge (\neg p \vee p)$ is valid

CNF and Validity

Theorem

A clause $L_1 \vee \dots \vee L_m$ is valid iff there are $1 \leq i, j \leq m$ such that L_i is $\neg L_j$.

Checking validity for formulas in CNF is very easy! For each clause of the formula, check if it contains a literal and its negation.

Example

- $(\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q$ is not valid
- $(\neg q \vee p \vee q) \wedge (\neg p \vee p)$ is valid

Satisfiability is not so easy!

$$\varphi \text{ satisfiable} \quad \text{iff} \quad \neg\varphi \text{ is not valid}$$

Conversion to CNF

- The previous method to transform a formula into an equivalent one in CNF **can blow-up exponentially!**
- There exist transformations into CNF that avoid an exponential increase in size by **preserving satisfiability rather than equivalence**.
- These transformations are guaranteed to only linearly increase the size of the formula, but introduce new variables (e.g., [Tseitin transformation](#)).
- Two formulas are **equisatisfiable** if either both formulas are satisfiable or both are not
 - Equisatisfiable formulas may disagree for a particular choice of variables.

SAT solvers algorithms

Davis-Putnam algorithm

- First attempt at a better-than-brute-force SAT algorithm (1960)
 - Original algorithm tackles first-order logic
 - We present the propositional case
- We assume as **input a formula A in CNF**
 - a set of clauses
 - a set of sets of literals
- The **DP algorithm** rewrites the set of clauses until
 - A is \top (the set is empty) then returns **sat**, or
 - A contains an empty clause \perp return **unsat**

Resolution rule

$$\boxed{\frac{p \vee \alpha \quad \neg p \vee \beta}{\alpha \vee \beta} \quad \textit{resolution}}$$

$$\boxed{\frac{p \vee p \vee \alpha}{p \vee \alpha} \quad \textit{merging}}$$

Resolution rule

$$\frac{p \vee \alpha \quad \neg p \vee \beta}{\alpha \vee \beta} \quad \textit{resolution}$$

$$\frac{p \vee p \vee \alpha}{p \vee \alpha} \quad \textit{merging}$$

Example

$$\frac{x_1 \vee x_2 \vee x_3 \quad x_1 \vee \neg x_2 \vee x_4}{x_1 \vee x_1 \vee x_3 \vee x_4}$$

$$\frac{x_1 \vee x_1 \vee x_3 \vee x_4}{x_1 \vee x_3 \vee x_4}$$

Resolution rule

If a variable p occurs both **positively** and **negatively** in clauses of A :

- Let $C_{pos} = \{A_1 \vee p, A_2 \vee p, \dots\}$ be the clauses in A in which p occurs positively
- Let $C_{neg} = \{B_1 \vee \neg p, B_2 \vee \neg p, \dots\}$ be the clauses in A in which p occurs negatively
- Remove these two sets of clauses from A , and replace them with the new set

$$\{A_i \vee B_j \mid A_i \vee p \in C_{pos}, B_j \vee \neg p \in C_{neg}\}$$

Davis-Putnam algorithm

- Iteratively apply the following steps:
 - Select variable x
 - Apply resolution rule between every pair of clauses of the form $(x \vee \alpha)$ and $(\neg x \vee \beta)$
 - Remove all clauses containing either x or $\neg x$
- Terminate if
 - The empty formula is derived (\top) and then return **sat**, or
 - An empty clause is derived (\perp) and then return **unsat**

Davis-Putnam algorithm

Example

$$1. (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)$$

Davis-Putnam algorithm

Example

1. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)$
2. $(\neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)$

Davis-Putnam algorithm

Example

1. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)$
2. $(\neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)$
3. $(\neg x_3 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)$

Davis-Putnam algorithm

Example

1. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)$
2. $(\neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)$
3. $(\neg x_3 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)$
4. $(\neg x_3 \vee x_3) \wedge (x_3)$

Davis-Putnam algorithm

Example

1. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)$
2. $(\neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)$
3. $(\neg x_3 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4)$
4. $(\neg x_3 \vee x_3) \wedge (x_3)$
5. \top

Formula is **SAT**

Davis-Putnam algorithm

Example

1. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2) \wedge (\neg x_2 \vee x_3)$

Davis-Putnam algorithm

Example

1. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2) \wedge (\neg x_2 \vee x_3)$
2. $(\neg x_2 \vee \neg x_3) \wedge (x_2) \wedge (\neg x_2 \vee x_3)$

Davis-Putnam algorithm

Example

1. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2) \wedge (\neg x_2 \vee x_3)$
2. $(\neg x_2 \vee \neg x_3) \wedge (x_2) \wedge (\neg x_2 \vee x_3)$
3. $(\neg x_3) \wedge (x_3)$

Davis-Putnam algorithm

Example

1. $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2) \wedge (\neg x_2 \vee x_3)$
2. $(\neg x_2 \vee \neg x_3) \wedge (x_2) \wedge (\neg x_2 \vee x_3)$
3. $(\neg x_3) \wedge (x_3)$
4. \perp

Formula is **UNSAT**

Davis-Putnam algorithm

Main issues of the approach:

- In which order should the resolution steps be performed?
- In which order the variables should be selected? (variable elimination)
- Worst-case exponential in memory consumption!

Faster algorithms

- Davis-Putnam algorithm: the refinements
 - Add specific cases to order variable elimination steps.
 - The [pure literal rule](#) and the [unit propagation rule](#)

Faster algorithms

- Davis-Putnam algorithm: the refinements
 - Add specific cases to order variable elimination steps.
 - The **pure literal rule** and the **unit propagation rule**
- Davis-Putnam-Logemann-Loveland algorithm
 - Standard **backtrack search**
 - space efficient DP

- Davis-Putnam algorithm: the refinements
 - Add specific cases to order variable elimination steps.
 - The **pure literal rule** and the **unit propagation rule**
- Davis-Putnam-Logemann-Loveland algorithm
 - Standard **backtrack search**
 - space efficient DP
- Conflict-Driven Clause Learning algorithm
 - An extension of DPLL with:
 - **Clause learning**
 - **Non-chronological backtracking**
 - Clause learning can be performed with various strategies
 - CDCL algorithms are used in almost all modern SAT solvers

Davis-Putnam algorithm: the refinements

Add specific cases to order variable elimination steps.

- Iteratively apply the following steps:
 - Apply the pure literal rule and unit propagation
 - Select variable x
 - Apply resolution rule between every pair of clauses of the form $(x \vee \alpha)$ and $(\neg x \vee \beta)$
 - Remove all clauses containing either x or $\neg x$
- Terminate if
 - The empty formula is derived (\top) and then return sat, or
 - An empty clause is derived (\perp) and then return unsat

Pure literal rule

If a variable p occurs either **only positively** or **only negatively** in A ,
delete all clauses of A in which p occurs.

- A literal is **pure** if occurs only positively or negatively in a CNF formula
- **Pure literal rule:** eliminate first pure literals since no resolvent are produced!
- Applying a variable elimination step on a pure literal strictly reduced the number of clauses!
- Preserves satisfiability, not logical equivalence!

Pure literal rule

If a variable p occurs either **only positively** or **only negatively** in A ,
delete all clauses of A in which p occurs.

- A literal is **pure** if occurs only positively or negatively in a CNF formula
- **Pure literal rule:** eliminate first pure literals since no resolvent are produced!
- Applying a variable elimination step on a pure literal strictly reduced the number of clauses!
- Preserves satisfiability, not logical equivalence!

Example

In $(\neg x_1 \vee x_2) \wedge (x_3 \vee \neg x_2) \wedge (x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_4)$ the pure literals are $\neg x_1$ and x_3 .

Unit propagation rule

If $/$ is a unit clause in A , then update A by:

- removing all clauses which have $/$ as a disjunct, and
- updating all clauses in A containing $\sim /$ as a disjunct by removing that disjunct

- Specific case of resolution
- Only shorten clauses!
- a.k.a. Boolean constraint propagation or BCP
- Is arguably the key component to fast SAT solving
- Since clauses are shortened, new unit clauses may appear.
Empty clauses also!
- Apply unit propagation while new unit clauses are produced.
- Preserves logical equivalence!

DP algorithm

Example

1. $p \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r \vee s \vee t)$

DP algorithm

Example

1. $p \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r \vee s \vee t)$

We apply the Pure literal rule for s and t and we delete the last clause.

DP algorithm

Example

$$1. \ p \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r \vee s \vee t)$$

We apply the Pure literal rule for s and t and we delete the last clause.

$$2. \ p \wedge (\neg p \vee q) \wedge (\neg q \vee r)$$

DP algorithm

Example

1. $p \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r \vee s \vee t)$

We apply the Pure literal rule for s and t and we delete the last clause.

2. $p \wedge (\neg p \vee q) \wedge (\neg q \vee r)$

We apply the Unit propagation rule for p .

DP algorithm

Example

$$1. p \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r \vee s \vee t)$$

We apply the **Pure literal rule** for s and t and we delete the last clause.

$$2. p \wedge (\neg p \vee q) \wedge (\neg q \vee r)$$

We apply the **Unit propagation rule** for p .

$$3. q \wedge (\neg q \vee r)$$

DP algorithm

Example

1. $p \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r \vee s \vee t)$

We apply the Pure literal rule for s and t and we delete the last clause.

2. $p \wedge (\neg p \vee q) \wedge (\neg q \vee r)$

We apply the Unit propagation rule for p .

3. $q \wedge (\neg q \vee r)$

We apply the Unit propagation rule for q .

DP algorithm

Example

1. $p \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r \vee s \vee t)$

We apply the **Pure literal rule** for s and t and we delete the last clause.

2. $p \wedge (\neg p \vee q) \wedge (\neg q \vee r)$

We apply the **Unit propagation rule** for p .

3. $q \wedge (\neg q \vee r)$

We apply the **Unit propagation rule** for q .

4. r

DP algorithm

Example

1. $p \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r \vee s \vee t)$

We apply the Pure literal rule for s and t and we delete the last clause.

2. $p \wedge (\neg p \vee q) \wedge (\neg q \vee r)$

We apply the Unit propagation rule for p .

3. $q \wedge (\neg q \vee r)$

We apply the Unit propagation rule for q .

4. r

We apply the Unit propagation rule for r .

DP algorithm

Example

1. $p \wedge (\neg p \vee q) \wedge (\neg q \vee r) \wedge (\neg r \vee s \vee t)$

We apply the Pure literal rule for s and t and we delete the last clause.

2. $p \wedge (\neg p \vee q) \wedge (\neg q \vee r)$

We apply the Unit propagation rule for p .

3. $q \wedge (\neg q \vee r)$

We apply the Unit propagation rule for q .

4. r

We apply the Unit propagation rule for r .

5. \top

The formula is SAT

DP: The limits

- The approach runs easily out of memory
- The solution: using backtrack search!

Davis-Putnam-Logemann- Loveland algorithm

Preliminary definitions

- Propositional variable can be assigned value **False** or **True**.
 - In some contexts variables may be **unassigned**
- A clause is **satisfied** if at least one of its literals is assigned value **true**
 - $(x_1 \vee \neg x_2 \vee \neg x_3)$
- A clause is **unsatisfied** if all of its literals are assigned value **false**
 - $(x_1 \vee \neg x_2 \vee \neg x_3)$
- A clause is **unit** if it contains one single unassigned literal and all other literals are assigned value **false**
 - $(x_1 \vee \neg x_2 \vee \neg x_3)$
- A formula is **satisfied** if **all** its clauses are satisfied
- A formula is **unsatisfied** if **at least one** of its clauses is unsatisfied

Davis-Putnam-Logemann-Loveland algorithm

- DPLL algorithm
- Standard backtrack search
- space efficient DP

DPLL(F, \mathcal{I}):

- Apply unit propagation
- If conflict identified, return UNSAT
- Apply the pure literal rule
- If F is satisfied (empty), return SAT
- Select decision variable x
 - If $\text{DPLL}(F, \mathcal{I} \cup x) = \text{SAT}$ return SAT
 - return $\text{DPLL}(F, \mathcal{I} \cup x)$

Notes:

- ✗ We use red to denote that a variable / literal is false
- ✗ We use green to denote that a variable / literal is true

Conflict all disjuncts of a clause are assigned false.

Pure literals in backtrack search

As before.

Example

$$(\neg x_1 \vee x_2) \wedge (x_3 \vee \neg x_2) \wedge (x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_4)$$

becomes

$$(x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_4)$$

Unit propagation in backtrack search

Unit clause rule in backtrack search:

Given a unit clause, its only unassigned literal must be assigned value **true** for the clause to be satisfied.

Example

For unit clause ($x_1 \vee \neg x_2 \vee \neg x_3$), the variable x_3 must be assigned value **false**.

Unit propagation rule: Iterated application of the unit clause rule.

Unit propagation in backtrack search

Example (1)

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$

Unit propagation in backtrack search

Example (1)

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$
- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$

Unit propagation in backtrack search

Example (1)

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$
- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$
- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$

Unit propagation in backtrack search

Example (2)

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$

Unit propagation in backtrack search

Example (2)

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$
- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$

Unit propagation in backtrack search

Example (2)

- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$
- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$
- $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$

Conflict!

Unit propagation can **satisfy** clauses but can also **unsatisfy** clauses (i.e. **conflicts**)

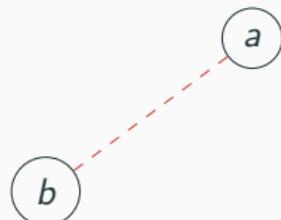
An example of DPLL

$$\begin{aligned}(a \vee \neg b \vee d) \wedge \\(a \vee \neg b \vee e) \wedge \\(\neg b \vee \neg d \vee \neg e) \wedge \\(a \vee b \vee c \vee d) \wedge \\(a \vee b \vee c \vee \neg d) \wedge \\(a \vee b \vee \neg c \vee e) \wedge \\(a \vee b \vee \neg c \vee \neg e)\end{aligned}$$

An example of DPLL

Select decision variable: *a*

$$\begin{aligned} & (\textcolor{red}{a} \vee \neg b \vee d) \wedge \\ & (\textcolor{red}{a} \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (\textcolor{red}{a} \vee b \vee c \vee d) \wedge \\ & (\textcolor{red}{a} \vee b \vee c \vee \neg d) \wedge \\ & (\textcolor{red}{a} \vee b \vee \neg c \vee e) \wedge \\ & (\textcolor{red}{a} \vee b \vee \neg c \vee \neg e) \end{aligned}$$



Conflict

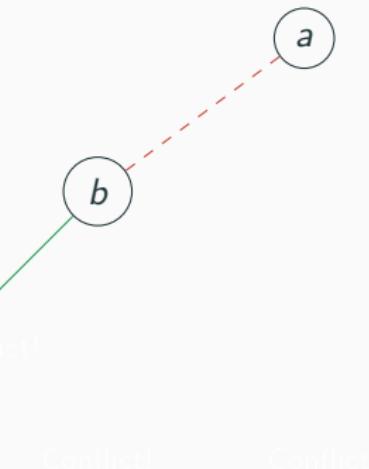
Conflict

Conflict

An example of DPLL

Select decision variable: *b*

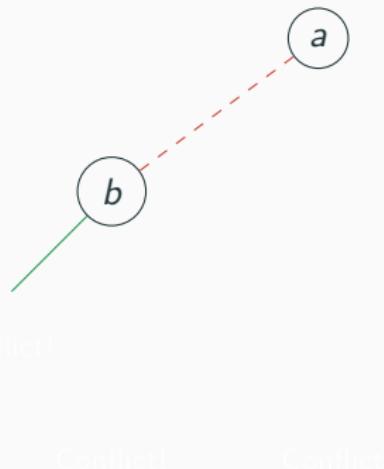
$$\begin{aligned} & (a \vee \neg b \vee d) \wedge \\ & (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge \\ & (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge \\ & (a \vee b \vee \neg c \vee \neg e) \end{aligned}$$



An example of DPLL

Unit propagation: *d*

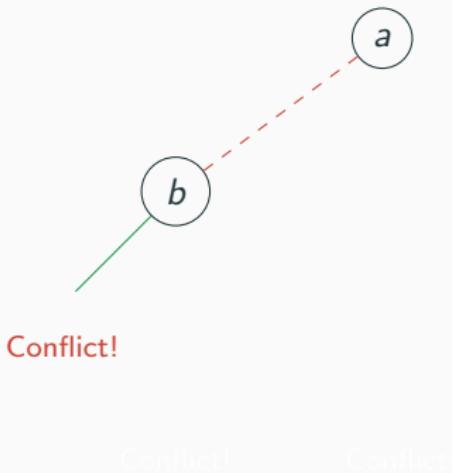
$$\begin{aligned} & (a \vee \neg b \vee d) \wedge \\ & (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge \\ & (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge \\ & (a \vee b \vee \neg c \vee \neg e) \end{aligned}$$



An example of DPLL

Unit propagation: e

$$\begin{aligned} & (a \vee \neg b \vee d) \wedge \\ & (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge \\ & (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge \\ & (a \vee b \vee \neg c \vee \neg e) \end{aligned}$$



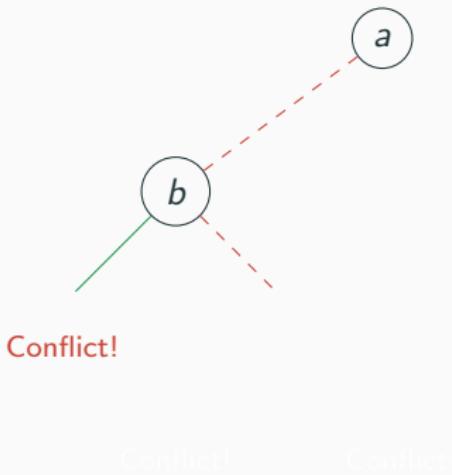
Conflict!

Conflict!

An example of DPLL

Select decision variable: *b*

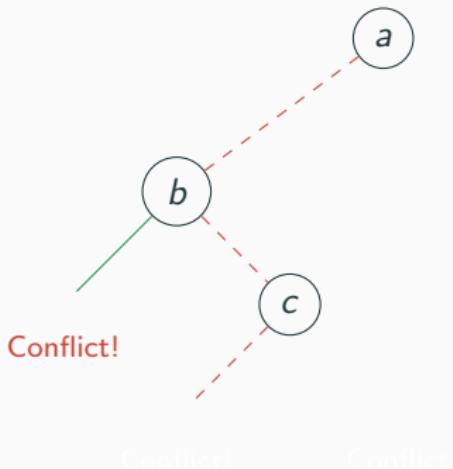
$$\begin{aligned} & (a \vee \neg b \vee d) \wedge \\ & (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge \\ & (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge \\ & (a \vee b \vee \neg c \vee \neg e) \end{aligned}$$



An example of DPLL

Select decision variable: c

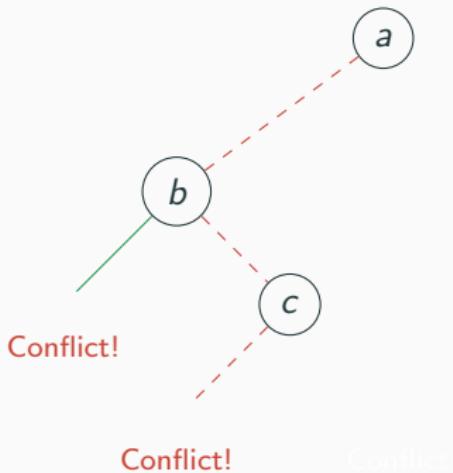
$$\begin{aligned} & (a \vee \neg b \vee d) \wedge \\ & (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge \\ & (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge \\ & (a \vee b \vee \neg c \vee \neg e) \end{aligned}$$



An example of DPLL

Unit propagation: *d*

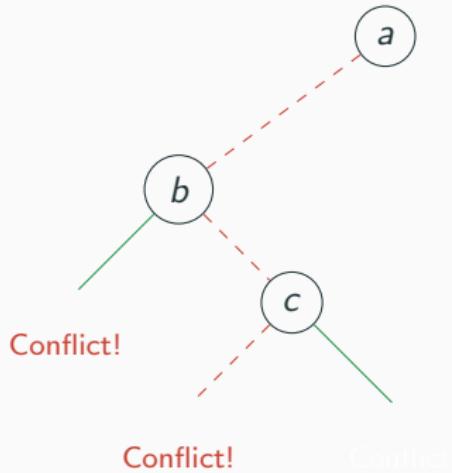
$$\begin{aligned} & (a \vee \neg b \vee d) \wedge \\ & (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge \\ & (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge \\ & (a \vee b \vee \neg c \vee \neg e) \end{aligned}$$



An example of DPLL

Select decision variable: c

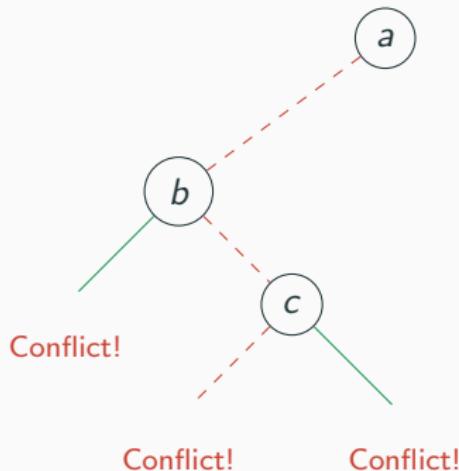
$$\begin{aligned} & (a \vee \neg b \vee d) \wedge \\ & (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge \\ & (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge \\ & (a \vee b \vee \neg c \vee \neg e) \end{aligned}$$



An example of DPLL

Unit propagation: e

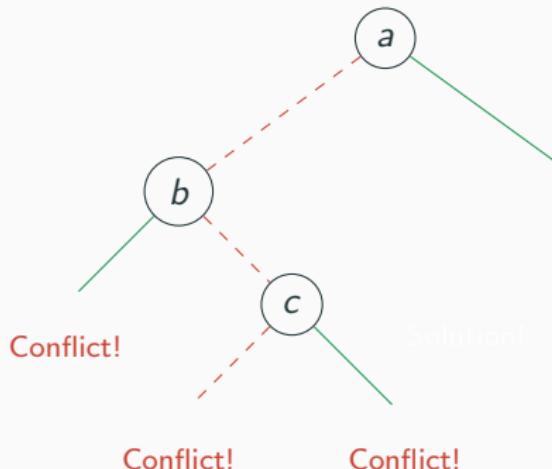
$$\begin{aligned} & (a \vee \neg b \vee d) \wedge \\ & (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge \\ & (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge \\ & (a \vee b \vee \neg c \vee \neg e) \end{aligned}$$



An example of DPLL

Select decision variable: *a*

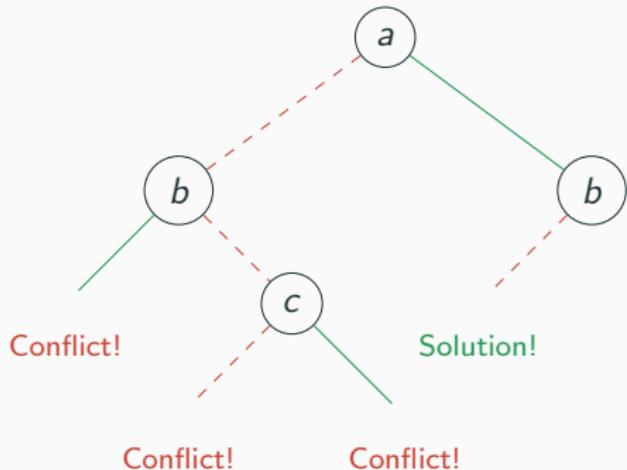
$(a \vee \neg b \vee d) \wedge$
 $(a \vee \neg b \vee e) \wedge$
 $(\neg b \vee \neg d \vee \neg e) \wedge$
 $(a \vee b \vee c \vee d) \wedge$
 $(a \vee b \vee c \vee \neg d) \wedge$
 $(a \vee b \vee \neg c \vee e) \wedge$
 $(a \vee b \vee \neg c \vee \neg e)$



An example of DPLL

Select decision variable: *b*

$$\begin{aligned} & (a \vee \neg b \vee d) \wedge \\ & (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge \\ & (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge \\ & (a \vee b \vee \neg c \vee \neg e) \end{aligned}$$



Conflict-Driven Clause Learning algorithm

Conflict-Driven Clause Learning algorithm

- CDCL
- An extension of DPLL with:
 - Clause learning
 - Non-chronological backtracking
- Clause learning can be performed with various strategies
- CDCL algorithms are used in almost all modern SAT solvers

Clause learning

During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict.

Example

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

Clause learning

During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict.

Example

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

- Assume decision $c = \text{False}$ and $f = \text{False}$

Clause learning

During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict.

Example

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

- Assume decision $c = \text{False}$ and $f = \text{False}$
- Assign $a = \text{False}$

Clause learning

During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict.

Example

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

- Assume decision $c = \text{False}$ and $f = \text{False}$
- Assign $a = \text{False}$
- Unit propagation b

Clause learning

During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict.

Example

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

- Assume decision $c = \text{False}$ and $f = \text{False}$
- Assign $a = \text{False}$
- Unit propagation b
- Unit propagation d

Clause learning

During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict.

Example

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

- Assume decision $c = \text{False}$ and $f = \text{False}$
- Assign $a = \text{False}$
- Unit propagation b
- Unit propagation d and e

Clause learning

During backtrack search, for each conflict learn new clause, which explains and prevents repetition of the same conflict.

Example

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

- Assume decision $c = \text{False}$ and $f = \text{False}$
- Assign $a = \text{False}$
- Unit propagation b
- Unit propagation d and e
- A conflict is reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied
- $\varphi \wedge \neg a \wedge \neg c \wedge \neg f \rightarrow \perp$, therefore $\varphi \rightarrow a \vee c \vee f$
- Learn new clause $(a \vee c \vee f)$

Non-chronological backtracking

- aka conflict directed backjumping
- During backtrack search, for each conflict backtrack to one of the causes of conflict.

Non-chronological backtracking

Example

$$\begin{aligned}\varphi = & (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ & (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

Non-chronological backtracking

Example

$$\begin{aligned}\varphi = & (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ & (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

- Assume decision $c = \text{False}$, $f = \text{False}$, $h = \text{False}$ and $i = \text{False}$

Non-chronological backtracking

Example

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

- Assume decision $c = \text{False}$, $f = \text{False}$, $h = \text{False}$ and $i = \text{False}$
- Learnt clause $(a \vee c \vee f)$ unit-propagates a

Non-chronological backtracking

Example

$$\begin{aligned}\varphi = & (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ & (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

- Assume decision $c = \text{False}$, $f = \text{False}$, $h = \text{False}$ and $i = \text{False}$
- Assignment $a = \text{False}$ cause conflict. Learnt clause $(a \vee c \vee f)$ implies a
- Unit propagation g

Non-chronological backtracking

Example

$$\begin{aligned}\varphi = & (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ & (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

- Assume decision $c = \text{False}$, $f = \text{False}$, $h = \text{False}$ and $i = \text{False}$
- Assignment $a = \text{False}$ cause conflict. Learnt clause $(a \vee c \vee f)$ implies a
- Unit propagation g, b

Non-chronological backtracking

Example

$$\begin{aligned}\varphi = & (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ & (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

- Assume decision $c = \text{False}$, $f = \text{False}$, $h = \text{False}$ and $i = \text{False}$
- Assignment $a = \text{False}$ cause conflict. Learnt clause $(a \vee c \vee f)$ implies a
- Unit propagation g, b, d

Non-chronological backtracking

Example

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

- Assume decision $c = \text{False}$, $f = \text{False}$, $h = \text{False}$ and $i = \text{False}$
- Assignment $a = \text{False}$ cause conflict. Learnt clause $(a \vee c \vee f)$ implies a
- Unit propagation g , b , d , and e

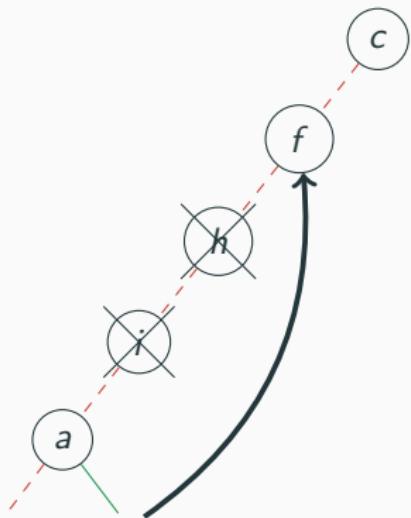
Non-chronological backtracking

Example

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

- Assume decision $c = \text{False}$, $f = \text{False}$, $h = \text{False}$ and $i = \text{False}$
- Assignment $a = \text{False}$ cause conflict. Learnt clause $(a \vee c \vee f)$ implies a
- Unit propagation g , b , d , and e
- A conflict is again reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied
- Learn new clause $(c \vee f)$

Non-chronological backtracking



Conflict! Conflict!

$$a \vee c \vee f \quad c \vee f$$

- Learnt clause: $c \vee f$
- Need to backtrack, given new clause
- Backtrack to most recent decision
 $f = \text{false}$
- Clause learning and non-chronological backtracking are hallmarks of modern SAT solvers

C07 – SMT Solvers

Program Verification

FMI · Denisa Diaconescu · Spring 2022

Overview

What are SMT solvers?

First-order logic (in 5 min)

Some first-order theories

How SMT solvers works?

What are SMT solvers?

The SMT problem

The SMT problem:

Given a first-order logic formula, with symbols from (possibly several) theories, does it have a model?

Is the formula satisfiable? If so, how?

The SAT problem is a special case, in which

- the formula is quantifier-free, without function symbols or equality
- no theories are used

SAT solving algorithms are an important ingredient in SMT solvers

First-order theories

- Whereas the language of SAT solvers is Boolean logic, the language of SMT solvers is **first-order logic**.
- **First-order theories** allow us to capture structures which are used by programs (e.g., arrays, integers) and enable reasoning about them.
- **Validity in first order logic (FOL) is undecidable!**
 - **Lambda calculus** – Alonzo Church (1936)
 - **Turing machines** – Alan Turing (1937)
 - **Recursive functions** – Kurt Gödel (1934) and Stephen Kleene (1936)
- Validity in particular first order theories is (sometimes) decidable.

Combine propositional satisfiability search techniques with solvers for specific first-order theories:

- Linear arithmetic
- Bit vectors
- Arrays
- ...



Applications of SMT solvers

SMT solvers are used as core engines in many tools in

- program analysis and verification
- software engineering
- hardware verification
- ...
- symbolic execution
- concolic execution

Many SMT solvers:

- Z3 (Microsoft)
- Yices
- MathSAT
- CVC4
- ...

First-order logic (in 5 min)

- The language includes the Boolean operations of propositional logic, but instead of propositional variables, more complicated expressions are allowed.
- A first-order language must specify its signature: the set of constant, function, and predicate symbols that are allowed.
- Each predicate and function symbol has an associated arity: a natural number indicating how many arguments it takes.
 - Equality is a special predicate symbol of arity 2
 - Constant symbols can also be thought of as functions whose arity is 0

Example (Propositional logic)

- Equality: no
- Predicate symbols: x_1, x_2, \dots
- Constant symbols: none
- Function symbols: none

Example (Propositional logic)

- Equality: no
- Predicate symbols: x_1, x_2, \dots
- Constant symbols: none
- Function symbols: none

Example (Elementary Number Theory)

- Equality: yes
- Predicate symbols: $<$
- Constant symbols: 0
- Function symbols: S (successor), $+$, $*$

- Terms
 - Variables and constants are terms
 - For each function symbol f of arity n , and terms t_1, \dots, t_n , $f(t_1, \dots, t_n)$ is a term

- **Terms**
 - **Variables** and **constants** are terms
 - For each function symbol f of arity n , and terms t_1, \dots, t_n , $f(t_1, \dots, t_n)$ is a term
- **Atomic formulas** are expressions of the form
 - $P(t_1, \dots, t_n)$ where P is a predicate symbol of arity n and t_1, \dots, t_n are terms.
 - Predicate symbols of arity 0 are called **propositional atoms**.
 - $t = t'$ if the logic is with equality where t, t' are terms.

- Terms
 - Variables and constants are terms
 - For each function symbol f of arity n , and terms t_1, \dots, t_n , $f(t_1, \dots, t_n)$ is a term
- Atomic formulas are expressions of the form
 - $P(t_1, \dots, t_n)$ where P is a predicate symbol of arity n and t_1, \dots, t_n are terms.
 - Predicate symbols of arity 0 are called propositional atoms.
 - $t = t'$ if the logic is with equality where t, t' are terms.
- An atomic formula or its negation is called a literal.

- Terms
 - Variables and constants are terms
 - For each function symbol f of arity n , and terms t_1, \dots, t_n , $f(t_1, \dots, t_n)$ is a term
- Atomic formulas are expressions of the form
 - $P(t_1, \dots, t_n)$ where P is a predicate symbol of arity n and t_1, \dots, t_n are terms.
 - Predicate symbols of arity 0 are called propositional atoms.
 - $t = t'$ if the logic is with equality where t, t' are terms.
- An atomic formula or its negation is called a literal.
- Formulas are built from literals using the Boolean operators and quantification. If α is a formula, then for every variables x
 - $\forall x. \alpha$ is a formula
 - $\exists x. \alpha$ is a formula

Given a signature Σ and a set V of variables, a **structure** \mathcal{M} of Σ consists of:

1. A nonempty set M called the **domain** of \mathcal{M}
2. For each constant c in Σ , an element $c^{\mathcal{M}} \in M$.
3. For each n -ary function symbol f in Σ , an n -ary function $f^{\mathcal{M}} : M^n \rightarrow M$.
4. For each n -ary predicate symbol P in Σ , an n -ary relation $P^{\mathcal{M}} \subseteq M^n$.

An **interpretation** I of the variables in V into a structure \mathcal{M} maps each variable $v \in V$ to an element $I(v) \in M$.

Example

Consider the signature with a single predicate symbol \in and a single constant symbol \emptyset .

Example

Consider the signature with a single predicate symbol \in and a single constant symbol \emptyset .

A possible structure \mathcal{M} for this signature has the domain \mathbb{N} , the set of natural numbers, $\in^{\mathcal{M}} = <$, and $\emptyset^{\mathcal{M}} = 0$.

Example

Consider the signature with a single predicate symbol \in and a single constant symbol \emptyset .

A possible structure \mathcal{M} for this signature has the domain \mathbb{N} , the set of natural numbers, $\in^{\mathcal{M}} = <$, and $\emptyset^{\mathcal{M}} = 0$.

Consider the sentence $\exists x \forall y \neg(y \in x)$.

Example

Consider the signature with a single predicate symbol \in and a single constant symbol \emptyset .

A possible structure \mathcal{M} for this signature has the domain \mathbb{N} , the set of natural numbers, $\in^{\mathcal{M}} = <$, and $\emptyset^{\mathcal{M}} = 0$.

Consider the sentence $\exists x \forall y \neg(y \in x)$.

What does this sentence mean in this structure?

Example

Consider the signature with a single predicate symbol \in and a single constant symbol \emptyset .

A possible structure \mathcal{M} for this signature has the domain \mathbb{N} , the set of natural numbers, $\in^{\mathcal{M}} = <$, and $\emptyset^{\mathcal{M}} = 0$.

Consider the sentence $\exists x \forall y \neg(y \in x)$.

What does this sentence mean in this structure?

There is a natural number x such that no other natural number is smaller than x .

Example

Consider the signature with a single predicate symbol \in and a single constant symbol \emptyset .

A possible structure \mathcal{M} for this signature has the domain \mathbb{N} , the set of natural numbers, $\in^{\mathcal{M}} = <$, and $\emptyset^{\mathcal{M}} = 0$.

Consider the sentence $\exists x \forall y \neg(y \in x)$.

What does this sentence mean in this structure?

There is a natural number x such that no other natural number is smaller than x .

Is this sentence true in the structure?

Example

Consider the signature with a single predicate symbol \in and a single constant symbol \emptyset .

A possible structure \mathcal{M} for this signature has the domain \mathbb{N} , the set of natural numbers, $\in^{\mathcal{M}} = <$, and $\emptyset^{\mathcal{M}} = 0$.

Consider the sentence $\exists x \forall y \neg(y \in x)$.

What does this sentence mean in this structure?

There is a natural number x such that no other natural number is smaller than x .

Is this sentence true in the structure?

Since 0 has this property, the sentence is true in this structure.

Term interpretation

Given

- a structure \mathcal{M} of Σ , and
- an interpretation I of V into \mathcal{M} ,

We can inductively extend the interpretation I to all terms over Σ with variables from V , as follows:

- $\bar{I}(v) = I(v)$ for any $v \in V$
- $\bar{I}(c) = c^{\mathcal{M}}$ for any constant symbol c in Σ
- $\bar{I}(f(t_1, \dots, t_n)) = f^{\mathcal{M}}(\bar{I}(t_1), \dots, \bar{I}(t_n))$, for any n -ary function symbol $f \in \Sigma$ and any terms $(t_i)_{1 \leq i \leq n}$

Satisfaction

Given

- a structure \mathcal{M} of Σ , and
- an interpretation I of V into \mathcal{M} ,

We can inductively define a satisfaction relation for Σ -formulas with variables from V , as follows:

- $\mathcal{M}, I \models t_1 = t_2$ iff $\bar{I}(t_1) = \bar{I}(t_2)$
- $\mathcal{M}, I \models P(t_1, \dots, t_n)$ iff $(\bar{I}(t_1), \dots, \bar{I}(t_n)) \in P^{\mathcal{M}}$
- $\mathcal{M}, I \models \phi_1 \wedge \phi_2$ iff $\mathcal{M}, I \models \phi_1$ and $\mathcal{M}, I \models \phi_2$
- $\mathcal{M}, I \models \neg\phi$ iff $\mathcal{M}, I \not\models \phi$
- $\mathcal{M}, I \models \exists x.\phi$ iff there exists $a \in M$ such that $\mathcal{M}, I[x \leftarrow a] \models \phi$

Satisfiability and validity in a structure

Given

- a structure \mathcal{M} of Σ , and
- a Σ formula ϕ with variables from V

We say that

- ϕ is satisfiable in \mathcal{M} iff $\mathcal{M}, I \models \phi$ for **some** interpretation I
- ϕ is valid in \mathcal{M} iff $\mathcal{M}, I \models \phi$ for **any** interpretation I

Satisfiability and validity in a structure

Given

- a structure \mathcal{M} of Σ , and
- a Σ formula ϕ with variables from V

We say that

- ϕ is satisfiable in \mathcal{M} iff $\mathcal{M}, I \models \phi$ for **some** interpretation I
- ϕ is valid in \mathcal{M} iff $\mathcal{M}, I \models \phi$ for **any** interpretation I

Note that ϕ is valid iff $\neg\phi$ is not satisfiable. Indeed,

- ϕ is valid iff $\mathcal{M}, I \models \phi$ for any I
- iff there exists no I such that $\mathcal{M}, I \not\models \phi$
- iff there exists no I such that $\mathcal{M}, I \models \neg\phi$
- iff $\neg\phi$ is not satisfiable

Satisfiability, validity, and free variables

Given a formula ϕ and the set X of its free variables

- If I_1, I_2 interpretations such that $I_1|_X = I_2|_X$
- Then $\mathcal{M}, I_1 \models \phi$ iff $\mathcal{M}, I_2 \models \phi$

Hence, if a formula ϕ does not have free variables

- The satisfaction relation does not depend on interpretations
We can therefore write $\mathcal{M} \models \phi$
- satisfiability and validity coincide for ϕ

Satisfiability, validity, and closed formulas

Given a formula ϕ and the set $\{x_1, \dots, x_n\}$ of its free variables,

- ϕ satisfiable in \mathcal{M} iff $\mathcal{M} \models \exists x_1. \dots \exists x_n. \phi$
- ϕ valid in \mathcal{M} iff $\mathcal{M} \models \forall x_1. \dots \forall x_n. \phi$

Satisfiability and validity in a theory

A **theory** is a set of sentences (no free variables).

Given a theory T , a formula φ is

- **T -valid** if φ is satisfied by all models of T for all interpretations of V .
- **T -satisfiable** if it is satisfied by some model of T for some interpretations of V .

Drinker's paradox

Example

$$\exists x(D(x) \rightarrow \forall y D(y))$$

*There is someone in the pub such that
if he/she is drinking, then everyone in the pub is drinking.*

How is this formula?

1. valid
2. unsatisfiable
3. satisfiable but not valid



Drinker's paradox:

Example

$$\exists x(D(x) \rightarrow \forall y D(y))$$

*There is someone in the pub such that
if he/she is drinking, then everyone in the pub is drinking.*

How is this formula?

1. valid
2. unsatisfiable
3. satisfiable but not valid



SMT problem

The validity problem for T is the problem of deciding, for each formula φ , whether φ is T -valid.

The validity problem for T is the problem of deciding, for each formula φ , whether φ is T -valid.

The satisfiability problem for T , or the problem of satisfiability modulo theories, is the problem of deciding, for each formula φ , whether φ is T -satisfiable.

$$\varphi \text{ is } T\text{-valid} \quad \text{iff} \quad \neg\varphi \text{ is } T\text{-unsatisfiable.}$$

SMT problem

The validity problem for T is the problem of deciding, for each formula φ , whether φ is T -valid.

The satisfiability problem for T , or the problem of satisfiability modulo theories, is the problem of deciding, for each formula φ , whether φ is T -satisfiable.

φ is T -valid iff $\neg\varphi$ is T -unsatisfiable.

SMT = satisfiability modulo theories

Satisfiability modulo theories

It is important to make a distinction between SMT and standard first order satisfiability.

For example, is the following sentence satisfiable?

$$\text{read}(\text{write}(a, i, v), i) \neq v$$

Satisfiability modulo theories

It is important to make a distinction between SMT and standard first order satisfiability.

For example, is the following sentence satisfiable?

$$\text{read}(\text{write}(a, i, v), i) \neq v$$

If the set of allowable models is unrestricted, then the answer is **yes**.

However, if we only consider models that obey the axioms for **read** and **write** then the answer is **no**.

Quiz time!

<https://www.questionpro.com/t/AT4NiZr2Xb>

Some first-order theories

First-order theories

A **first-order theory** T is defined by

- **Signature** Σ_T = set of constant, function, and predicate symbols
 - Have no meaning
- **Axioms** A_T = set of Σ_T -sentences (no free variables)
 - Provide meaning for the symbols of Σ_T

A **fragment** of a theory T is a syntactically restricted subset of formulas of the theory.

First-order theories

A first-order theory T is defined by

- Signature Σ_T = set of constant, function, and predicate symbols
 - Have no meaning
- Axioms A_T = set of Σ_T -sentences (no free variables)
 - Provide meaning for the symbols of Σ_T

A fragment of a theory T is a syntactically restricted subset of formulas of the theory.

Example

The quantifier-free fragment of a theory T (denoted $QFF T$) is the set of formulas without quantifiers that are valid in T .

First-order theories

A first-order theory T is defined by

- Signature Σ_T = set of constant, function, and predicate symbols
 - Have no meaning
- Axioms A_T = set of Σ_T -sentences (no free variables)
 - Provide meaning for the symbols of Σ_T

A fragment of a theory T is a syntactically restricted subset of formulas of the theory.

Example

The quantifier-free fragment of a theory T (denoted $QFF T$) is the set of formulas without quantifiers that are valid in T .

A theory is decidable if for every formula in the theory we can automatically check whether the formula is valid or not.

Similarly for fragments of a theory.

First-order theories

- In principle, SMT can be applied to any theory T .
- In practice, when people talk about SMT, they are usually referring to a small set of specific theories.
- We will consider a few examples of theories which are of particular interest in program analysis and verification applications.

First-order theories

Theory	Description	Full Fragment	No Quantifiers
T_E	Equality	NO	YES
T_{PA}	Peano arithmetic	NO	NO
T_N	Presburger arithmetic	YES	YES
T_Z	Linear Integers	YES	YES
T_R	Reals (with *)	YES	YES
T_Q	Rationals (without *)	YES	YES
T_{RDS}	Recursive Data Structures	NO	YES
T_{RDS}^+	Acyclic Recursive Data Structures	YES	YES
T_A	Arrays	NO	YES
$T_A^=$	Arrays with extensionality	NO	YES

source: *The Calculus of Computation*, Manna and Bradley

Theory of Equality

Signature Σ_E :

- Any function, predicate, and constant
- The predicate $=$ which is interpreted (i.e., defined via axioms)

Axioms A_E :

- $\forall x. x = x$
- $\forall x, y. x = y \rightarrow y = x$
- $\forall x, y, z. x = y \wedge y = z \rightarrow x = z$
- $\forall x_1, \dots, \forall x_n, y_1, \dots, y_n. x_1 = y_1 \wedge \dots \wedge x_n = y_n \rightarrow f(x_1, \dots, x_n) = f(y_1, \dots, y_n)$

Theory of Equality

Signature Σ_E :

- Any function, predicate, and constant
- The predicate $=$ which is interpreted (i.e., defined via axioms)

Axioms A_E :

- $\forall x. x = x$
- $\forall x, y. x = y \rightarrow y = x$
- $\forall x, y, z. x = y \wedge y = z \rightarrow x = z$
- $\forall x_1, \dots, \forall x_n, y_1, \dots, y_n. x_1 = y_1 \wedge \dots \wedge x_n = y_n \rightarrow f(x_1, \dots, x_n) = f(y_1, \dots, y_n)$

T_E is undecidable

QFF T_E is decidable

Theory of Equality

Example

$$(a = b) \wedge (b = c) \rightarrow (g(f(a), b) = g(f(c), a))$$

Exercise: Is this valid? If yes, prove it.

Arithmetic: Natural numbers and Integers

Natural numbers $\mathbb{N} = \{0, 1, 2, \dots\}$

Integers $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$

Three theories:

- Peano arithmetic T_{PA}
 - Natural numbers with addition (+), multiplication (*), equality (=)
- Presburger arithmetic T_N
 - Natural numbers with addition (+), equality (=)
- Theory of integers T_Z
 - Integers with addition (+), subtraction (-), comparison (>), equality (=), multiplication by constants

Theory of Peano Arithmetic

Signature Σ_E :

- Constants: 0, 1
- Binary functions: +, *
- Predicate: =

Axioms A_{PA} :

- $\forall x. \neg(x + 1 = 0)$
- $\forall x. x + 0 = x$
- $\forall x. x * 0 = 0$
- $\forall x, y. x + 1 = y + 1 \rightarrow x = y$
- $\forall x, y. x + (y + 1) = (x + y) + 1$
- $\forall x, y. x * (y + 1) = (x * y) + x$
- For each formula $\varphi(x, \bar{y})$ in the language Σ_E ,
$$\forall \bar{y}. \varphi(0, \bar{y}) \wedge (\forall x. \varphi(x, \bar{y}) \rightarrow \varphi(x + 1, \bar{y})) \rightarrow \forall x. \varphi(x, \bar{y})$$
 (induction)

Theory of Peano Arithmetic

Signature Σ_E :

- Constants: 0, 1
- Binary functions: +, *
- Predicate: =

Axioms A_{PA} :

- $\forall x. \neg(x + 1 = 0)$
- $\forall x. x + 0 = x$
- $\forall x. x * 0 = 0$
- $\forall x, y. x + 1 = y + 1 \rightarrow x = y$
- $\forall x, y. x + (y + 1) = (x + y) + 1$
- $\forall x, y. x * (y + 1) = (x * y) + x$
- For each formula $\varphi(x, \bar{y})$ in the language Σ_E ,
$$\forall \bar{y}. \varphi(0, \bar{y}) \wedge (\forall x. \varphi(x, \bar{y}) \rightarrow \varphi(x + 1, \bar{y})) \rightarrow \forall x. \varphi(x, \bar{y})$$
 (induction)

T_{PA} is undecidable
 $QFF\ T_{PA}$ is undecidable

Theory of Peano Arithmetic

Example

Is the formula

$$3 * x + 2 = 2 * y \text{ in } T_{PA}?$$

Theory of Peano Arithmetic

Example

Is the formula

$$3 * x + 2 = 2 * y \text{ in } T_{PA}?$$

Yes! It can be written as

$$(1 + 1 + 1) * x + 1 + 1 = (1 + 1) * y$$

Theory of Presburger Arithmetic

Signature Σ_N :

- Constants: 0, 1
- Binary functions: + NO multiplication!
- Predicate: =

Axioms A_N :

- $\forall x. \neg(x + 1 = 0)$
- $\forall x. x + 0 = x$
- $\forall x, y. x + 1 = y + 1 \rightarrow x = y$
- $\forall x, y. x + (y + 1) = (x + y) + 1$
- For each formula $\varphi(x, \bar{y})$ in the language Σ_E ,
 $\forall \bar{y}. \varphi(0, \bar{y}) \wedge (\forall x. \varphi(x, \bar{y}) \rightarrow \varphi(x + 1, \bar{y})) \rightarrow \forall x. \varphi(x, \bar{y})$ (induction)

Theory of Presburger Arithmetic

Signature Σ_N :

- Constants: 0, 1
- Binary functions: + NO multiplication!
- Predicate: =

Axioms A_N :

- $\forall x. \neg(x + 1 = 0)$
- $\forall x. x + 0 = x$
- $\forall x, y. x + 1 = y + 1 \rightarrow x = y$
- $\forall x, y. x + (y + 1) = (x + y) + 1$
- For each formula $\varphi(x, \bar{y})$ in the language Σ_E ,
 $\forall \bar{y}. \varphi(0, \bar{y}) \wedge (\forall x. \varphi(x, \bar{y}) \rightarrow \varphi(x + 1, \bar{y})) \rightarrow \forall x. \varphi(x, \bar{y})$ (induction)

T_N is decidable

QFF T_N is decidable

Theory of Integers

Signature Σ_Z :

- Constants: $\dots, -3, -2, -1, 0, 1, 2, 3, \dots$
- Unary function: $\dots, -3*, -2*, 2*, 3*, \dots$
(intended meaning $2 * x$ is $x + x$, $-3 * x$ is $-x - x - x$)
- Binary functions: $+$ and $-$
- Predicate: $=$ and $>$

T_Z is decidable

$QFF\ T_Z$ is decidable

Theory of Arrays

Signature Σ_A :

- Functions:
 - $\text{read}(_, _)$
 - written for simplicity as $_[_]$
 - e.g. $a[i]$
 - $\text{write}(_, _, _)$
 - e.g. $\text{write}(a, v, i)$ denotes the array a' where $a[v] = i$ and all other entries are the same as a
- Predicate: $=$

Axioms A_A :

- Same as A_E
- $\forall a, i, j. \ i = j \rightarrow a[i] = a[j]$
- $\forall a, v, i, j. \ i = j \rightarrow \text{write}(a, i, v)[j] = v$
- $\forall a, v, i, j. \ i \neq j \rightarrow \text{write}(a, i, v)[j] = a[j]$

Theory of Arrays

Signature Σ_A :

- Functions:
 - $\text{read}(_, _)$
 - written for simplicity as $_[_]$
 - e.g. $a[i]$
 - $\text{write}(_, _, _)$
 - e.g. $\text{write}(a, v, i)$ denotes the array a' where $a[v] = i$ and all other entries are the same as a
- Predicate: $=$

Axioms A_A :

- Same as A_E
- $\forall a, i, j. \ i = j \rightarrow a[i] = a[j]$
- $\forall a, v, i, j. \ i = j \rightarrow \text{write}(a, i, v)[j] = v$
- $\forall a, v, i, j. \ i \neq j \rightarrow \text{write}(a, i, v)[j] = a[j]$

T_A is undecidable

$QFF\ T_A$ is decidable

How SMT solvers works?

Satisfiability modulo theories

There are two main approaches for SMT solvers:

- The eager approach
 - Tries to find ways of encoding an entire SMT problem into SAT.
 - There are a variety of techniques
 - For some theories, this works quite well.

Satisfiability modulo theories

There are two main approaches for SMT solvers:

- The eager approach
 - Tries to find ways of encoding an entire SMT problem into SAT.
 - There are a variety of techniques
 - For some theories, this works quite well.
- The lazy approach
 - Tries to combine SAT and theory reasoning.
 - The basis for most modern SMT solvers.

SMT: The Big Questions

1. How to solve conjunctions of literals in a theory?
 - Use a Theory solver
2. How to combine a theory solver and a SAT solver to reason about arbitrary formulas?
 - The DPLL(T) framework
3. How to combine theory solvers for several theories?
 - The Nelson-Oppen method and its variants

- Given a theory T , a Theory solver for T takes as input a set (interpreted as an implicit conjunction) φ of literals and determines whether φ is T -satisfiable.
 - φ is T -satisfiable if there is some model \mathcal{M} of T such that φ holds in \mathcal{M} .
- In order to integrate a Theory solver into a modern SMT solver, it is helpful if the Theory solver can do more than just check satisfiability.

Characteristics of Theory solvers

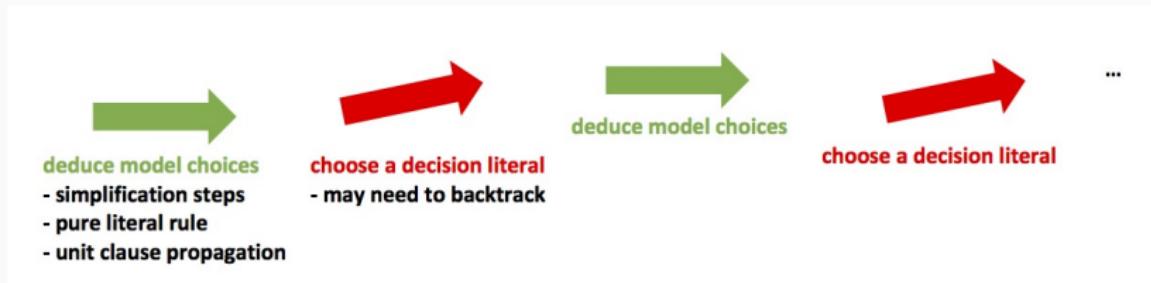
Some desirable characteristics of Theory solvers include:

- **Incrementality** – easy to add new literals or backtrack to a previous state
- **Layered/Lazy** – able to detect simple inconsistencies quickly, able to detect difficult inconsistencies eventually
- **Equality Propagating** – if Theory solvers can detect when two terms are equivalent, this greatly simplifies theory combination
- **Model Generating** – when reporting T -satisfiable, the Theory solver also provides a concrete value for each variable or function symbol
- **Proof Generating** – when reporting T -unsatisfiable, the Theory solver also provides a checkable proof

Propositional Abstraction

- An **atom** is a formula without propositional connectives or quantifiers
 - depending on the signature $f(a) = b, m * n \leq 42$ could be atoms; 42 is not
 - a propositional atom is an uninterpreted constant symbol of sort Bool
- A (first-order) **literal** is an atom or its negation
- For a given signature Σ , we define a signature Σ^P containing only:
 - the **propositional Σ -atoms**
 - a **fresh propositional atom** for each non-propositional Σ -atom
- We then fix an injective mapping from the non-propositional Σ -atoms to the Σ^P -atoms.
- For a Σ -formula φ , the formula φ^P is the **propositional abstraction** of φ , given by replacing all non-propositional Σ -atoms in φ with their image under this mapping.
- An Σ -formula φ is **propositional unsatisfiable** if $\varphi^P \models \perp$.
- An Σ -formula φ **propositionally entails** an Σ -formula ψ if $\varphi^P \models \psi^P$.
 - Note that $\varphi^P \models \psi^P$ implies $\varphi \models \psi$, but **not necessarily vice-versa**.

Recall DPLL/CDCL Algorithms

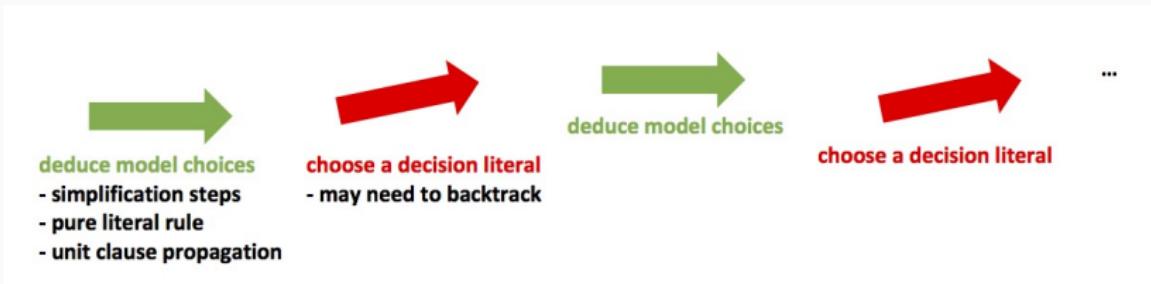


...until...

- conflict reached
 - backtrack - try flipping a decision literal
 - (if CDCL) learn new clause, back-jump
- model found
 - return the model

Adapting DPLL to DPLL(T)

Run DPLL on the propositional abstraction φ^P of the T -input formula φ



...until...

- **conflict reached:** backtrack/jump, learn clauses as usual
- **model found** (represented by a set Γ of literals)
 - It is not necessarily a T -model!
 - Ask theory solver: **is Γ T -satisfiable?**
 - If yes, we are done.
 - If no, backtrack in the original search.
 - (CDCL) get a T -unsatisfiable subset for clause learning/back-jumping

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$
- SAT solver returns model $[1, 2, 3, \neg 4]$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$
- SAT solver returns model $[1, 2, 3, \neg 4]$
- Theory solver detects $[1, 3, \neg 4]$ T-unsat

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$
- SAT solver returns model $[1, 2, 3, \neg 4]$
- Theory solver detects $[1, 3, \neg 4]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2, \neg 1 \vee \neg 3 \vee 4]$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$
- SAT solver returns model $[1, 2, 3, \neg 4]$
- Theory solver detects $[1, 3, \neg 4]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2, \neg 1 \vee \neg 3 \vee 4]$
- SAT solver detects unsat

Theory combination

- Given a theory T , a Theory solver for T takes as input a set (interpreted as an implicit conjunction) φ of literals and determines whether φ is T -satisfiable.
- We are often interested in using two or more theories at the same time.
- Can we combine two theory solvers to get a theory solver for the combined theory?

Theory combination

- Given a theory T , a Theory solver for T takes as input a set (interpreted as an implicit conjunction) φ of literals and determines whether φ is T -satisfiable.
- We are often interested in using two or more theories at the same time.
- Can we combine two theory solvers to get a theory solver for the combined theory?

Example

The following formula uses both T_E and T_Z

$$\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

The Nelson-Oppen Method

A very general method for combining theory solvers is the **Nelson-Oppen method**.

This method is applicable when:

1. The signatures Σ_i are disjoint.
2. The theories T_i are stably-infinite.
 - A Σ -theory T is **stably-infinite** if every T -satisfiable quantifier-free Σ -formula is satisfiable in an infinite model.
3. The formulas to be tested for satisfiability are **conjunctions of quantifier-free literals**.

Extensions exist that can relax each of these restrictions in some cases.

The Nelson-Oppen Method

Some definitions:

- A member of Σ_i is an *i-symbol*.
- A term t is an *i-term* if it starts with an *i-symbol*.
- An *atomic i-formula* is
 - an application of an *i-predicate*,
 - an equation whose lhs is an *i-term*, or
 - an equation whose lhs is a variable and whose rhs is an *i-term*
- An *i-literal* is an atomic *i-formula* or the negation of one.
- An occurrence of a term t in either an *i-term* or an *i-literal* is *i-alien* if it is a j -term with $i \neq j$ and all of its super-terms (if any) are *i-terms*.
- An expression is *pure* if it contains only variables and *i-symbols* for some i .

Conversion to separate form

Given a conjunction of literals φ , we want to convert it into a **separate form**: a T -equisatisfiable conjunction of literals $\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n$ where each φ_i is a Σ_i -formula.

We have the following algorithm:

1. Let ψ be **some literal** in φ .
2. If ψ is a **pure i -literal**, for some i , remove ψ from φ and add ψ to φ_i .
If φ is **empty** then **stop**; otherwise **goto step 1**.
3. Otherwise, ψ is an **i -literal** for some i .
Let t be a **term occurring i -alien** in ψ .
Replace t in φ with a **new variable** z and add $z = t$ to φ .
Goto step 1.

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We convert φ to a separate form:

- $\varphi_E := ?$
- $\varphi_Z := ?$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We convert φ to a separate form:

- $\varphi_E := ?$
- $\varphi_Z := 1 \leq x$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x} \wedge \cancel{x \leq 2} \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We convert φ to a separate form:

- $\varphi_E := ?$
- $\varphi_Z := 1 \leq x \wedge \cancel{x \leq 2}$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2} \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We convert φ to a separate form:

- $\varphi_E := ?$
- $\varphi_Z := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2} \wedge f(x) \neq f(y) \wedge f(x) \neq f(2) \wedge y = 1$$

We convert φ to a separate form:

- $\varphi_E := ?$
- $\varphi_Z := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2} \wedge f(x) \neq f(y) \wedge f(x) \neq f(2) \wedge y = 1$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2} \wedge f(x) \neq f(y) \wedge \cancel{f(x) \neq f(2)} \wedge y = 1$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y)$
- $\varphi_Z := \cancel{1 \leq x \wedge x \leq 2}$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2} \wedge f(x) \neq f(y) \wedge \cancel{f(x) \neq f(z)} \wedge y = 1 \wedge z = 2$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2} \wedge f(x) \neq f(y) \wedge f(x) \neq f(z) \wedge y = 1 \wedge z = 2$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(y) \wedge f(x) \neq f(z) \wedge y = 1 \wedge z = 2$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(y) \wedge f(x) \neq f(z) \wedge y = 1 \wedge z = 2$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(y) \wedge f(x) \neq f(z) \wedge y = 1 \wedge z = 2$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

The Nelson-Oppen Method

- As each φ_i is a Σ_i -formula, we can run a Theory solver Sat_i for each φ_i .
- If any Sat_i reports that φ_i is unsatisfiable, then φ is unsatisfiable.
- The converse is not true in general!
- We need a way for the decision procedures to communicate with each other about shared variables.
- If S is a set of terms and \sim is an equivalence relation on S , then the arrangement of S induced by \sim is

$$Ar_{\sim} = \{x = y \mid x \sim y\} \cup \{x \neq y \mid x \not\sim y\}$$

The Nelson-Oppen Method

Suppose that T_1 and T_2 are theories with disjoint signatures Σ_1 and Σ_2 .

Let $T = \bigcup T_i$ and $\Sigma = \bigcup \Sigma_i$.

Given a Σ -formula φ and decision procedures Sat_1 and Sat_2 for T_1 and T_2 , we wish to determine if φ is T -satisfiable.

The non-deterministic Nelson-Oppen algorithm:

1. Convert φ to its **separate form** $\varphi_1 \wedge \varphi_2$.
2. Let S be the **set of variables shared** between φ_1 and φ_2 .
Guess an **equivalence relation** \sim on S .
3. **Run Sat_1** on $\varphi_1 \cup Ar_{\sim}$.
4. **Run Sat_2** on $\varphi_2 \cup Ar_{\sim}$.

The Nelson-Oppen Method

Suppose that T_1 and T_2 are theories with disjoint signatures Σ_1 and Σ_2 .

Let $T = \bigcup T_i$ and $\Sigma = \bigcup \Sigma_i$.

Given a Σ -formula φ and decision procedures Sat_1 and Sat_2 for T_1 and T_2 , we wish to determine if φ is T -satisfiable.

The non-deterministic Nelson-Oppen algorithm:

1. Convert φ to its **separate form** $\varphi_1 \wedge \varphi_2$.
2. Let S be the **set of variables shared** between φ_1 and φ_2 .
Guess an **equivalence relation** \sim on S .
3. **Run Sat_1** on $\varphi_1 \cup Ar_{\sim}$.
4. **Run Sat_2** on $\varphi_2 \cup Ar_{\sim}$.

If there exists an equivalence relation \sim such that both Sat_1 and Sat_2 succeed, then φ is **T -satisfiable**.

If no such equivalence relation exists, then φ is **T -unsatisfiable**.

The Nelson-Oppen Method

Suppose that T_1 and T_2 are theories with disjoint signatures Σ_1 and Σ_2 .

Let $T = \bigcup T_i$ and $\Sigma = \bigcup \Sigma_i$.

Given a Σ -formula φ and decision procedures Sat_1 and Sat_2 for T_1 and T_2 , we wish to determine if φ is T -satisfiable.

The non-deterministic Nelson-Oppen algorithm:

1. Convert φ to its **separate form** $\varphi_1 \wedge \varphi_2$.
2. Let S be the **set of variables shared** between φ_1 and φ_2 .
Guess an **equivalence relation** \sim on S .
3. **Run Sat_1** on $\varphi_1 \cup Ar_{\sim}$.
4. **Run Sat_2** on $\varphi_2 \cup Ar_{\sim}$.

If there exists an equivalence relation \sim such that both Sat_1 and Sat_2 succeed, then φ is **T -satisfiable**.

If no such equivalence relation exists, then φ is **T -unsatisfiable**.

The generalization to more than two theories is straightforward.

The Nelson-Oppen Method

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We first convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

The Nelson-Oppen Method

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We first convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

The shared variables are $\{x, y, z\}$.

There are 5 possible arrangements based on equivalence classes of x, y , and z (see *Bell number*).

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
 - $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
 - $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$
-
1. $\{x = y, x = z, y = z\}$
 2. $\{x = y, x \neq z, y \neq z\}$
 3. $\{x \neq y, x = z, y \neq z\}$
 4. $\{x \neq y, x \neq z, y = z\}$
 5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_E
2. $\{x = y, x \neq z, y \neq z\}$
3. $\{x \neq y, x = z, y \neq z\}$
4. $\{x \neq y, x \neq z, y = z\}$
5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_E
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_E
3. $\{x \neq y, x = z, y \neq z\}$
4. $\{x \neq y, x \neq z, y = z\}$
5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_E
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_E
3. $\{x \neq y, x = z, y \neq z\}$ inconsistent with T_E
4. $\{x \neq y, x \neq z, y = z\}$
5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_E
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_E
3. $\{x \neq y, x = z, y \neq z\}$ inconsistent with T_E
4. $\{x \neq y, x \neq z, y = z\}$ inconsistent with T_Z
5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_E
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_E
3. $\{x \neq y, x = z, y \neq z\}$ inconsistent with T_E
4. $\{x \neq y, x \neq z, y = z\}$ inconsistent with T_Z
5. $\{x \neq y, x \neq z, y \neq z\}$ inconsistent with T_Z

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_E
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_E
3. $\{x \neq y, x = z, y \neq z\}$ inconsistent with T_E
4. $\{x \neq y, x \neq z, y = z\}$ inconsistent with T_Z
5. $\{x \neq y, x \neq z, y \neq z\}$ inconsistent with T_Z

Conclusion: φ is $T_E \cup T_Z$ -unsatisfiable!

Recall the ingredients:

- Theory solvers for different theories
- Combine a Theory solver and a SAT solver
- Combine Theory solvers for different theories

References

- Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.
- Lecture Notes on "Techniques for Program Analysis and Verification", Stanford, Clark Barrett.
- Lecture Notes on "Program verification", ETH Zurich, Alexander Summers.
- Lecture Notes on "Computer-Aided Reasoning for Software Engineering", University of Washington, Emina Torlak.

C08 – SMT Solvers, Symbolic execution

Program Verification

FMI · Denisa Diaconescu · Spring 2022

Overview

How SMT solvers works?

Symbolic execution

How SMT solvers works?

The SMT problem

The SMT problem:

Given a first-order logic formula, with symbols from (possibly several) theories, does it have a model?

Is the formula satisfiable? If so, how?

The SAT problem is a special case, in which

- the formula is quantifier-free, without function symbols or equality
- no theories are used

SAT solving algorithms are an important ingredient in SMT solvers

First-order theories

- Whereas the language of SAT solvers is Boolean logic, the language of SMT solvers is **first-order logic**.
- **First-order theories** allow us to capture structures which are used by programs (e.g., arrays, integers) and enable reasoning about them.
- **Validity in first order logic (FOL) is undecidable!**
 - **Lambda calculus** – Alonzo Church (1936)
 - **Turing machines** – Alan Turing (1937)
 - **Recursive functions** – Kurt Gödel (1934) and Stephen Kleene (1936)
- Validity in particular first order theories is (sometimes) decidable.

Combine propositional satisfiability search techniques with solvers for specific first-order theories:

- Linear arithmetic
- Bit vectors
- Arrays
- ...



Satisfiability modulo theories

There are two main approaches for SMT solvers:

- The eager approach
 - Tries to find ways of encoding an entire SMT problem into SAT.
 - There are a variety of techniques
 - For some theories, this works quite well.

Satisfiability modulo theories

There are two main approaches for SMT solvers:

- The eager approach
 - Tries to find ways of encoding an entire SMT problem into SAT.
 - There are a variety of techniques
 - For some theories, this works quite well.
- The lazy approach
 - Tries to combine SAT and theory reasoning.
 - The basis for most modern SMT solvers.

SMT: The Big Questions

1. How to solve conjunctions of literals in a theory?
 - Use a Theory solver
2. How to combine a theory solver and a SAT solver to reason about arbitrary formulas?
 - The DPLL(T) framework
3. How to combine theory solvers for several theories?
 - The Nelson-Oppen method and its variants

Theory solver

- Given a theory T , a Theory solver for T takes as input a set (interpreted as an implicit conjunction) φ of literals and determines whether φ is T -satisfiable.
 - φ is T -satisfiable if there is some model \mathcal{M} of T such that φ holds in \mathcal{M} .
- In order to integrate a Theory solver into a modern SMT solver, it is helpful if the Theory solver can do more than just check satisfiability.

Characteristics of Theory solvers

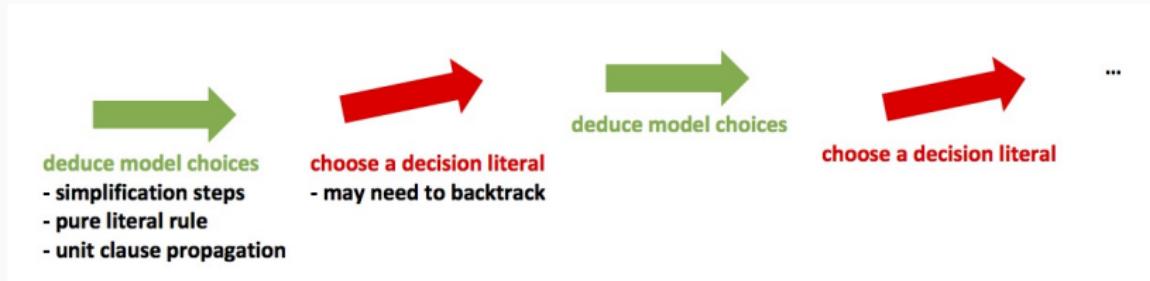
Some desirable characteristics of Theory solvers include:

- **Incrementality** – easy to add new literals or backtrack to a previous state
- **Layered/Lazy** – able to detect simple inconsistencies quickly, able to detect difficult inconsistencies eventually
- **Equality Propagating** – if Theory solvers can detect when two terms are equivalent, this greatly simplifies theory combination
- **Model Generating** – when reporting T -satisfiable, the Theory solver also provides a concrete value for each variable or function symbol
- **Proof Generating** – when reporting T -unsatisfiable, the Theory solver also provides a checkable proof

Propositional Abstraction

- An **atom** is a formula without propositional connectives or quantifiers
 - depending on the signature $f(a) = b, m * n \leq 42$ could be atoms; 42 is not
 - a propositional atom is an uninterpreted constant symbol of sort Bool
- A (first-order) **literal** is an atom or its negation
- For a given signature Σ , we define a signature Σ^P containing only:
 - the **propositional Σ -atoms**
 - a **fresh propositional atom** for each non-propositional Σ -atom
- We then fix an injective mapping from the non-propositional Σ -atoms to the Σ^P -atoms.
- For a Σ -formula φ , the formula φ^P is the **propositional abstraction** of φ , given by replacing all non-propositional Σ -atoms in φ with their image under this mapping.
- An Σ -formula φ is **propositional unsatisfiable** if $\varphi^P \models \perp$.
- An Σ -formula φ **propositionally entails** an Σ -formula ψ if $\varphi^P \models \psi^P$.
 - Note that $\varphi^P \models \psi^P$ implies $\varphi \models \psi$, but **not necessarily vice-versa**.

Recall DPLL/CDCL Algorithms

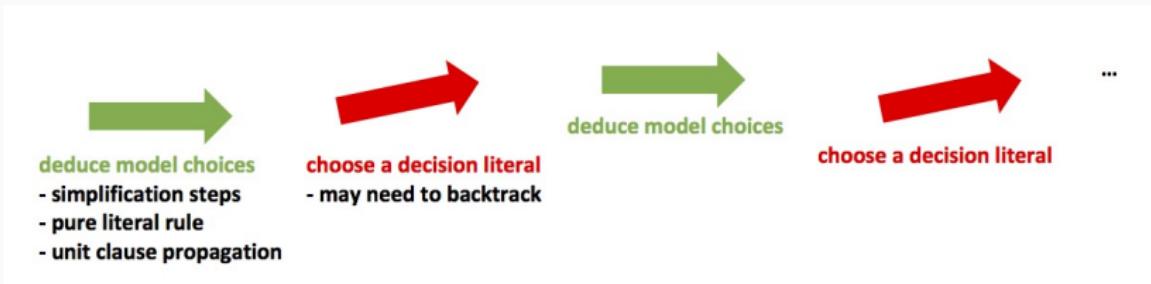


...until...

- conflict reached
 - backtrack - try flipping a decision literal
 - (if CDCL) learn new clause, back-jump
- model found
 - return the model

Adapting DPLL to DPLL(T)

Run DPLL on the propositional abstraction φ^P of the T -input formula φ



...until...

- conflict reached: backtrack/jump, learn clauses as usual
- model found (represented by a set Γ of literals)
 - It is not necessarily a T -model!
 - Ask theory solver: is Γ T -satisfiable?
 - If yes, we are done.
 - If no, backtrack in the original search.
 - (CDCL) get a T -unsatisfiable subset for clause learning/back-jumping

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$
- SAT solver returns model $[1, 2, 3, \neg 4]$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$
- SAT solver returns model $[1, 2, 3, \neg 4]$
- Theory solver detects $[1, 3, \neg 4]$ T-unsat

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$
- SAT solver returns model $[1, 2, 3, \neg 4]$
- Theory solver detects $[1, 3, \neg 4]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2, \neg 1 \vee \neg 3 \vee 4]$

Adapting DPLL to DPLL(T)

Example

$$\underbrace{g(a) = c}_{1} \wedge (\underbrace{f(g(a)) \neq f(c)}_{\neg 2} \vee \underbrace{g(a) = d}_{3}) \wedge \underbrace{c \neq d}_{\neg 4}$$

- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4]$ (i.e. $[[1], [-2, 3], [-4]]$)
- SAT solver returns model $[1, \neg 2, \neg 4]$
- Theory solver detects $[1, \neg 2]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2]$
- SAT solver returns model $[1, 2, 3, \neg 4]$
- Theory solver detects $[1, 3, \neg 4]$ T-unsat
- Call SAT solver with input $[1, \neg 2 \vee 3, \neg 4, \neg 1 \vee 2, \neg 1 \vee \neg 3 \vee 4]$
- SAT solver detects unsat

Theory combination

- Given a theory T , a Theory solver for T takes as input a set (interpreted as an implicit conjunction) φ of literals and determines whether φ is T -satisfiable.
- We are often interested in using two or more theories at the same time.
- Can we combine two theory solvers to get a theory solver for the combined theory?

Theory combination

- Given a theory T , a Theory solver for T takes as input a set (interpreted as an implicit conjunction) φ of literals and determines whether φ is T -satisfiable.
- We are often interested in using two or more theories at the same time.
- Can we combine two theory solvers to get a theory solver for the combined theory?

Example

The following formula uses both T_E and T_Z

$$\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

The Nelson-Oppen Method

A very general method for combining theory solvers is the **Nelson-Oppen method**.

This method is applicable when:

1. The signatures Σ_i are disjoint.
2. The theories T_i are stably-infinite.
 - A Σ -theory T is **stably-infinite** if every T -satisfiable quantifier-free Σ -formula is satisfiable in an infinite model.
3. The formulas to be tested for satisfiability are **conjunctions of quantifier-free literals**.

Extensions exist that can relax each of these restrictions in some cases.

The Nelson-Oppen Method

Some definitions:

- A member of Σ_i is an *i-symbol*.
- A term t is an *i-term* if it starts with an *i-symbol*.
- An *atomic i-formula* is
 - an application of an *i-predicate*,
 - an equation whose lhs is an *i-term*, or
 - an equation whose lhs is a variable and whose rhs is an *i-term*
- An *i-literal* is an atomic *i-formula* or the negation of one.
- An occurrence of a term t in either an *i-term* or an *i-literal* is *i-alien* if it is a j -term with $i \neq j$ and all of its super-terms (if any) are *i-terms*.
- An expression is *pure* if it contains only variables and *i-symbols* for some i .

Conversion to separate form

Given a conjunction of literals φ , we want to convert it into a **separate form**: a T -equisatisfiable conjunction of literals $\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n$ where each φ_i is a Σ_i -formula.

We have the following algorithm:

1. Let ψ be **some literal** in φ .
2. If ψ is a **pure i -literal**, for some i , remove ψ from φ and add ψ to φ_i .
If φ is **empty** then **stop**; otherwise **goto step 1**.
3. Otherwise, ψ is an **i -literal** for some i .
Let t be a **term occurring i -alien** in ψ .
Replace t in φ with a **new variable** z and add $z = t$ to φ .
Goto step 1.

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We convert φ to a separate form:

- $\varphi_E := ?$
- $\varphi_Z := ?$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We convert φ to a separate form:

- $\varphi_E := ?$
- $\varphi_Z := 1 \leq x$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x} \wedge \cancel{x \leq 2} \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We convert φ to a separate form:

- $\varphi_E := ?$
- $\varphi_Z := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2} \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We convert φ to a separate form:

- $\varphi_E := ?$
- $\varphi_Z := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2} \wedge f(x) \neq f(y) \wedge f(x) \neq f(2) \wedge y = 1$$

We convert φ to a separate form:

- $\varphi_E := ?$
- $\varphi_Z := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2} \wedge f(x) \neq f(y) \wedge f(x) \neq f(2) \wedge y = 1$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2} \wedge f(x) \neq f(y) \wedge \cancel{f(x) \neq f(2)} \wedge y = 1$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y)$
- $\varphi_Z := \cancel{1 \leq x \wedge x \leq 2}$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2} \wedge f(x) \neq f(y) \wedge \cancel{f(x) \neq f(z)} \wedge y = 1 \wedge z = 2$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = \cancel{1 \leq x \wedge x \leq 2} \wedge f(x) \neq f(y) \wedge f(x) \neq f(z) \wedge y = 1 \wedge z = 2$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(y) \wedge f(x) \neq f(z) \wedge y = 1 \wedge z = 2$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(y) \wedge f(x) \neq f(z) \wedge y = 1 \wedge z = 2$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

Conversion to separate form

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi = 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(y) \wedge f(x) \neq f(z) \wedge y = 1 \wedge z = 2$$

We convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

The Nelson-Oppen Method

- As each φ_i is a Σ_i -formula, we can run a Theory solver Sat_i for each φ_i .
- If any Sat_i reports that φ_i is unsatisfiable, then φ is unsatisfiable.
- The converse is not true in general!
- We need a way for the decision procedures to communicate with each other about shared variables.
- If S is a set of terms and \sim is an equivalence relation on S , then the arrangement of S induced by \sim is

$$Ar_{\sim} = \{x = y \mid x \sim y\} \cup \{x \neq y \mid x \not\sim y\}$$

The Nelson-Oppen Method

Suppose that T_1 and T_2 are theories with disjoint signatures Σ_1 and Σ_2 .

Let $T = \bigcup T_i$ and $\Sigma = \bigcup \Sigma_i$.

Given a Σ -formula φ and decision procedures Sat_1 and Sat_2 for T_1 and T_2 , we wish to determine if φ is T -satisfiable.

The non-deterministic Nelson-Oppen algorithm:

1. Convert φ to its **separate form** $\varphi_1 \wedge \varphi_2$.
2. Let S be the **set of variables shared** between φ_1 and φ_2 .
Guess an **equivalence relation** \sim on S .
3. **Run Sat_1** on $\varphi_1 \cup Ar_{\sim}$.
4. **Run Sat_2** on $\varphi_2 \cup Ar_{\sim}$.

The Nelson-Oppen Method

Suppose that T_1 and T_2 are theories with disjoint signatures Σ_1 and Σ_2 .

Let $T = \bigcup T_i$ and $\Sigma = \bigcup \Sigma_i$.

Given a Σ -formula φ and decision procedures Sat_1 and Sat_2 for T_1 and T_2 , we wish to determine if φ is T -satisfiable.

The non-deterministic Nelson-Oppen algorithm:

1. Convert φ to its **separate form** $\varphi_1 \wedge \varphi_2$.
2. Let S be the **set of variables shared** between φ_1 and φ_2 .
Guess an **equivalence relation** \sim on S .
3. **Run Sat_1** on $\varphi_1 \cup Ar_{\sim}$.
4. **Run Sat_2** on $\varphi_2 \cup Ar_{\sim}$.

If there exists an equivalence relation \sim such that both Sat_1 and Sat_2 succeed, then φ is **T -satisfiable**.

If no such equivalence relation exists, then φ is **T -unsatisfiable**.

The Nelson-Oppen Method

Suppose that T_1 and T_2 are theories with disjoint signatures Σ_1 and Σ_2 .

Let $T = \bigcup T_i$ and $\Sigma = \bigcup \Sigma_i$.

Given a Σ -formula φ and decision procedures Sat_1 and Sat_2 for T_1 and T_2 , we wish to determine if φ is T -satisfiable.

The non-deterministic Nelson-Oppen algorithm:

1. Convert φ to its **separate form** $\varphi_1 \wedge \varphi_2$.
2. Let S be the **set of variables shared** between φ_1 and φ_2 .
Guess an **equivalence relation** \sim on S .
3. **Run Sat_1** on $\varphi_1 \cup Ar_\sim$.
4. **Run Sat_2** on $\varphi_2 \cup Ar_\sim$.

If there exists an equivalence relation \sim such that both Sat_1 and Sat_2 succeed, then φ is **T -satisfiable**.

If no such equivalence relation exists, then φ is **T -unsatisfiable**.

The generalization to more than two theories is straightforward.

The Nelson-Oppen Method

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We first convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

The Nelson-Oppen Method

Example

Consider the following $\Sigma_E \cup \Sigma_Z$:

$$\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

We first convert φ to a separate form:

- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

The shared variables are $\{x, y, z\}$.

There are 5 possible arrangements based on equivalence classes of x, y , and z (see *Bell number*).

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
 - $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
 - $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$
-
1. $\{x = y, x = z, y = z\}$
 2. $\{x = y, x \neq z, y \neq z\}$
 3. $\{x \neq y, x = z, y \neq z\}$
 4. $\{x \neq y, x \neq z, y = z\}$
 5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_E
2. $\{x = y, x \neq z, y \neq z\}$
3. $\{x \neq y, x = z, y \neq z\}$
4. $\{x \neq y, x \neq z, y = z\}$
5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_E
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_E
3. $\{x \neq y, x = z, y \neq z\}$
4. $\{x \neq y, x \neq z, y = z\}$
5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_E
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_E
3. $\{x \neq y, x = z, y \neq z\}$ inconsistent with T_E
4. $\{x \neq y, x \neq z, y = z\}$
5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_E
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_E
3. $\{x \neq y, x = z, y \neq z\}$ inconsistent with T_E
4. $\{x \neq y, x \neq z, y = z\}$ inconsistent with T_Z
5. $\{x \neq y, x \neq z, y \neq z\}$

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_E
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_E
3. $\{x \neq y, x = z, y \neq z\}$ inconsistent with T_E
4. $\{x \neq y, x \neq z, y = z\}$ inconsistent with T_Z
5. $\{x \neq y, x \neq z, y \neq z\}$ inconsistent with T_Z

The Nelson-Oppen Method

Example

- $\varphi := 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$
- $\varphi_E := f(x) \neq f(y) \wedge f(x) \neq f(z)$
- $\varphi_Z := 1 \leq x \wedge x \leq 2 \wedge y = 1 \wedge z = 2$

1. $\{x = y, x = z, y = z\}$ inconsistent with T_E
2. $\{x = y, x \neq z, y \neq z\}$ inconsistent with T_E
3. $\{x \neq y, x = z, y \neq z\}$ inconsistent with T_E
4. $\{x \neq y, x \neq z, y = z\}$ inconsistent with T_Z
5. $\{x \neq y, x \neq z, y \neq z\}$ inconsistent with T_Z

Conclusion: φ is $T_E \cup T_Z$ -unsatisfiable!

Recall the ingredients:

- Theory solvers for different theories
- Combine a Theory solver and a SAT solver
- Combine Theory solvers for different theories

Symbolic execution

Symbolic execution

- Symbolic execution is widely used in practice.
- Tools based on symbolic execution have found serious errors and security vulnerabilities in various systems:
 - Networks servers
 - File systems
 - Device drivers
 - Unix utilities
 - Computer vision code
 - ...

Symbolic execution: Tools

- Stanford's KLEE
<http://klee.llvm.org/>
- Nasa's Java PathFinder
<http://javapathfinder.sourceforge.net/>
- Microsoft Research's SAFE
- UC Berkeley's CUTE

Symbolic execution

At any point during program execution, **symbolic execution** keeps two formulas:

- **symbolic store** and
- **path constraint**

Therefore, at any point in time the **symbolic state** is described as the conjunction of these two formulas.

Symbolic store

The **value of variables** at any moment in time are given by a function

$$\sigma_s : \textit{Var} \rightarrow \textit{Sym}$$

- \textit{Var} is the set of variables
- \textit{Sym} is a set of **symbolic values**
- σ_s is called a **symbolic store**

Example

$$\sigma_s : \textit{Var} \mapsto \textit{Sym}, \quad x \mapsto x0, \quad y \mapsto y0$$

Arithmetic expression evaluation simply manipulates the symbolic values.

Example

Suppose the symbolic store is $\sigma_s : x \mapsto x0, y \mapsto y0$.

Then $z = x + y$ will produce the new symbolic store

$$\sigma'_s : x \mapsto x0, y \mapsto y0, z \mapsto x0 + y0$$

We literally keep the **symbolic expression** $x0 + y0$.

Path constraint

- The analysis keeps a **path constraint** (*pct*) which records the history of all branches taken so far.
- The path constraint is simply a **formula**.
- The formula is typically in a decidable logical fragment without quantifiers.
- At the start of the analysis, the path constraint is **true**.
- Evaluation of **conditionals** affects the path constraint, but not the symbolic store.

Path constraint

Example

Suppose the symbolic store is $\sigma_s : x \mapsto x_0, y \mapsto y_0$.

Suppose the path constraint is $pct = x_0 > 10$.

Let us evaluate `if(x > y + 1) {5: ...}`

At label 5, we will get the symbolic store σ_s . It does not change!

But, at label 5, we will get an updated path constraint:

$$pct = x_0 > 10 \wedge x_0 > y_0 + 1$$

Symbolic execution - example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

Can you find the inputs that make the program reach the ERROR?

Lets execute this example with classic symbolic execution

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution - example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x, y);  
}
```

The `read()` functions read a value from the input and because we don't know what those read values are, we set the values of `x` and `y` to fresh symbolic values called `x0` and `y0`

`pct` is true because so far we have not executed any conditionals

$$\sigma_s : \begin{array}{l} x \mapsto x0, \\ y \mapsto y0 \end{array} \quad pct : \text{true}$$

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution - example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

$$\sigma_s : \begin{aligned} x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0 \end{aligned}$$

pct : true

Here, we simply executed the function `twice()` and added the new symbolic value for `z`.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution - example

```
int twice(int v) {  
    return 2 * v;  
}
```

```
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}
```

```
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

We forked the analysis into 2 paths: the true and the false path. So we **duplicate** the state of the analysis.

This is the result if $x = z$:

$$\sigma_s : \begin{array}{l} x \mapsto x_0, \\ y \mapsto y_0 \\ z \mapsto 2*y_0 \end{array}$$

$$pct : x_0 = 2*y_0$$

This is the result if $x \neq z$:

$$\sigma_s : \begin{array}{l} x \mapsto x_0, \\ y \mapsto y_0 \\ z \mapsto 2*y_0 \end{array}$$

$$pct : x_0 \neq 2*y_0$$

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution - example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

We can avoid further exploring a path if we know the constraint `pct` is **unsatisfiable**. In this example, both `pct`'s are **satisfiable** so we need to keep exploring both paths.

This is the result if $x = z$:

$$\sigma_s : \begin{aligned} x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0 \end{aligned}$$

$$pct : x_0 = 2*y_0$$

This is the result if $x \neq z$:

$$\sigma_s : \begin{aligned} x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0 \end{aligned}$$

$$pct : x_0 \neq 2*y_0$$

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution - example

```
int twice(int v) {  
    return 2 * v;  
}
```

```
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}
```

```
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

Lets explore the path when $x == z$ is true.
Once again we get 2 more paths.

This is the result if $x > y + 10$:

$$\begin{aligned}\sigma_s : \quad x &\mapsto x_0, \\ &y \mapsto y_0 \\ &z \mapsto 2*y_0\end{aligned}$$

$$\begin{aligned}pct : \quad x_0 &= 2*y_0 \\ &\wedge \\ &x_0 > y_0+10\end{aligned}$$

This is the result if $x \leq y + 10$:

$$\begin{aligned}\sigma_s : \quad x &\mapsto x_0, \\ &y \mapsto y_0 \\ &z \mapsto 2*y_0\end{aligned}$$

$$\begin{aligned}pct : \quad x_0 &= 2*y_0 \\ &\wedge \\ &x_0 \leq y_0+10\end{aligned}$$

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution - example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

So the following path reaches “**ERROR**”.

This is the result if $x > y + 10$:

$$\sigma_s : \begin{aligned} x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0 \end{aligned}$$

$$\text{pct} : \begin{aligned} x_0 &= 2*y_0 \\ &\wedge \\ x_0 &> y_0+10 \end{aligned}$$

We can now ask the SMT solver for a satisfying assignment to the pct formula.

For instance, $x_0 = 40$, $y_0 = 20$ is a satisfying assignment. That is, running the program with those concrete inputs triggers the error.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Handling Loops - a limitation

```
int F(unsigned int k) {  
    int sum = 0;  
    int i = 0;  
    for ( ; i < k; i++)  
        sum += i;  
    return sum;  
}
```

- A serious limitation of symbolic execution is [handling unbounded loops](#).
- Symbolic execution runs the program for a finite number of paths.
- But what happens if we do not know the bound on a loop?
- [The symbolic execution will keep running forever!](#)

Handling Loops - bound loops

```
int F(unsigned int k) {  
    int sum = 0;  
    int i = 0;  
    for ( ; i < 2; i++)  
        sum += i;  
    return sum;  
}
```

- A common solution in practice is to **provide some loop bound**.
 - In the above example, we can bound **k** to say 2.
 - This is an example of an **under-approximation**
- Practical symbolic analyzers usually under-approximate as most programs have unknown loop bounds.

Handling Loops - loop invariants

```
int F(unsigned int k) {  
    int sum = 0;  
    int i = 0;  
    for ( ; i < k; i++)  
        sum += i;  
    return sum;  
}
```

loop invariant

- Another solution is to **provide a loop invariant**.
- This technique is rarely used for large programs because it is **difficult to provide** such invariants manually.
- It can also lead to **over-approximation**.

Constraint solving - challenges

- Constraint solving is fundamental to symbolic execution.
- An SMT solver is continuously invoked during analysis.
- Often, the main **roadblock to performance** of symbolic execution engines is the time spent in constraint solving.
- Important features:
 - The SMT solver supports as **many decidable logical fragments** as possible.
 - Some tools use more than one SMT solver.
 - The SMT solver can **solve large formulas quickly**.
 - The symbolic execution engines tries to reduce the burden in calling the SMT solver by **exploring domain specific insights**.

Key optimization - caching

- The analyzer will invoke the SMT solver with **similar formulas**.
- The symbolic execution engine can keep a **map (cache)** of formulas to a satisfying assignment for the formulas.
- When the engine builds a new formula and would like to find a satisfying assignment for that formula, it can **first access the cache**, before calling the SMT solver.

Key optimization - caching

Example

Suppose the cache contains the mapping:

Formula	Solution
$(x + y < 10) \wedge (x > 5)$	$\rightarrow \{x = 6, y = 3\}$

If we get a **weaker formula** as a query, say $(x + y < 10)$, then we can immediately **reuse the solution already found in the cache**, without calling the SMT solver.

If we get a **stronger formula** as a query, say $(x + y < 10) \wedge (x > 5) \wedge (y \geq 0)$, then we can quickly **try the solution in the cache** and see if it works, without calling the solver (in this example, it works).

When constraint solving fails

Despite best efforts, the program may be using constraints in a fragment which the SMT solver does not handle (well).

For example, the SMT solver does not handle [non-linear constraints](#) well.

When constraint solving fails - example

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

Here, we changed the `twice()` function to contain a **non-linear** result.

Let us see what happens when we symbolically execute the program now...

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

When constraint solving fails - example

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

This is the result if $x = z$:

$$\begin{aligned}\sigma_s : \quad x &\mapsto x_0, \\ &y \mapsto y_0 \\ &z \mapsto y_0 * y_0\end{aligned}$$

$$pct : x_0 = y_0 * y_0$$

Now, if we are to invoke the SMT solver with the pct formula, it would be **unable** to compute satisfying assignments, precluding us from knowing whether the path is feasible or not.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

References

- Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.
- Lecture Notes on "Techniques for Program Analysis and Verification", Stanford, Clark Barrett.
- Lecture Notes on "Program verification", ETH Zurich, Alexander Summers.
- Lecture Notes on "Computer-Aided Reasoning for Software Engineering", University of Washington, Emina Torlak.

C09 – Concolic execution & Model checking

Program Verification

FMI · Denisa Diaconescu · Spring 2022

Overview

Concolic execution

What is Model Checking?

Concolic execution

Concolic execution

concolic = concrete + symbolic

- Combines both symbolic execution and concrete (normal) execution.
- The basic idea is to have the concrete execution drive the symbolic execution.
- The programs run as usual (it needs to be given some input), but in addition it also maintains the usual symbolic information.

Concolic execution - Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

The `read()` functions read a value from the input. Suppose we read $x = 22$ and $y = 7$.

We will keep both the concrete store and the symbolic store and path constraint.

$$\sigma : x \mapsto 22, \\ y \mapsto 7$$

$$\sigma_s : x \mapsto x_0, \\ y \mapsto y_0$$

pct : true

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Concolic execution - Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

$$\sigma : x \mapsto 22,
y \mapsto 7,
z \mapsto 14$$

$$\sigma_s : x \mapsto x_0,
y \mapsto y_0
z \mapsto 2*y_0$$

pct : true

The concrete execution will now take the 'else' branch of $z == x$.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Concolic execution - Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

Hence, we get:

$$\sigma : \begin{aligned} x &\mapsto 22, \\ y &\mapsto 7, \\ z &\mapsto 14 \end{aligned}$$

$$\sigma_s : \begin{aligned} x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0 \end{aligned}$$

$$pct : x_0 \neq 2*y_0$$

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Concolic execution - Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

At this point, concolic execution decides that it would like to explore the “true” branch of $x == z$ and hence it needs to generate concrete inputs in order to explore it. Towards such inputs, it negates the pct constraint, obtaining:

$pct : x_0 = 2 * y_0$

It then calls the SMT solver to find a satisfying assignment of that constraint. Let us suppose the SMT solver returns:

$x_0 \mapsto 2, y_0 \mapsto 1$

The concolic execution then runs the program with this input.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Concolic execution - Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

With the input $x \mapsto 2$, $y \mapsto 1$ we reach this program point with the following information:

$$\begin{aligned}\sigma : x &\mapsto 2, \\ y &\mapsto 1, \\ z &\mapsto 2\end{aligned}$$

$$\begin{aligned}\sigma_s : x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0\end{aligned}$$

$$pct : x_0 = 2*y_0$$

Continuing further we get:

Concolic execution - Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

We reach the “else” branch of $x > y + 10$

$$\begin{aligned}\sigma : x &\mapsto 2, \\ y &\mapsto 1, \\ z &\mapsto 2\end{aligned}$$

$$\begin{aligned}\sigma_s : x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0\end{aligned}$$

$$\begin{aligned}pct : x_0 &= 2*y_0 \\ &\wedge \\ x_0 &\leq y_0+10\end{aligned}$$

Again, concolic execution may want to explore the ‘true’ branch of $x > y + 10$.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Concolic execution - Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

We reach the “else” branch of $x > y + 10$

$$\begin{aligned}\sigma : x &\mapsto 2, \\ y &\mapsto 1, \\ z &\mapsto 2\end{aligned}$$

$$\begin{aligned}\sigma_s : x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto 2*y_0\end{aligned}$$

$$\begin{aligned}pct : x_0 &= 2*y_0 \\ &\wedge \\ x_0 &\leq y_0+10\end{aligned}$$

Concolic execution now negates the conjunct
 $x_0 \leq y_0+10$ obtaining:

$$x_0 = 2*y_0 \wedge x_0 > y_0+10$$

A satisfying assignment is: $x_0 \mapsto 30, y_0 \mapsto 15$

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Concolic execution - Example

```
int twice(int v) {  
    return 2 * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

If we run the program with the input:

$x_0 \mapsto 30, y_0 \mapsto 15$

we will now reach the **ERROR** state.

As we can see from this example, by keeping the symbolic information, the concrete execution can use that information in order to obtain new inputs.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Non-linear constraints - Example

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

Let us again consider our example and see what concolic execution would do with non-linear constraints.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Non-linear constraints - Example

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

The `read()` functions read a value from the input. Suppose we read $x = 22$ and $y = 7$.

$$\sigma : x \mapsto 22, \\ y \mapsto 7$$

$$\sigma_s : x \mapsto x_0, \\ y \mapsto y_0$$

pct : true

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Non-linear constraints - Example

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

$$\sigma : x \mapsto 22,
y \mapsto 7,
z \mapsto 49$$

$$\sigma_s : x \mapsto x_0,
y \mapsto y_0
z \mapsto y_0 * y_0$$

pct : true

The concrete execution will now take
the 'else' branch of $x == z$.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Non-linear constraints - Example

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

Hence, we get:

$$\begin{aligned}\sigma : x &\mapsto 22, \\ y &\mapsto 7, \\ z &\mapsto 49\end{aligned}$$

$$\begin{aligned}\sigma_s : x &\mapsto x_0, \\ y &\mapsto y_0 \\ z &\mapsto y_0 \cdot y_0\end{aligned}$$

$$pct : x_0 \neq y_0 \cdot y_0$$

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Non-linear constraints - Example

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

However, here we have a non-linear constraint $x_0 \neq y_0 * y_0$. If we would like to explore the true branch we negate the constraint, obtaining $x_0 = y_0 * y_0$ but again we have a **non-linear constraint**!

In this case, concolic execution simplifies the constraint by plugging in the concrete values for y_0 in this case, 7, obtaining the simplified constraint:

$$x_0 = 49$$

Hence, it now runs the program with the input

$$x \mapsto 49, \quad y \mapsto 7$$

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Non-linear constraints - Example

```
int twice(int v) {  
    return v * v;  
}  
  
void test(int x, int y) {  
    z = twice(y);  
    if (x == z) {  
        if (x > y + 10)  
            ERROR;  
    }  
}  
  
int main() {  
    x = read();  
    y = read();  
    test(x,y);  
}
```

Running with the input

$x \mapsto 49$, $y \mapsto 7$

will reach the error state.

However, notice that with these inputs, if we try to simplify non-linear constraints by plugging in concrete values (as concolic execution does), then concolic execution we will never reach the else branch of the `if (x > y + 10)` statement.

source: Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.

Symbolic execution

- Is a popular technique for analyzing programs.
 - Completely automated
 - Relies on SMT solvers
- To terminate, may need to bound loops.
 - Leads to under-approximation
- To handle non-linear constraints and external environment, mixes concrete and symbolic execution (concolic execution).

Quiz time!

<https://www.questionpro.com/t/AT4NiZr80r>

What is Model Checking?

The big picture

- Application domain: mostly concurrent, reactive systems
- Verify that a system satisfies a property
 1. Model the system in the model checker's description language.
Call this model M .
 2. Express the property to be verified in the model checker's specification language. Call this formula ϕ .
 3. Run the model checker to show that M satisfies ϕ .
- Automatic for finite-state models

Example

- Two processes executed in parallel
- Each process undergoes transitions $n \rightarrow r \rightarrow c \rightarrow n \rightarrow \dots$ where
 - n denotes "not in critical section"
 - r denotes "requesting to enter critical section"
 - c denotes "critical section"

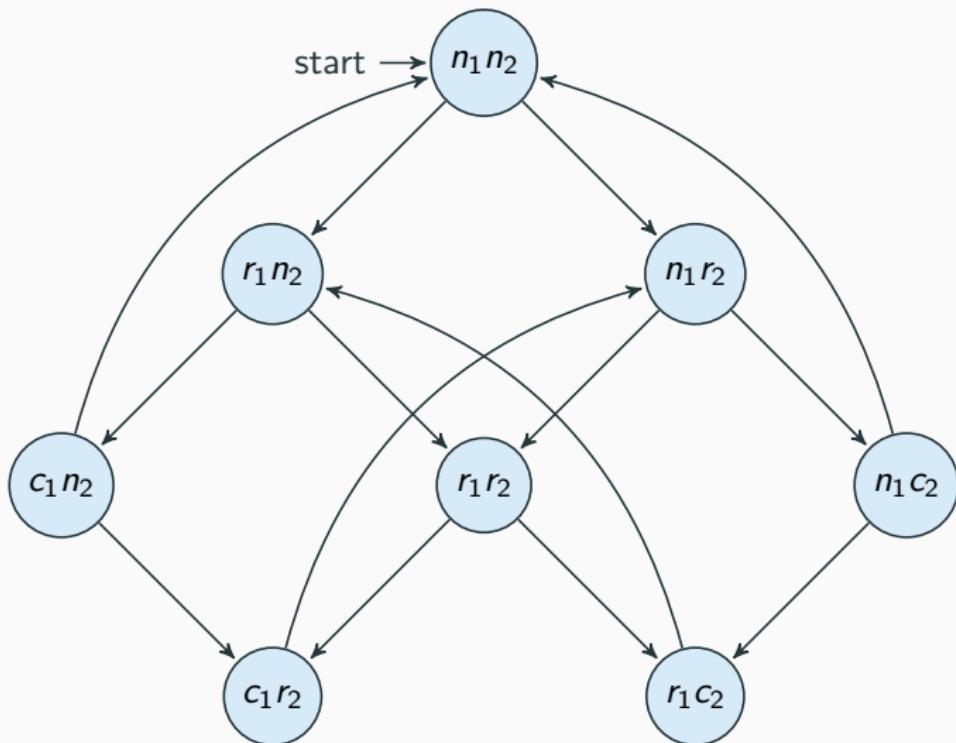
Example

- Two processes executed in parallel
- Each process undergoes transitions $n \rightarrow r \rightarrow c \rightarrow n \rightarrow \dots$ where
 - n denotes "not in critical section"
 - r denotes "requesting to enter critical section"
 - c denotes "critical section"
- Requirements
 - Safety: only one process may execute critical section code at any point
 - Liveness: whenever a process requests to enter its critical section, it will eventually be allowed to do so
 - Non-blocking: a process can always request to enter its critical section

Example (cont.)

- We write a program P to fulfill these requirements. But is it really doing its job?
- We construct a model M for P such that M captures the relevant behaviour of P .

Example (cont.)



Example (cont.)

- Based on a definition of when a model satisfies a property, we can determine whether M satisfies P 's required properties.
- **Example:** M satisfies the **safety** requirement if no state reachable from the start state (including itself) is labeled c_1c_2 . Thus our M satisfies P 's safety requirement.
- **NOTE:** the conclusion that P satisfies these requirements depends on the (unverified) assumption that M is a faithful representation of all the relevant aspects of P .

Uses of Model Checking

Verification of specific properties of

- Hardware circuits
- Communication protocols
- Control software
- Embedded systems
- Device drivers
- ...

Uses of Model Checking

Model Checking has been used to

- Check Microsoft Windows device drivers for bugs
 - The [Static Driver Verifier](#) tool
 - [SLAM](#) project of Microsoft
- The [SPIN](#) tool
 - <http://spinroot.com>
 - <http://spinroot.com/spin/success.html>
 - Flood control barrier control software
 - Parts of *Mars Science Laboratory, Deep Space 1, Cassini, the Mars Exploration Rovers, Deep Impact*
- [PEPA](#) (Performance Evaluation Process Algebra)
 - <http://www.dcs.ed.ac.uk/pepa/>
 - Multiprocess systems
 - Biological systems
- ...

The ACM Turing Award in 2007



Edmund Clarke



Allen Emerson



Joseph Sifakis

*"for their role in developing Model-Checking
into a highly effective verification technology
that is widely adopted in the hardware and software industries"*

Model Checking - Models

A model of some system has

- A finite set of **states**
- A subset of states considered as the **initial states**
- A **transition relation** which, given a state, describes all the states that can be reached "in one time step"

Model Checking - Models

A model of some system has

- A finite set of **states**
- A subset of states considered as the **initial states**
- A **transition relation** which, given a state, describes all the states that can be reached "in one time step"

Refinements of this setup can handle:

- Infinite state spaces
- Continuous state spaces
- Probabilistic Transitions
- ...

Model Checking - Models

Models are always abstraction of reality.

- We must choose what to model and what not to model
- There are limitations forced by the formalism
 - e.g., here we are limited to finite state models
- There will be things we do not understand sufficiently to model
 - e.g., people



Source: *La trahison des images* by René Magritte. Licensed under Fair use via Wikipedia
<http://en.wikipedia.org/wiki/File:MagrittePipe.jpg#mediaviewer/File:MagrittePipe.jpg>

Model Checking - Specifications

We are interested in specifying behaviours of systems over time (use [Temporal Logic](#)).

Specifications are built from

- primitive properties of individual states
 - e.g., *is on, is off, is active, is reading*
- propositional connectives $\wedge, \vee, \neg, \rightarrow$
- temporal connectives
 - e.g., *At all times, the system is not simultaneously reading and writing.*
 - e.g., *If a request signal is asserted at some time, a corresponding grant signal will be asserted within 10 time units.*

Model Checking - Specifications

The exact set of temporal connectives differs across temporal logics.

Logics can differ in how they treat time:

- Continuous time vs. Discrete time
- Linear Time vs. Branching time
- ...

Linear vs. Branching Time

Linear Time

- Considers **paths** (sequences of states)
- Questions of the form
 - *For all paths, does some path property hold?*
 - *Does there exist a path such that some path property holds?*

Linear vs. Branching Time

Linear Time

- Considers **paths** (sequences of states)
- Questions of the form
 - *For all paths, does some path property hold?*
 - *Does there exist a path such that some path property holds?*

Branching Time

- Considers **trees** of possible future states from each initial state
- Questions can become more complex
 - *For all states reachable from an initial state, does there exist an onwards path to a state satisfying some property?*

References

- Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.
- Lecture Notes on "Techniques for Program Analysis and Verification", Stanford, Clark Barrett.
- Lecture Notes on "Program verification", ETH Zurich, Alexander Summers.
- Lecture Notes on "Computer-Aided Reasoning for Software Engineering", University of Washington, Emin Török.

C10 – Model checking

Program Verification

FMI · Denisa Diaconescu · Spring 2022

Overview

Model Checking

LTL Logic

CTL Logic

Model checking algorithm for CTL

Model Checking

The big picture

- Application domain: mostly concurrent, reactive systems
- Verify that a system satisfies a property
 1. Model the system in the model checker's description language.
Call this model M .
 2. Express the property to be verified in the model checker's specification language. Call this formula ϕ .
 3. Run the model checker to show that M satisfies ϕ .
- Automatic for finite-state models

Model Checking - Models

A model of some system has

- A finite set of **states**
- A subset of states considered as the **initial states**
- A **transition relation** which, given a state, describes all the states that can be reached "in one time step"

Model Checking - Specifications

We are interested in specifying behaviours of systems over time (use [Temporal Logic](#)).

Specifications are built from

- primitive properties of individual states
 - e.g., *is on, is off, is active, is reading*
- propositional connectives $\wedge, \vee, \neg, \rightarrow$
- temporal connectives
 - e.g., *At all times, the system is not simultaneously reading and writing.*
 - e.g., *If a request signal is asserted at some time, a corresponding grant signal will be asserted within 10 time units.*

Linear vs. Branching Time

Linear Time

- Considers **paths** (sequences of states)
- Questions of the form
 - *For all paths, does some path property hold?*
 - *Does there exist a path such that some path property holds?*

Branching Time

- Considers **trees** of possible future states from each initial state
- Questions can become more complex
 - *For all states reachable from an initial state, does there exist an onwards path to a state satisfying some property?*

LTL Logic

Syntax

LTL = Linear(-time) Temporal Logic

Assume some set *Atoms* of atomic propositions.

LTL formulas are defined by:

$$\varphi := p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \mathbf{X} \varphi \mid \mathbf{F} \varphi \mid \mathbf{G} \varphi \mid \varphi \mathbf{U} \psi$$

Pronunciation:

- $\mathbf{X} \varphi$ – neXt φ
- $\mathbf{F} \varphi$ – Future φ
- $\mathbf{G} \varphi$ – Globally φ
- $\varphi \mathbf{U} \psi$ – φ Until ψ

Precedence high-to-low:

- $\mathbf{X}, \mathbf{F}, \mathbf{G}, \neg$
- \mathbf{U}
- \wedge, \vee
- \rightarrow

Example

The following are LTL formulas:

- $\mathbf{F} p \wedge \mathbf{G} q \rightarrow p \mathbf{U} r$
- $\mathbf{F}(p \rightarrow \mathbf{G} r) \vee \neg q \mathbf{U} p$
- $p \mathbf{U} (q \mathbf{U} r)$
- $\mathbf{G} \mathbf{F} p \rightarrow \mathbf{F}(q \vee s)$

Syntax

Example

The following are LTL formulas:

- $\mathbf{F} p \wedge \mathbf{G} q \rightarrow p \mathbf{U} r$
- $\mathbf{F}(p \rightarrow \mathbf{G} r) \vee \neg q \mathbf{U} p$
- $p \mathbf{U} (q \mathbf{U} r)$
- $\mathbf{G} \mathbf{F} p \rightarrow \mathbf{F}(q \vee s)$

The following are **not** LTL formulas:

- $\mathbf{U} r$
- $q \mathbf{G} p$

Informal semantics

LTL formulas are evaluated at a position i along a path π through the system:

- An atomic proposition p holds if p is true in the state at position i .
- The propositional connectives $\neg, \wedge, \vee, \rightarrow$ have their usual meanings.
- Meaning of temporal connectives:
 - $X\varphi$ holds if φ holds at the next position
 - $F\varphi$ holds if there exists a future position where φ holds
 - $G\varphi$ holds if φ holds in all future positions
 - $\varphi U \psi$ holds if there exists a future position where ψ holds, and φ holds for all positions prior to that

A path is a sequence of states connected by transitions.

Example

G invariant

- *invariant* is true for all future positions

Example

G invariant

- *invariant* is true for all future positions

G $\neg(\text{read} \wedge \text{write})$

- In all future positions, it is not the case that *read* and *write*

Example

\mathbf{G} invariant

- *invariant* is true for all future positions

$\mathbf{G} \neg(\text{read} \wedge \text{write})$

- In all future positions, it is not the case that *read* and *write*

$\mathbf{G}(\text{request} \rightarrow \mathbf{F} \text{ grant})$

- At every point in the future, a *request* implies that there exists a future point where *grant* holds.

Example

$G(request \rightarrow (request \mathbf{U} grant))$

- At every point in the future, a *request* implies that there exists a future point where *grant* holds, and *request* holds up until that point.

Example

$\mathbf{G}(\mathit{request} \rightarrow (\mathit{request} \mathbf{U} \mathit{grant}))$

- At every point in the future, a *request* implies that there exists a future point where *grant* holds, and *request* holds up until that point.

$\mathbf{G} \mathbf{F} \mathit{enabled}$

- In all future positions, there is a future position where *enabled* holds.

Example

$G(request \rightarrow (request \mathbf{U} grant))$

- At every point in the future, a *request* implies that there exists a future point where *grant* holds, and *request* holds up until that point.

$G F enabled$

- In all future positions, there is a future position where *enabled* holds.

$F G enabled$

- There is a future position, from which all future positions have *enabled* holding.

Transition Systems and Paths

A transition system (or model) $\mathcal{M} = (S, \rightarrow, L)$ consists of

- S a finite set of states
- $\rightarrow \subseteq S \times S$ a transition relation
- $L : S \rightarrow \mathcal{P}(\text{Atoms})$ a labelling function

such that for all $s_1 \in S$ there exists $s_2 \in S$ with $s_1 \rightarrow s_2$ (*serial condition*).

Note:

- *Atoms* is a fixed set of atomic propositions and $\mathcal{P}(\text{Atoms})$ is the powerset of *Atoms*. Thus, $L(s)$ is just the set of atomic propositions that are true in state s .

Transition Systems and Paths

A **transition system** (or model) $\mathcal{M} = (S, \rightarrow, L)$ consists of

- S a **finite set of states**
- $\rightarrow \subseteq S \times S$ a **transition relation**
- $L : S \rightarrow \mathcal{P}(\text{Atoms})$ a **labelling function**

such that for all $s_1 \in S$ there exists $s_2 \in S$ with $s_1 \rightarrow s_2$ (*serial condition*).

Note:

- *Atoms* is a fixed set of atomic propositions and $\mathcal{P}(\text{Atoms})$ is the powerset of *Atoms*. Thus, $L(s)$ is just the set of atomic propositions that are true in state s .

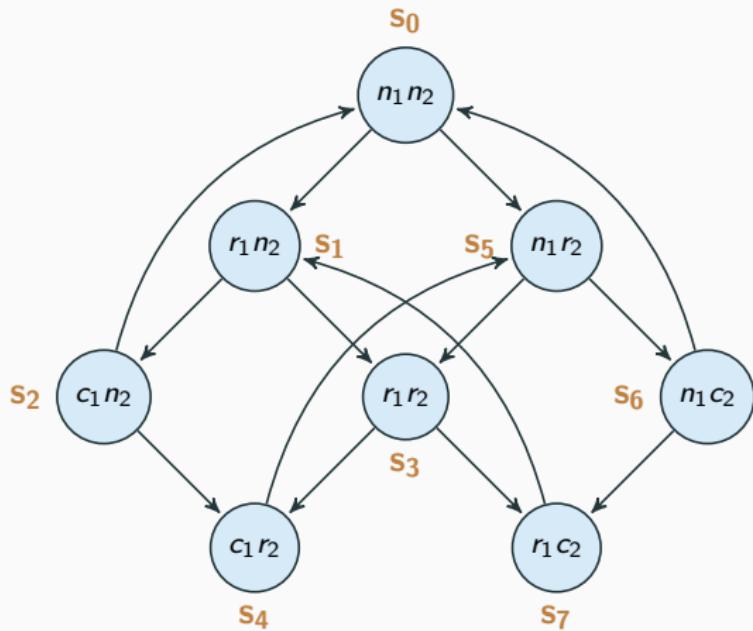
A **path** π in a transition system $\mathcal{M} = (S, \rightarrow, L)$ is an infinite sequence of states s_0, s_1, s_2, \dots such that for all $i \geq 0$, $s_i \rightarrow s_{i+1}$.

Paths are written as $\pi = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$

We write π^i for the suffix starting at s_i . For example, π^3 is $s_3 \rightarrow s_4 \rightarrow \dots$

Transition Systems and Paths

Example



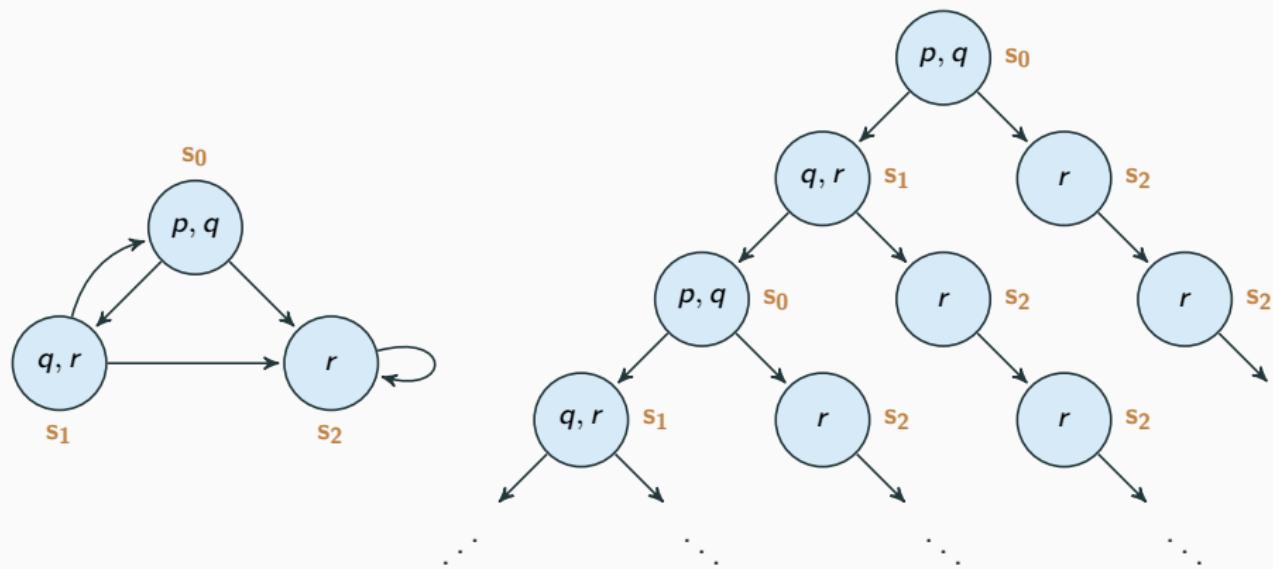
$Atoms = \{n_1, n_2, r_1, r_2, c_1, c_2\}$

$\mathcal{M} = (S, \rightarrow, L)$ where

- $S = \{s_0, s_1, \dots, s_7\}$
- $\rightarrow = \{(s_0, s_1), (s_0, s_5), \dots\}$
- $L(s_0) = \{n_1, n_2\}$
- $L(s_1) = \{r_1, n_2\}$
- \dots

Unwinding a Transition System

Visualise all computational paths from a given state s by **unwinding the transition system** to obtain an infinite computation tree.



Satisfaction relation

Let $\mathcal{M} = (S, \rightarrow, L)$ be a model and $\pi = s_0 \rightarrow s_1 \rightarrow \dots$ be a path in \mathcal{M} .

"The path π satisfies the LTL formula φ "

$$\pi \models \varphi$$

$$\pi \models \top$$

$$\pi \not\models \perp$$

$$\pi \models p \quad \text{iff} \quad p \in L(s_0)$$

$$\pi \models \varphi \wedge \psi \quad \text{iff} \quad \pi \models \varphi \text{ and } \pi \models \psi$$

$$\pi \models \varphi \vee \psi \quad \text{iff} \quad \pi \models \varphi \text{ or } \pi \models \psi$$

$$\pi \models \varphi \rightarrow \psi \quad \text{iff} \quad \pi \models \varphi \text{ implies } \pi \models \psi$$

Satisfaction relation

- | | | |
|--|-----|--|
| $\pi \models \text{X } \varphi$ | iff | $\pi^1 \models \varphi$ |
| $\pi \models \text{F } \varphi$ | iff | there exists $i \geq 0$ such that $\pi^i \models \varphi$ |
| $\pi \models \text{G } \varphi$ | iff | for all $i \geq 0$ we have $\pi^i \models \varphi$ |
| $\pi \models \varphi_1 \text{ U } \varphi_2$ | iff | there exists $i \geq 0$ such that $\pi^i \models \varphi_2$ and
for all $j \in \{0, \dots, i-1\}$ we have $\pi^j \models \varphi_1$ |

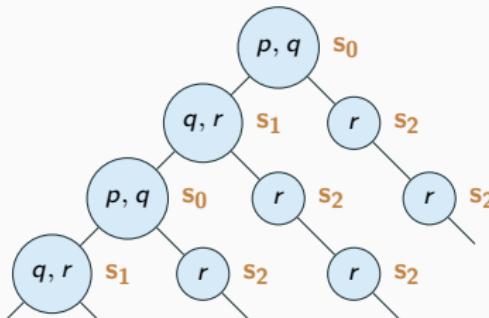
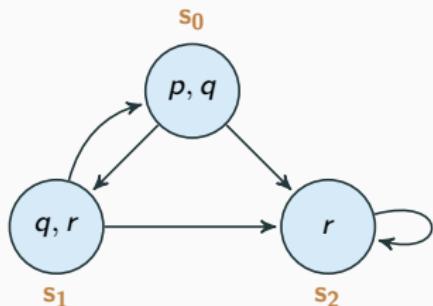
We write

$$\mathcal{M}, s \models \varphi$$

if for every path π of a model \mathcal{M} starting at state s we have $\pi \models \varphi$.

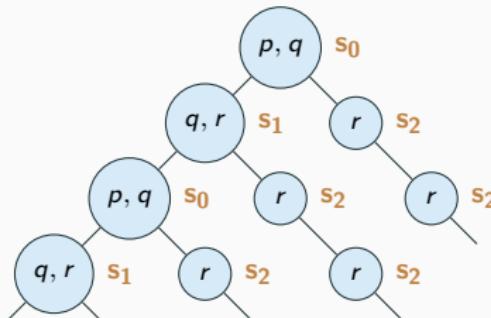
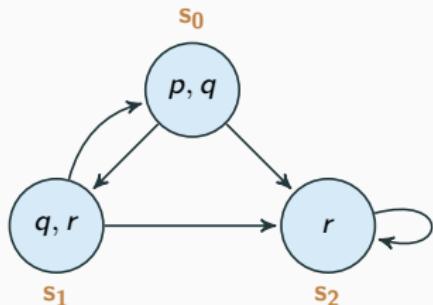
Satisfaction relation

Example



Satisfaction relation

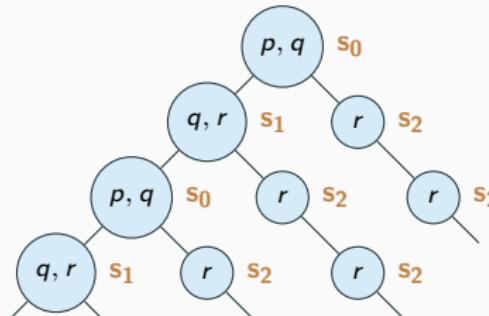
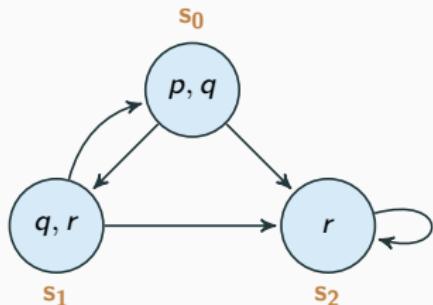
Example



1. $\mathcal{M}, s_0 \models p \wedge q$

Satisfaction relation

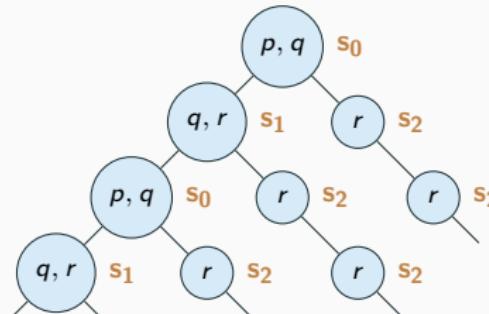
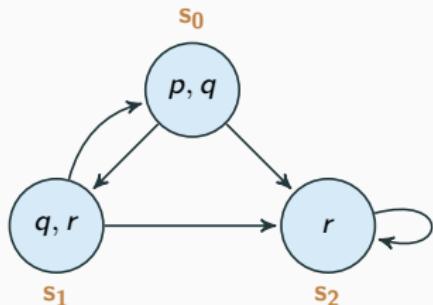
Example



1. $\mathcal{M}, s_0 \models p \wedge q$
2. $\mathcal{M}, s_0 \models \neg r$

Satisfaction relation

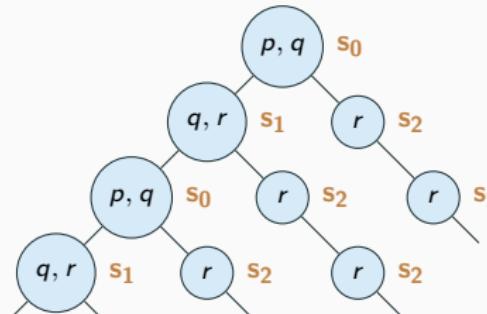
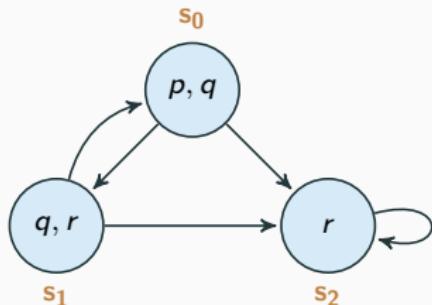
Example



1. $\mathcal{M}, s_0 \models p \wedge q$
2. $\mathcal{M}, s_0 \models \neg r$
3. $\mathcal{M}, s_0 \models \mathbf{X} r$

Satisfaction relation

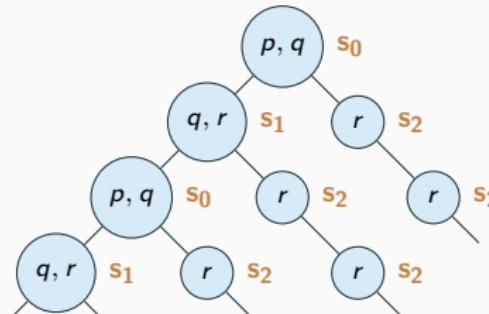
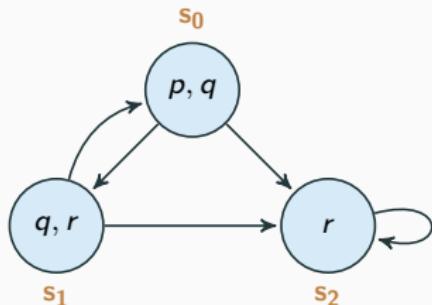
Example



1. $\mathcal{M}, s_0 \models p \wedge q$
2. $\mathcal{M}, s_0 \models \neg r$
3. $\mathcal{M}, s_0 \models \mathbf{X} r$
4. $\mathcal{M}, s_0 \not\models \mathbf{X} (q \wedge r)$

Satisfaction relation

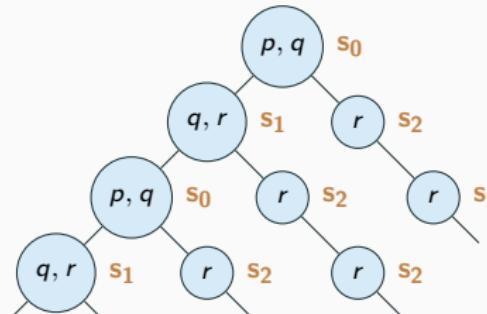
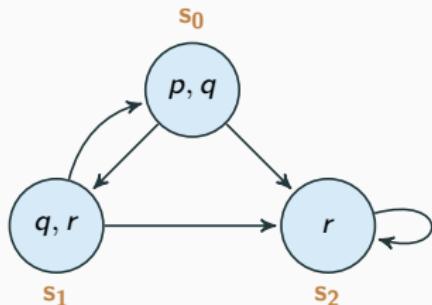
Example



1. $\mathcal{M}, s_0 \models p \wedge q$
 2. $\mathcal{M}, s_0 \models \neg r$
 3. $\mathcal{M}, s_0 \models \textcolor{blue}{X} r$
 4. $\mathcal{M}, s_0 \not\models \textcolor{blue}{X} (q \wedge r)$
 5. $\mathcal{M}, s_0 \models \textcolor{blue}{G} \neg(p \wedge r)$

Satisfaction relation

Example

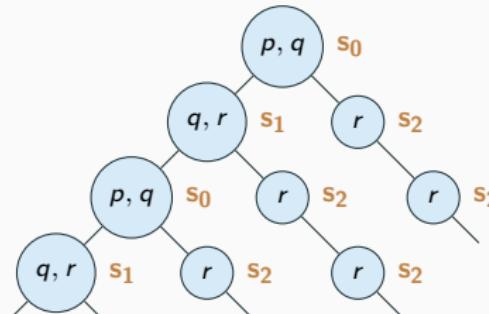
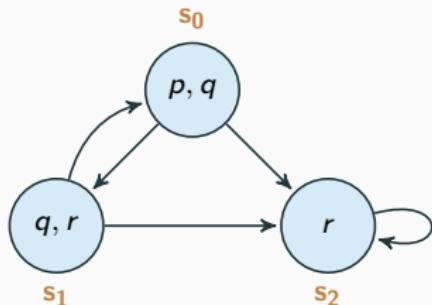


1. $\mathcal{M}, s_0 \models p \wedge q$
2. $\mathcal{M}, s_0 \models \neg r$
3. $\mathcal{M}, s_0 \models \mathbf{X} r$
4. $\mathcal{M}, s_0 \not\models \mathbf{X} (q \wedge r)$
5. $\mathcal{M}, s_0 \models \mathbf{G} \neg(p \wedge r)$

6. $\mathcal{M}, s_2 \models \mathbf{G} r$

Satisfaction relation

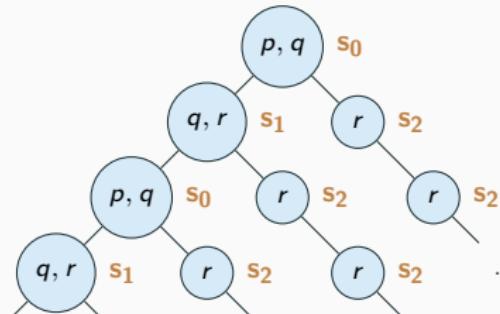
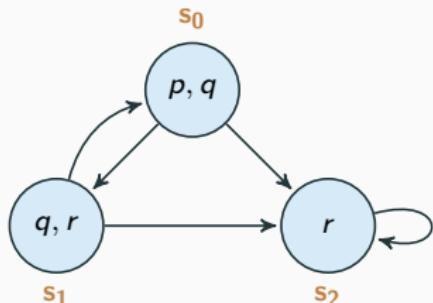
Example



1. $\mathcal{M}, s_0 \models p \wedge q$
2. $\mathcal{M}, s_0 \models \neg r$
3. $\mathcal{M}, s_0 \models \mathbf{X} r$
4. $\mathcal{M}, s_0 \not\models \mathbf{X} (q \wedge r)$
5. $\mathcal{M}, s_0 \models \mathbf{G} \neg(p \wedge r)$
6. $\mathcal{M}, s_2 \models \mathbf{G} r$
7. $\mathcal{M}, s_0 \models \mathbf{F} (\neg q \wedge r) \rightarrow \mathbf{F} \mathbf{G} r$

Satisfaction relation

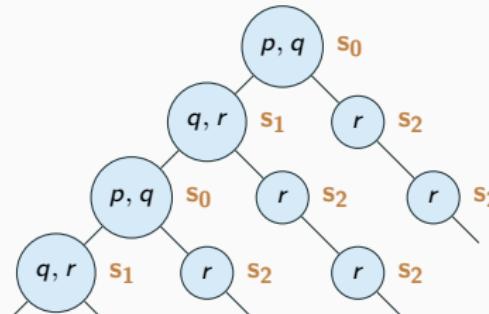
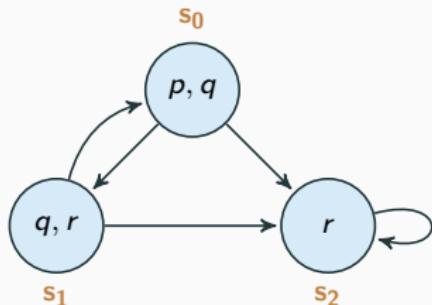
Example



1. $\mathcal{M}, s_0 \models p \wedge q$
2. $\mathcal{M}, s_0 \models \neg r$
3. $\mathcal{M}, s_0 \models \mathbf{X} r$
4. $\mathcal{M}, s_0 \not\models \mathbf{X} (q \wedge r)$
5. $\mathcal{M}, s_0 \models \mathbf{G} \neg(p \wedge r)$
6. $\mathcal{M}, s_2 \models \mathbf{G} r$
7. $\mathcal{M}, s_0 \models \mathbf{F} (\neg q \wedge r) \rightarrow \mathbf{F} \mathbf{G} r$
8. $\mathcal{M}, s_0 \not\models \mathbf{G} \mathbf{F} p$

Satisfaction relation

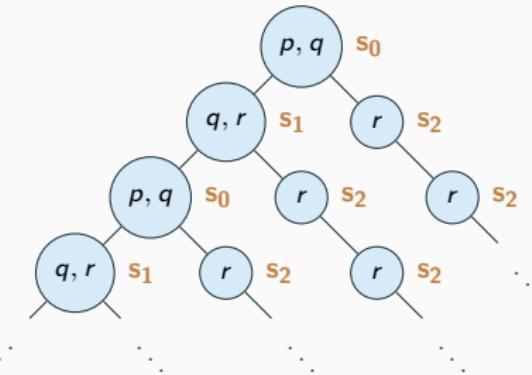
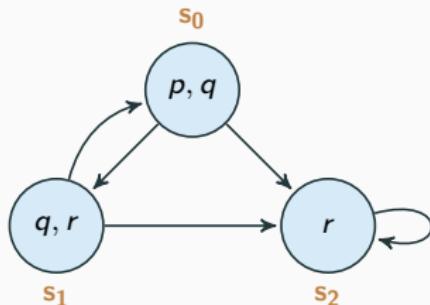
Example



1. $\mathcal{M}, s_0 \models p \wedge q$
2. $\mathcal{M}, s_0 \models \neg r$
3. $\mathcal{M}, s_0 \models \mathbf{X} r$
4. $\mathcal{M}, s_0 \not\models \mathbf{X} (q \wedge r)$
5. $\mathcal{M}, s_0 \models \mathbf{G} \neg(p \wedge r)$
6. $\mathcal{M}, s_2 \models \mathbf{G} r$
7. $\mathcal{M}, s_0 \models \mathbf{F} (\neg q \wedge r) \rightarrow \mathbf{F} \mathbf{G} r$
8. $\mathcal{M}, s_0 \not\models \mathbf{G} \mathbf{F} p$
9. $\mathcal{M}, s_0 \models \mathbf{G} \mathbf{F} p \rightarrow \mathbf{G} \mathbf{F} r$

Satisfaction relation

Example



1. $\mathcal{M}, s_0 \models p \wedge q$
2. $\mathcal{M}, s_0 \models \neg r$
3. $\mathcal{M}, s_0 \models \mathbf{X} r$
4. $\mathcal{M}, s_0 \not\models \mathbf{X} (q \wedge r)$
5. $\mathcal{M}, s_0 \models \mathbf{G} \neg(p \wedge r)$
6. $\mathcal{M}, s_2 \models \mathbf{G} r$
7. $\mathcal{M}, s_0 \models \mathbf{F} (\neg q \wedge r) \rightarrow \mathbf{F} \mathbf{G} r$
8. $\mathcal{M}, s_0 \not\models \mathbf{G} \mathbf{F} p$
9. $\mathcal{M}, s_0 \models \mathbf{G} \mathbf{F} p \rightarrow \mathbf{G} \mathbf{F} r$
10. $\mathcal{M}, s_0 \not\models \mathbf{G} \mathbf{F} r \rightarrow \mathbf{G} \mathbf{F} p$

Equivalences

Two formulas are **equivalent**, denoted $\varphi \equiv \psi$, if they are satisfied by the same models.

Equivalences from Propositional Logic:

$$\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi \quad \neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$$

Equivalences from LTL:

$$\neg X\varphi \equiv X\neg\varphi \quad \neg G\varphi \equiv F\neg\varphi \quad \neg F\varphi \equiv G\neg\varphi$$

Distributive laws:

$$G(\varphi \wedge \psi) \equiv G\varphi \wedge G\psi \quad F(\varphi \vee \psi) \equiv F\varphi \vee F\psi$$

Equivalences

Inter-definitions:

$$\mathbf{F}\varphi \equiv \neg \mathbf{G}\neg\varphi \quad \mathbf{G}\varphi \equiv \neg \mathbf{F}\neg\varphi \quad \mathbf{F}\varphi \equiv \top \mathbf{U} \varphi$$

Idempotency:

$$\mathbf{FF}\varphi \equiv \mathbf{F}\varphi \quad \mathbf{GG}\varphi \equiv \mathbf{G}\varphi$$

Some more surprising equivalences:

$$\mathbf{GFG}\varphi \equiv \mathbf{FG}\varphi \quad \mathbf{FGF}\varphi \equiv \mathbf{GF}\varphi \quad \mathbf{G}(\mathbf{F}\varphi \vee \mathbf{F}\varphi) \equiv \mathbf{GF}\varphi \vee \mathbf{GF}\psi$$

CTL Logic

Syntax

CTL = Computation Tree Logic

Assume some set *Atoms* of atomic propositions.

CTL formulas are defined by:

$$\begin{aligned}\varphi := & \ p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \\ & \textbf{AX} \varphi \mid \textbf{EX} \varphi \mid \textbf{AF} \varphi \mid \textbf{EF} \varphi \mid \textbf{AG} \varphi \mid \textbf{EG} \varphi \mid A[\varphi \mathbf{U} \varphi] \mid E[\varphi \mathbf{U} \varphi]\end{aligned}$$

Each temporal connective is a pair of a path quantifier:

- **A** – for all paths
- **E** – there exists a path

and an LTL-like temporal operator X, F, G, U .

Precedence (high-to low): (**AX, EX, AF, EF, AG, EG, \neg**), (**AU, EU**), (\wedge, \vee),
 \rightarrow

Example

The following are CTL formulas:

- $\text{AG}(q \rightarrow \text{EG } r)$
- $\text{EF E}[r \text{ U } q]$
- $\text{A}[p \text{ U EF } r]$
- $\text{EF EG } p \rightarrow \text{AF } r$

Example

The following are CTL formulas:

- $\text{AG}(q \rightarrow \text{EG } r)$
- $\text{EF E}[r \text{ U } q]$
- $\text{A}[p \text{ U EF } r]$
- $\text{EF EG } p \rightarrow \text{AF } r$

The following are **not** CTL formulas:

- $\text{EF } F r$
- $\text{A} \neg F \neg p$
- $G[r \text{ U } q]$
- $\text{EF}(r \text{ U } q)$

Transition Systems and Paths

(This is the same as for LTL)

A transition system (or model) $\mathcal{M} = (S, \rightarrow, L)$ consists of

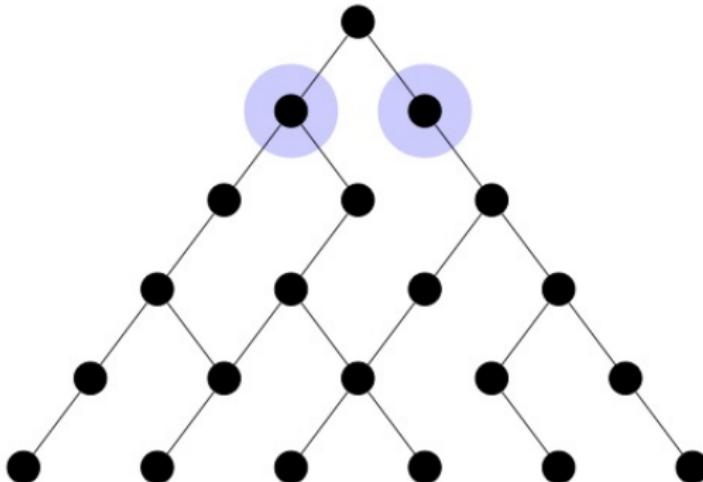
- S a finite set of states
- $\rightarrow \subseteq S \times S$ a transition relation
- $L : S \rightarrow \mathcal{P}(\text{Atoms})$ a labelling function

such that for all $\forall s_1 \in S$ there exists $s_2 \in S$ with $s_1 \rightarrow s_2$ (*serial condition*).

A path π in a transition system $\mathcal{M} = (S, \rightarrow, L)$ is an infinite sequence of states s_0, s_1, s_2, \dots such that for all $i \geq 0$, $s_i \rightarrow s_{i+1}$.

Paths are written as $\pi = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$

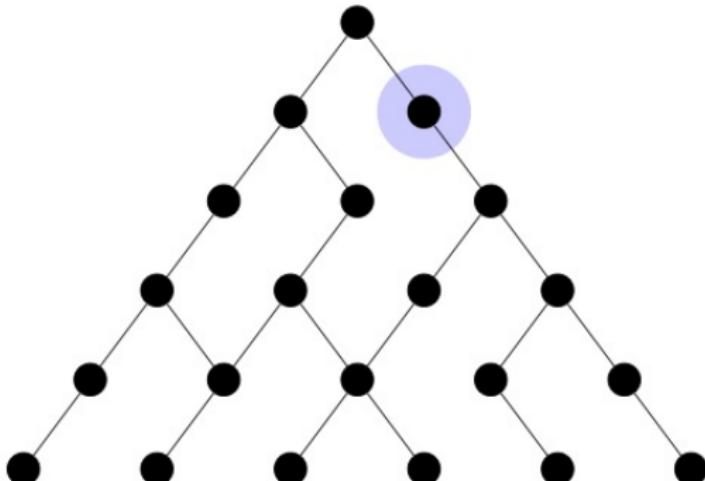
CTL in pictures



$\text{AX } \varphi$

For **every** next state, φ holds.

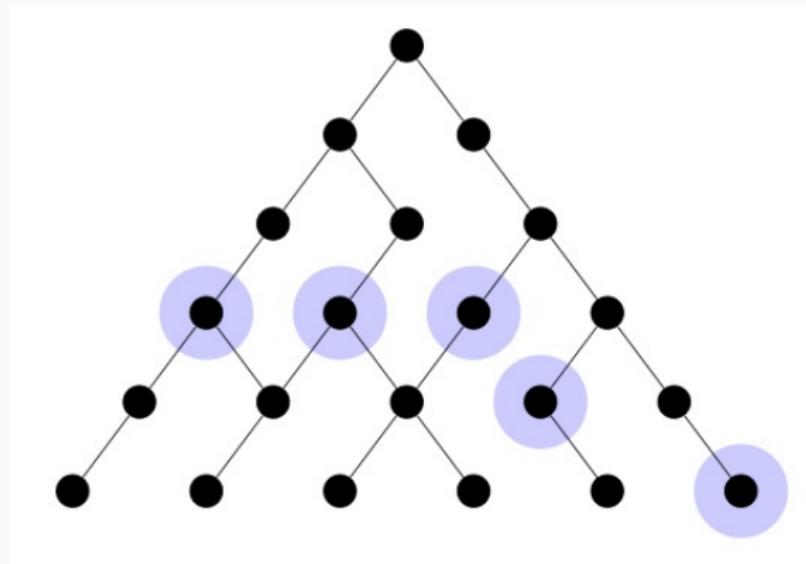
CTL in pictures



$\text{EX } \varphi$

There exists a next state where φ holds.

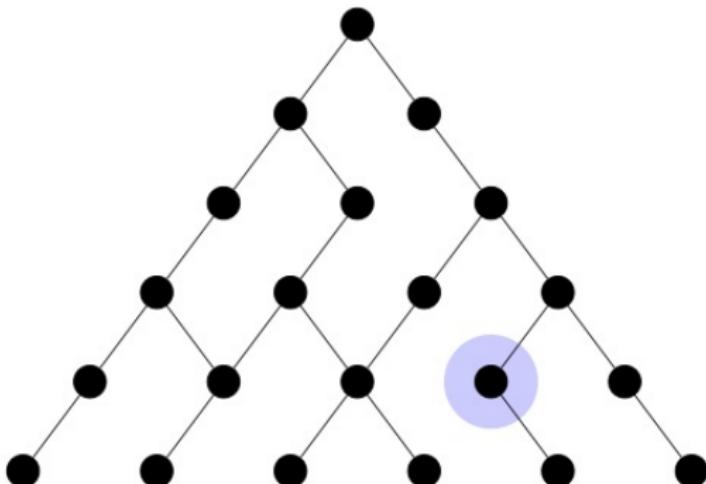
CTL in pictures



AF φ

For all paths, there exists a future state where φ holds.

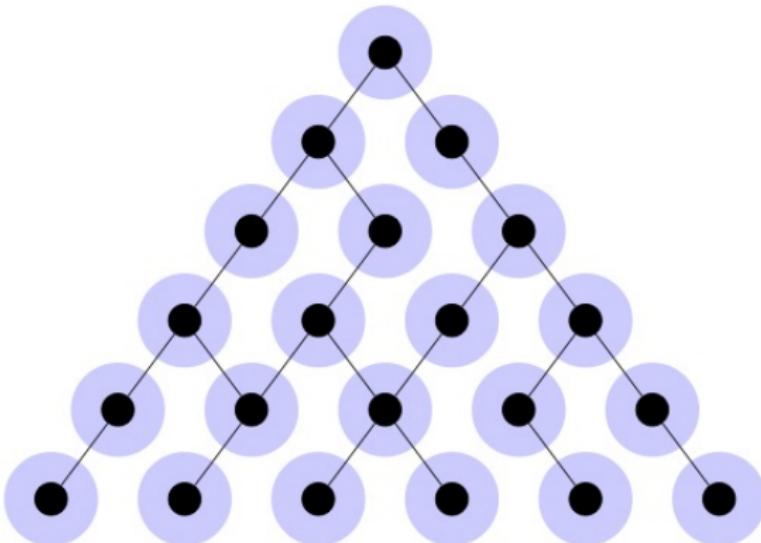
CTL in pictures



EF φ

There exists a path with a future state where φ holds.

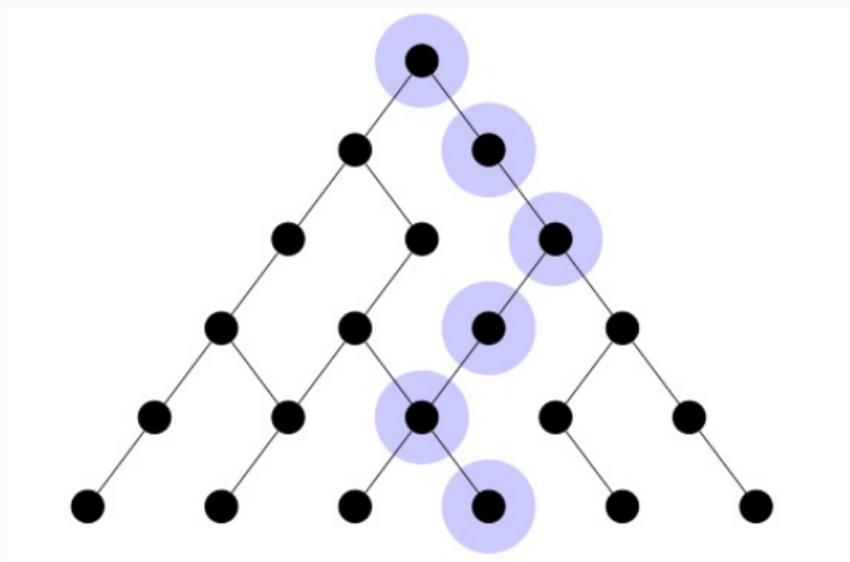
CTL in pictures



AG φ

For all paths, for all states along them, φ holds.

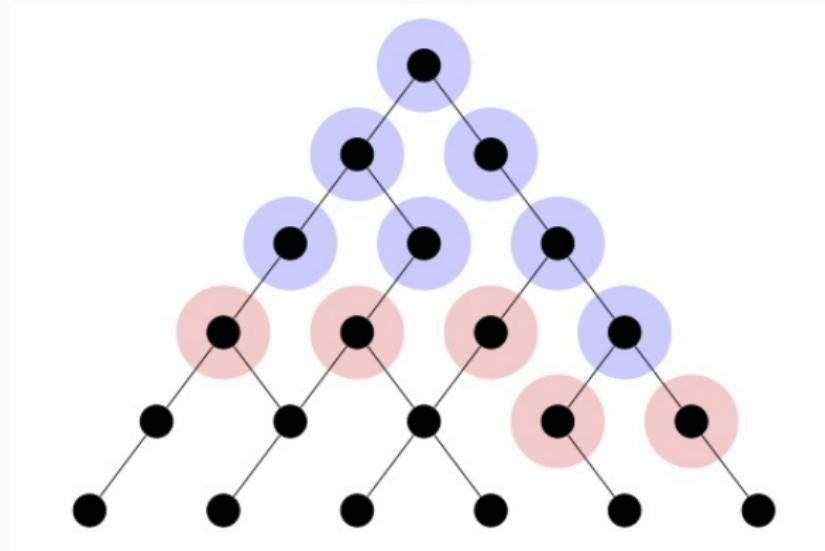
CTL in pictures



EG φ

There exists a path such that, for all states along it, φ holds.

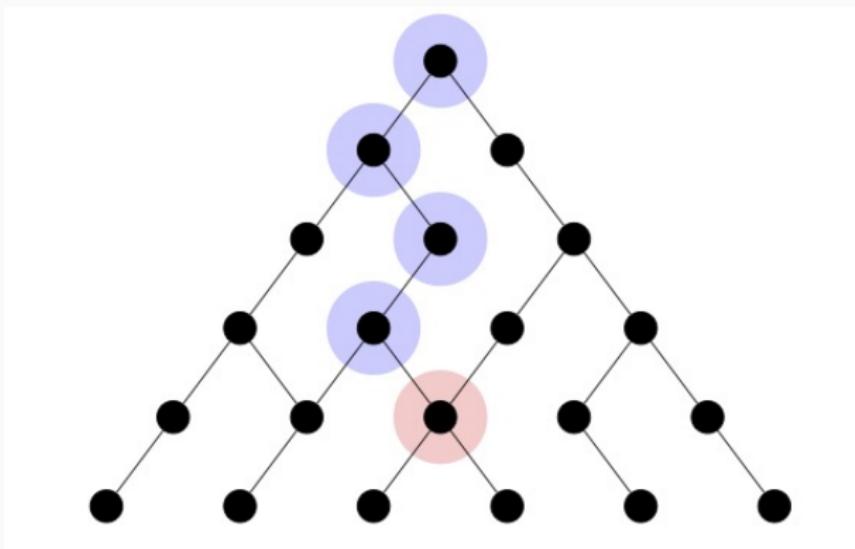
CTL in pictures



$$A[\varphi \mathbf{U} \psi]$$

For all paths, ψ eventually holds, and φ holds at all states earlier.

CTL in pictures



$$E[\varphi \mathbf{U} \psi]$$

There exists a path where ψ eventually holds, and φ holds at all states earlier.

Example

EF φ

- there exists a future state where eventually φ is true

AG AF φ

- for all future states, φ will eventually hold

AG($\varphi \rightarrow$ **AF** ψ)

- for all future states, if φ holds, then ψ will eventually hold

Example

$\mathbf{AG}(\varphi \rightarrow \mathbf{E}[\varphi \mathbf{U} \psi])$

- for all future states, if φ holds, then there is a future where ψ will eventually hold, and φ holds for all points in between

$\mathbf{AG}(\varphi \rightarrow \mathbf{EG} \psi)$

- for all future states, if φ holds, then there is a future where ψ always holds

$\mathbf{EF AG} \varphi$

- there exists a possible state in the future from where φ is always true

Satisfaction Relation

"The state s of the model \mathcal{M} satisfies the CTL formula φ "

$$\mathcal{M}, s \models \varphi$$

$$\mathcal{M}, s \models \top$$

$$\mathcal{M}, s \not\models \perp$$

$$\mathcal{M}, s \models p \quad \text{iff} \quad p \in L(s)$$

$$\mathcal{M}, s \models \varphi \wedge \psi \quad \text{iff} \quad \mathcal{M}, s \models \varphi \text{ and } \mathcal{M}, s \models \psi$$

$$\mathcal{M}, s \models \varphi \vee \psi \quad \text{iff} \quad \mathcal{M}, s \models \varphi \text{ or } \mathcal{M}, s \models \psi$$

$$\mathcal{M}, s \models \varphi \rightarrow \psi \quad \text{iff} \quad \mathcal{M}, s \models \varphi \text{ implies } \mathcal{M}, s \models \psi$$

$$\mathcal{M}, s \models \mathbf{AX} \varphi \quad \text{iff} \quad \text{for all } s' \in S \text{ such that } s \rightarrow s' \text{ we have } \mathcal{M}, s' \models \varphi$$

$$\mathcal{M}, s \models \mathbf{EX} \varphi \quad \text{iff} \quad \text{there exists } s' \in S \text{ such that } s \rightarrow s' \text{ and } \mathcal{M}, s' \models \varphi$$

Satisfaction Relation

Assume that $\pi = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$

- | | | |
|--|-----|--|
| $\mathcal{M}, s \models \text{AF } \varphi$ | iff | for all paths π such that $s_0 = s$
there exists i such that $\mathcal{M}, s_i \models \varphi$ |
| $\mathcal{M}, s \models \text{EF } \varphi$ | iff | there exists a path π such that $s_0 = s$ and
there exists i such that $\mathcal{M}, s_i \models \varphi$ |
| $\mathcal{M}, s \models \text{AG } \varphi$ | iff | for all paths π such that $s_0 = s$
for all i , $\mathcal{M}, s_i \models \varphi$ |
| $\mathcal{M}, s \models \text{EG } \varphi$ | iff | there exists a path π such that $s_0 = s$ and
for all i , $\mathcal{M}, s_i \models \varphi$ |
| $\mathcal{M}, s \models A[\varphi_1 \mathbf{U} \varphi_2]$ | iff | for all paths π such that $s_0 = s$
there exists i such that $\mathcal{M}, s_i \models \varphi_2$ and
for all $j < i$, $\mathcal{M}, s_j \models \varphi_1$ |
| $\mathcal{M}, s \models E[\varphi_1 \mathbf{U} \varphi_2]$ | iff | there exists a path π such that $s_0 = s$ and
there exists i such that $\mathcal{M}, s_i \models \varphi_2$ and
for all $j < i$, $\mathcal{M}, s_j \models \varphi_1$ |

CTL - Equivalences

$$\neg \mathbf{EX} \varphi \equiv \mathbf{AX} \neg \varphi$$

$$\neg \mathbf{EF} \varphi \equiv \mathbf{AG} \neg \varphi$$

$$\neg \mathbf{EG} \varphi \equiv \mathbf{AF} \neg \varphi$$

$$\mathbf{AF} \varphi \equiv \mathbf{A}[\top \mathbf{U} \varphi]$$

$$\mathbf{EF} \varphi \equiv \mathbf{E}[\top \mathbf{U} \varphi]$$

$$\mathbf{A}[\varphi \mathbf{U} \psi] \equiv \neg(\mathbf{E}[\neg \psi \mathbf{U} (\neg \varphi \wedge \neg \psi)] \vee \mathbf{EG} \neg \psi)$$

From these, one can show that the sets $\{\mathbf{AU}, \mathbf{EU}, \mathbf{EX}\}$ and $\{\mathbf{EU}, \mathbf{EG}, \mathbf{EX}\}$ are both adequate sets of temporal connectives.

Differences between LTL and CTL

LTL allows questions of the form:

- For all paths, does the LTL formula φ hold?
- Does there exist a path on which the LTL formula φ holds?
(Ask whether $\neg\varphi$ holds on all paths.)

CTL allows mixing of path quantifiers:

- $\mathbf{AG}(p \rightarrow \mathbf{EG} q)$
(For all paths, if p is true then there is a path on which q is always true.)
- Intuitively, in CTL we cannot fix a path π and talk about it.

Differences between LTL and CTL

However, some path properties are impossible to express in CTL.

The following formulas do not express the same thing:

- LTL: $G F p \rightarrow G F q$

for all paths π , if p holds infinitely often on π , then q holds infinitely often on π

- CTL: $AG AF p \rightarrow AG AF q$

if p holds infinitely often on all paths, then q holds infinitely often on all paths.

There exist fair refinements of CTL that address this issue to some extent.

Model checking algorithm for CTL

Model checking - The big picture

- An efficient algorithm to decide $\mathcal{M}, s \models \varphi$
- Typically without user interaction (fully automated)
- Different approaches: semantic, automata, tableau,...
- Provide a counterexample when $\mathcal{M}, s \not\models \varphi$.

The counterexample provides a clue to what is wrong:

- The system might be incorrect
- The model might be too abstract (in need of refinement)
- The specification might not be the intended one

State explosion problem

- The number of states grow exponentially with the number of system components.
- Different techniques to make state explosion less severe:
 - Abstraction
 - Induction
 - Partial order reduction
 - ...

- We present an algorithm which, given a model and a CTL formula, outputs **the set of states of the model that satisfy the formula**.
- The algorithm handles CTL formulas containing the connectives
$$\{\perp, \neg, \wedge, \mathbf{AF}, \mathbf{EU}, \mathbf{EX}\}$$
- Given an arbitrary CTL formula φ , we simply pre-process φ in order to write it in an equivalent form in terms of the above set of connectives.

The labelling algorithm

INPUT: a CTL model $\mathcal{M} = (S, \rightarrow, L)$ and a CTL formula φ

OUTPUT: the set of states of \mathcal{M} which satisfy φ

The labelling algorithm

INPUT: a CTL model $\mathcal{M} = (S, \rightarrow, L)$ and a CTL formula φ

OUTPUT: the set of states of \mathcal{M} which satisfy φ

1. Write φ in terms of the connectives $\{\perp, \neg, \wedge, \text{AF}, \text{EU}, \text{EX}\}$ using the equivalences in CTL

The labelling algorithm

INPUT: a CTL model $\mathcal{M} = (S, \rightarrow, L)$ and a CTL formula φ

OUTPUT: the set of states of \mathcal{M} which satisfy φ

1. Write φ in terms of the connectives $\{\perp, \neg, \wedge, \text{AF}, \text{EU}, \text{EX}\}$ using the equivalences in CTL
2. Label the states of \mathcal{M} with subformulas of φ that are satisfied there, starting with the smallest subformulas and working outwards towards φ .

The labelling algorithm

INPUT: a CTL model $\mathcal{M} = (S, \rightarrow, L)$ and a CTL formula φ

OUTPUT: the set of states of \mathcal{M} which satisfy φ

1. Write φ in terms of the connectives $\{\perp, \neg, \wedge, \text{AF}, \text{EU}, \text{EX}\}$ using the equivalences in CTL
2. Label the states of \mathcal{M} with subformulas of φ that are satisfied there, starting with the smallest subformulas and working outwards towards φ .

Suppose ψ is a subformula of φ . If ψ is

The labelling algorithm

INPUT: a CTL model $\mathcal{M} = (S, \rightarrow, L)$ and a CTL formula φ

OUTPUT: the set of states of \mathcal{M} which satisfy φ

1. Write φ in terms of the connectives $\{\perp, \neg, \wedge, \text{AF}, \text{EU}, \text{EX}\}$ using the equivalences in CTL
2. Label the states of \mathcal{M} with subformulas of φ that are satisfied there, starting with the smallest subformulas and working outwards towards φ .

Suppose ψ is a subformula of φ . If ψ is

- \perp : then no states are labelled with \perp

The labelling algorithm

INPUT: a CTL model $\mathcal{M} = (S, \rightarrow, L)$ and a CTL formula φ

OUTPUT: the set of states of \mathcal{M} which satisfy φ

1. Write φ in terms of the connectives $\{\perp, \neg, \wedge, \text{AF}, \text{EU}, \text{EX}\}$ using the equivalences in CTL
2. Label the states of \mathcal{M} with subformulas of φ that are satisfied there, starting with the smallest subformulas and working outwards towards φ .

Suppose ψ is a subformula of φ . If ψ is

- \perp : then no states are labelled with \perp
- p : then label s with p if $p \in L(s)$

The labelling algorithm

INPUT: a CTL model $\mathcal{M} = (S, \rightarrow, L)$ and a CTL formula φ

OUTPUT: the set of states of \mathcal{M} which satisfy φ

1. Write φ in terms of the connectives $\{\perp, \neg, \wedge, \text{AF}, \text{EU}, \text{EX}\}$ using the equivalences in CTL
2. Label the states of \mathcal{M} with subformulas of φ that are satisfied there, starting with the smallest subformulas and working outwards towards φ .

Suppose ψ is a subformula of φ . If ψ is

- \perp : then no states are labelled with \perp
- p : then label s with p if $p \in L(s)$
- $\psi_1 \wedge \psi_2$: label s with $\psi_1 \wedge \psi_2$ if s is already labelled with ψ_1 and ψ_2

The labelling algorithm

INPUT: a CTL model $\mathcal{M} = (S, \rightarrow, L)$ and a CTL formula φ

OUTPUT: the set of states of \mathcal{M} which satisfy φ

1. Write φ in terms of the connectives $\{\perp, \neg, \wedge, \text{AF}, \text{EU}, \text{EX}\}$ using the equivalences in CTL
2. Label the states of \mathcal{M} with subformulas of φ that are satisfied there, starting with the smallest subformulas and working outwards towards φ .

Suppose ψ is a subformula of φ . If ψ is

- \perp : then no states are labelled with \perp
- p : then label s with p if $p \in L(s)$
- $\psi_1 \wedge \psi_2$: label s with $\psi_1 \wedge \psi_2$ if s is already labelled with ψ_1 and ψ_2
- $\neg\psi_1$: label s with $\neg\psi_1$ if s is not already labelled with ψ_1

The labelling algorithm

- **EX** ψ_1 : label a state with **EX** ψ_1 if one of its successors is labelled with ψ_1

The labelling algorithm

- **EX** ψ_1 : label a state with **EX** ψ_1 if one of its successors is labelled with ψ_1
- **AF** ψ_1 :
 - If any state s is labelled with ψ_1 , label it with **AF** ψ_1
 - **Repeat**: label any state with **AF** ψ_1 if all its successor states are labelled with **AF** ψ_1 , until there is no change.

The labelling algorithm

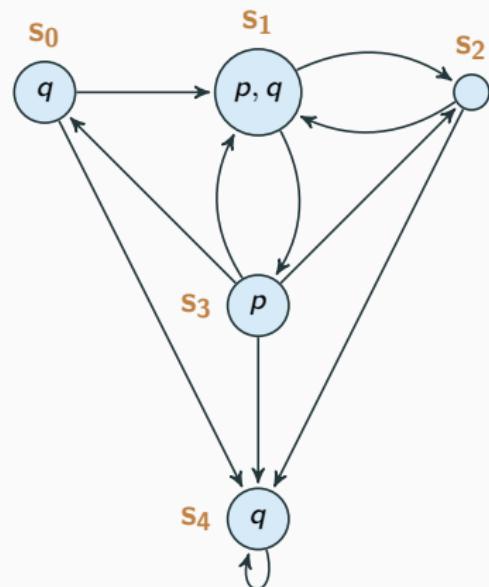
- **EX** ψ_1 : label a state with **EX** ψ_1 if one of its successors is labelled with ψ_1
- **AF** ψ_1 :
 - If any state s is labelled with ψ_1 , label it with **AF** ψ_1
 - **Repeat**: label any state with **AF** ψ_1 if all its successor states are labelled with **AF** ψ_1 , until there is no change.
- **E**[ψ_1 **U** ψ_2]:
 - If any state s is labelled with ψ_2 , label it with **E**[ψ_1 **U** ψ_2]
 - **Repeat**: label any state with **E**[ψ_1 **U** ψ_2] if it is labelled with ψ_1 and at least one of its successors is labelled with **E**[ψ_1 **U** ψ_2], until there is no change.

The labelling algorithm

Example

$$\text{AG}(q \rightarrow \text{EG } p) \equiv \neg \mathbf{E}[\neg \perp \text{ U } (q \wedge \mathbf{AF} \neg p)]$$

- Label q : $\{s_0, s_1, s_4\}$
- Label p : $\{s_1, s_3\}$
- Label $\neg \perp$: $\{s_0, \dots, s_4\}$

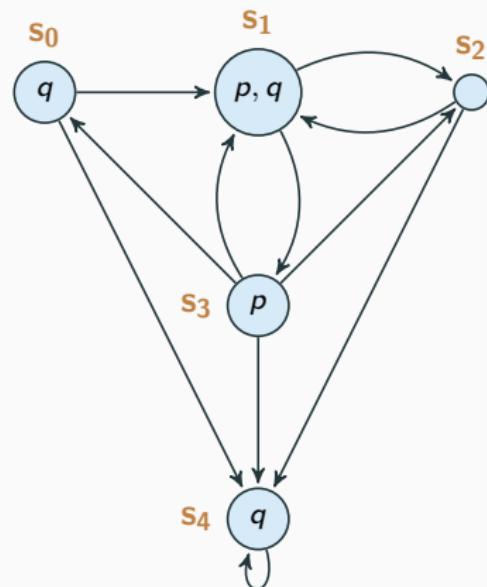


The labelling algorithm

Example

$$\mathbf{AG}(q \rightarrow \mathbf{EG} p) \equiv \neg \mathbf{E}[\neg \perp \mathbf{U} (q \wedge \mathbf{AF} \neg p)]$$

- Label q : $\{s_0, s_1, s_4\}$
- Label p : $\{s_1, s_3\}$
- Label $\neg p$: $\{s_0, s_2, s_4\}$
- Label $\neg \perp$: $\{s_0, \dots, s_4\}$

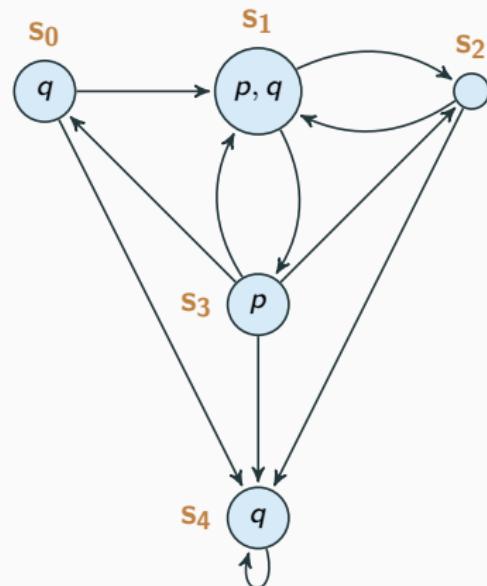


The labelling algorithm

Example

$$\mathbf{AG}(q \rightarrow \mathbf{EG} p) \equiv \neg \mathbf{E}[\neg \perp \mathbf{U} (q \wedge \mathbf{AF} \neg p)]$$

- Label q : $\{s_0, s_1, s_4\}$
- Label p : $\{s_1, s_3\}$
- Label $\neg p$: $\{s_0, s_2, s_4\}$
- Label $\mathbf{AF} \neg p$: $\{s_0, s_2, s_4\}$
 1. s_1 cannot be labeled because of s_3
 2. s_3 cannot be labeled because of s_1
- Label $\neg \perp$: $\{s_0, \dots, s_4\}$

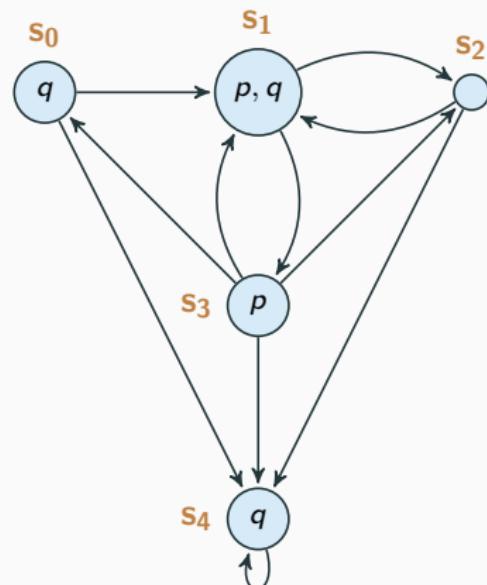


The labelling algorithm

Example

$$\mathbf{AG}(q \rightarrow \mathbf{EG} p) \equiv \neg \mathbf{E}[\neg \perp \mathbf{U} (q \wedge \mathbf{AF} \neg p)]$$

- Label q : $\{s_0, s_1, s_4\}$
- Label p : $\{s_1, s_3\}$
- Label $\neg p$: $\{s_0, s_2, s_4\}$
- Label $\mathbf{AF} \neg p$: $\{s_0, s_2, s_4\}$
 1. s_1 cannot be labeled because of s_3
 2. s_3 cannot be labeled because of s_1
- Label $q \wedge \mathbf{AF} \neg p$: $\{s_0, s_4\}$
- Label $\neg \perp$: $\{s_0, \dots, s_4\}$

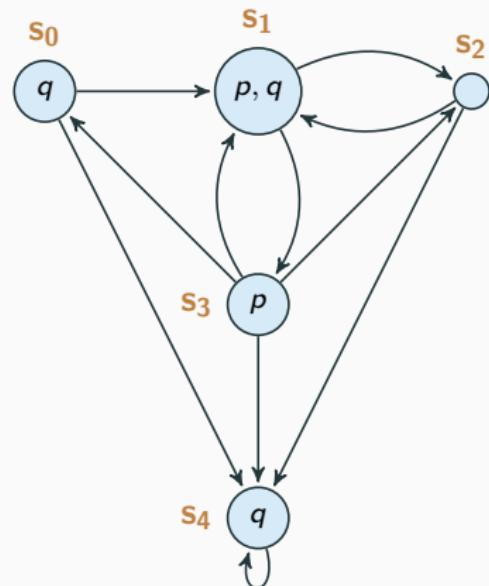


The labelling algorithm

Example

$$\text{AG}(q \rightarrow \text{EG } p) \equiv \neg \mathbf{E}[\neg \perp \text{ U } (q \wedge \text{AF} \neg p)]$$

- Label q : $\{s_0, s_1, s_4\}$
- Label p : $\{s_1, s_3\}$
- Label $\neg p$: $\{s_0, s_2, s_4\}$
- Label $\text{AF} \neg p$: $\{s_0, s_2, s_4\}$
 1. s_1 cannot be labeled because of s_3
 2. s_3 cannot be labeled because of s_1
- Label $q \wedge \text{AF} \neg p$: $\{s_0, s_4\}$
- Label $\neg \perp$: $\{s_0, \dots, s_4\}$
- Label $\mathbf{E}[\neg \perp \text{ U } (q \wedge \text{AF} \neg p)]$:
 1. $\{s_0, s_4\}$

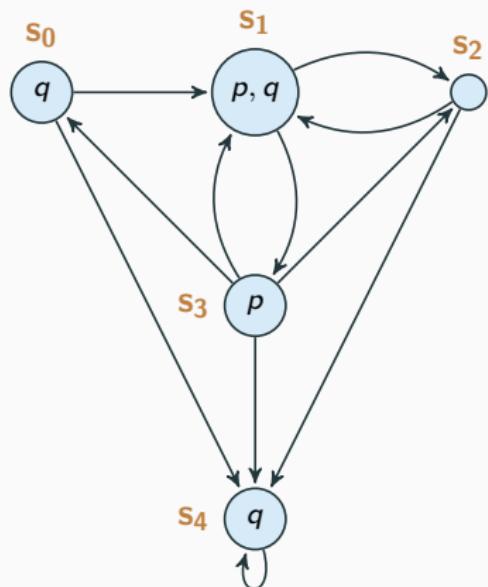


The labelling algorithm

Example

$$\text{AG}(q \rightarrow \text{EG } p) \equiv \neg \mathbf{E}[\neg \perp \text{ U } (q \wedge \mathbf{AF} \neg p)]$$

- Label q : $\{s_0, s_1, s_4\}$
- Label p : $\{s_1, s_3\}$
- Label $\neg p$: $\{s_0, s_2, s_4\}$
- Label $\mathbf{AF} \neg p$: $\{s_0, s_2, s_4\}$
 1. s_1 cannot be labeled because of s_3
 2. s_3 cannot be labeled because of s_1
- Label $q \wedge \mathbf{AF} \neg p$: $\{s_0, s_4\}$
- Label $\neg \perp$: $\{s_0, \dots, s_4\}$
- Label $\mathbf{E}[\neg \perp \text{ U } (q \wedge \mathbf{AF} \neg p)]$:
 1. $\{s_0, s_4\}$
 2. $\{s_0, s_4, s_2\}$

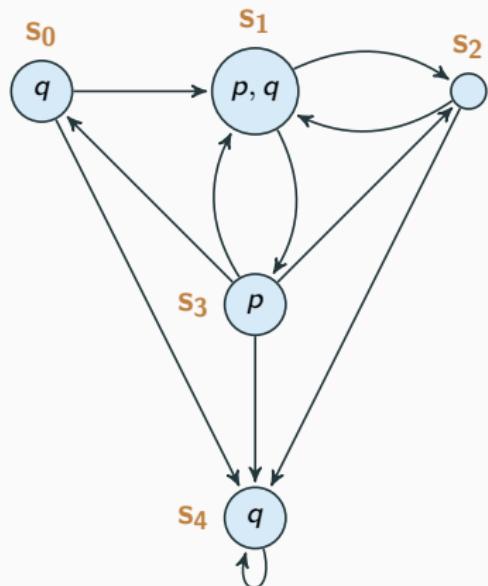


The labelling algorithm

Example

$$\text{AG}(q \rightarrow \text{EG } p) \equiv \neg \mathbf{E}[\neg \perp \text{ U } (q \wedge \text{AF} \neg p)]$$

- Label q : $\{s_0, s_1, s_4\}$
- Label p : $\{s_1, s_3\}$
- Label $\neg p$: $\{s_0, s_2, s_4\}$
- Label $\text{AF} \neg p$: $\{s_0, s_2, s_4\}$
 1. s_1 cannot be labeled because of s_3
 2. s_3 cannot be labeled because of s_1
- Label $q \wedge \text{AF} \neg p$: $\{s_0, s_4\}$
- Label $\neg \perp$: $\{s_0, \dots, s_4\}$
- Label $\mathbf{E}[\neg \perp \text{ U } (q \wedge \text{AF} \neg p)]$:
 1. $\{s_0, s_4\}$
 2. $\{s_0, s_4, s_2\}$
 3. $\{s_0, s_4, s_2, s_1\}$

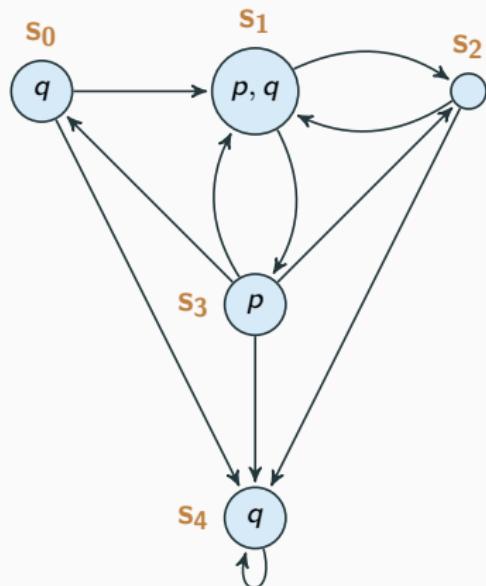


The labelling algorithm

Example

$$\text{AG}(q \rightarrow \text{EG } p) \equiv \neg \mathbf{E}[\neg \perp \text{ U } (q \wedge \mathbf{AF} \neg p)]$$

- Label q : $\{s_0, s_1, s_4\}$
- Label p : $\{s_1, s_3\}$
- Label $\neg p$: $\{s_0, s_2, s_4\}$
- Label $\mathbf{AF} \neg p$: $\{s_0, s_2, s_4\}$
 1. s_1 cannot be labeled because of s_3
 2. s_3 cannot be labeled because of s_1
- Label $q \wedge \mathbf{AF} \neg p$: $\{s_0, s_4\}$
- Label $\neg \perp$: $\{s_0, \dots, s_4\}$
- Label $\mathbf{E}[\neg \perp \text{ U } (q \wedge \mathbf{AF} \neg p)]$:
 1. $\{s_0, s_4\}$
 2. $\{s_0, s_4, s_2\}$
 3. $\{s_0, s_4, s_2, s_1\}$
 4. $\{s_0, s_4, s_2, s_1, s_3\}$

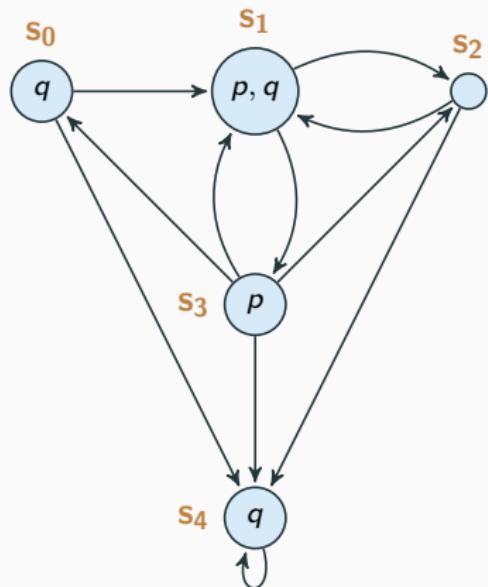


The labelling algorithm

Example

$$\text{AG}(q \rightarrow \text{EG } p) \equiv \neg \mathbf{E}[\neg \perp \text{ U } (q \wedge \mathbf{AF} \neg p)]$$

- Label q : $\{s_0, s_1, s_4\}$
- Label p : $\{s_1, s_3\}$
- Label $\neg p$: $\{s_0, s_2, s_4\}$
- Label $\mathbf{AF} \neg p$: $\{s_0, s_2, s_4\}$
 1. s_1 cannot be labeled because of s_3
 2. s_3 cannot be labeled because of s_1
- Label $q \wedge \mathbf{AF} \neg p$: $\{s_0, s_4\}$
- Label $\neg \perp$: $\{s_0, \dots, s_4\}$
- Label $\mathbf{E}[\neg \perp \text{ U } (q \wedge \mathbf{AF} \neg p)]$:
 1. $\{s_0, s_4\}$
 2. $\{s_0, s_4, s_2\}$
 3. $\{s_0, s_4, s_2, s_1\}$
 4. $\{s_0, s_4, s_2, s_1, s_3\}$
- Label $\neg \mathbf{E}[\neg \perp \text{ U } (q \wedge \mathbf{AF} \neg p)]$: \emptyset



The state explosion problem

Although the labelling algorithm is linear in the size of the model, unfortunately the size of the model is itself more often than not exponential in the number of variables and the number of components of the system which execute in parallel.

For example, adding a boolean variable to a program will double the complexity of verifying a property of it.

The tendency of state spaces to become very large is known as the **state explosion problem**.

The state explosion problem

A lot of research had gone into finding ways to overcoming it:

- Efficient data structures, called **ordered binary decision diagrams** (OBDDs) which represent **sets of states** instead of individual states.
- **Abstraction**: that one may interpret a model abstractly, uniformly or for a specific property.
- **Partial order reduction**: for asynchronous systems, several interleavings of components traces may be equivalent as far as satisfaction of the formula to be checked is concerned.
- **Induction**: model-checking systems with (e.g.) large numbers of identical, or similar, components can often be implemented by "induction" on this number.
- ...

References

- Lecture Notes on "Program Analysis", ETH Zurich, Martin Vechev.
- Lecture Notes on "Techniques for Program Analysis and Verification", Stanford, Clark Barrett.
- Lecture Notes on "Program verification", ETH Zurich, Alexander Summers.
- Lecture Notes on "Computer-Aided Reasoning for Software Engineering", University of Washington, Emin Török.