

Propositional Logic in Practice

Dependency Resolution

David Walker

Princeton University

Following notes by Zak Kincaid

Last Time: Propositional Logic

Consider a propositional signature P . Let p range over elements of P .

Syntax: $F ::= p \mid \text{true} \mid \text{false} \mid \neg F \mid F \wedge G \mid F \vee G \mid F \rightarrow G$

Semantics:

- A structure M assigns true or false to every atomic proposition p .
- $M \models F$ means that M **satisfies** F
 - we also say: M is a **model** of F
- $M \not\models F$ means that M does not satisfy F ; M is a **countermodel** of F
- If all M satisfy F then F is **valid**.

This time: Propositional Logic as a verification and problem-solving tool

Example Problem: Package Managers

Given you want to install package X, a package manager figures out:

- which *other* libraries or applications you need to install for X to work.

To install opam 2.6.0, I might need:

- gcc 5.4.0 or gcc 5.3.0 or gcc 5.2.9

There are lots of different package managers for all kinds of different collections of software.

apt

rpm

opam

cargo

brew

Package Managers

apt

rpm

opam

cargo

brew

Implementing a package manager sounds pretty easy:

- If you need to install package X, and it depends on Y, then install Y! (duh)

But, in reality:

- each package can depend on several *package groups*
 - a package group is a collection of other packages – any package in the group can satisfy the dependency for this package
- dependencies are *transitive*:
 - if package X depends upon B and C
 - then B might depend on Y, and C might depend on Z and so on.
- each package has many versions, *only one version can be installed at a time*
- *conflicts between the packages are possible*:
 - choosing B creates a conflict, while choosing C does not

Dependency Analysis is Complex

apt

rpm

opam

cargo

brew

Solution:

- represent dependencies as logical formula
- represent the “rules of the game” as more formula (constraints on the solution)
 - eg: only 1 version of each package may be installed at a time
- use a SAT solver to find a satisfying assignment

But isn't SAT a hard problem? (NP-Hard in fact?). Yes, but:

Industrial package managers are implemented this way!

How?

SAT Performance

Era / Milestone	Typical Variables Handled	Typical Clauses Handled	Key Architectural Innovations
Early 1990s <i>(Pre-CDCL)</i>	100 – 500	1,000 – 5,000	Classical DPLL search, truth-assignment backtracking, and simple unit resolution.
Late 1990s <i>(The CDCL Revolution)</i>	1,000 – 10,000	10,000 – 100,000	Introduction of GRASP and Chaff . Implemented Conflict-Driven Clause Learning (CDCL) , non-chronological backtracking, and implication graphs.
Early 2000s <i>(Engineering Optimization)</i>	100,000+	500,000+	Release of MiniSat . Breakthroughs include the 2-Watched-Literal scheme for rapid unit propagation and the VSIDS branching heuristic.
2010s <i>(Refined Management)</i>	1,000,000	3,000,000+	Advanced clause-deletion policies based on Literal Block Distance (LBD) (e.g., Glucose) and aggressive random restarts .
2020s – 2026 <i>(Modern Scale)</i>	1,000,000 – 10,000,000+	10,000,000 – 40,000,000+	Highly optimized solvers like Kissat . Heavy utilization of bounded variable elimination (BVE), inprocessing , and distributed parallel portfolio cloud systems.

progress on industrially relevant instances of SAT has followed an exponential curve since the 90s, doubling in performance on a regular basis

* AI-generated table!

Formalizing the Dependency Resolution Problem

Formalizing: Packages and Versions

(Following Burrows, *Modeling and Resolving Software Dependencies*)

Problem Set-up:

We are given a finite set of **packages** $P = \{ p_1, \dots, p_n \}$

```
P = { gcc, binutils, cpp }
```

For each package p , we are given a finite set of **versions** $V(p)$

```
V(gcc)      = { 0, 431, 540 }  
V(binutils) = { 0, 2261, 2270, 2271 }  
V(cpp)      = { 0, 531 }
```

Versions are identified using integers. (eg: we use 540 for gcc 5.4.0)

Version **0** reserved for the “uninstalled” version of any package.

A package p together with a version v (written $p.v$) are called an *instance*.

Formalizing: Dependencies

A finite set **D** of **dependencies**. Each one says: if this instance is installed, then at least one instance from this set must be too.

```
p.v    requires    { p1.v1,    . . . ,    pm.vm }
```

Example set of dependencies:

```
gcc.540    requires    { binutils.2261, binutils.2270, binutils.2271 }  
gcc.540    requires    { cpp.531 }  
gcc.431    requires    { binutils.2261 }  
cpp.531    requires    { binutils.2270, binutils.2271 }
```

Two rules for the same instance are **both** requirements:

- gcc.540 needs some binutils **and** it needs cpp.531.

Within one rule the choices are alternatives:

- Any one suffices.

Installations, and Consistency

An **installation** I maps each package to a version: $I : \text{Package} \rightarrow \text{Version}$

```
I = { gcc ↦ 540,    binutils ↦ 2270,    cpp ↦ 531 }
```

I is **consistent** when:

- for every dependency $p.v$ requires $\{ p_1.v_1, \dots, p_m.v_m \}$,
 - if $I(p) = v$ then:
 - $I(p_1) = v_1$ or ... or $I(p_m) = v_m$.

Note: Each dependency only makes requirements about the packages/versions that are installed.

A rule about `gcc.431` generates no requirements when `gcc.540` installed

The Dependency Resolution Problem

Given

packages P , a version map V , dependencies D , and requested instances $p_{i1}.v_{j1}, \dots, p_{im}.v_{jm}$

Ask

is there a *consistent* installation I with $I(p_{i1}) = v_{j1}$ and ... and $I(p_{im}) = v_{jm}$?

If yes, then What is that installation?

Using SAT to Solve The Dependency Resolution Problem

Strategy:

- **Define the atomic propositions** we want to use
 - explain how they relate to our real-world problem
 - eg: some propositions will be true when we want to install that package
- **Construct the formula F**
 - for any model M such that $M \models F$, M should tell us the information we need to solve our problem
 - eg: the model tells us which propositions are true – that is, the packages we need to install
- **Run the SAT solver on F**
 - find a model, or no model, if the constraints can't be satisfied.

One Variable per Instance

For each package version $p.v$, introduce one atomic proposition named “ $p.v$ ”.

$p.v$ holds iff version v of package p is installed

Our universe has nine instances, so nine propositions, named as follows:

```
gcc.0          gcc.431          gcc.540
binutils.0     binutils.2261    binutils.2270  binutils.2271
cpp.0          cpp.531
```

Note: We’ve been using letters like p and q to refer to atomic propositions.

We are now going to use “ $p.v$ ” to refer to an atomic proposition.

Even though its name is made up of several characters (“ p ”, “ $.$ ”, “ v ”), it is still just a name for one atomic proposition. It does not have any substructure.

(I could have used the names p_1 , p_2 , p_3 , p_4 , etc but it would have been difficult to keep track of which proposition stood for which real-world thing.)

Constraint 1: At Least One Version

Every package must have some version selected (possibly the uninstalled one).

To ensure our solution has this property, we need to generate a formula that is true if and only if that property holds.

Our formula looks like this:

```
for each package p with  $V(p) = \{ v_1, \dots, v_m \}$  :
```

$$p.v_1 \vee \dots \vee p.v_m$$

For our example:

```
(gcc.0  $\vee$  gcc.431  $\vee$  gcc.540)  
 $\wedge$  (binutils.0  $\vee$  binutils.2261  $\vee$  binutils.2270  $\vee$  binutils.2271)  
 $\wedge$  (cpp.0  $\vee$  cpp.531)
```

Constraint 2: At Most One Version of a Package

You cannot install two versions of the same package at once.

Again, we need to construct a formula that is true only when this property holds:

```
for each package p and each pair  $v_i, v_j$  with  $i < j$  :  
 $\neg(p.v_i) \vee \neg(p.v_j)$ 
```

For our example:

```
( $\neg gcc.0 \vee \neg gcc.431$ )  
 $\wedge (\neg gcc.0 \vee \neg gcc.540)$   
 $\wedge (\neg gcc.431 \vee \neg gcc.540)$   
  
... and six more for binutils, one for cpp
```

- One clause per *pair* of versions: $m(m-1) / 2$ of them for a package with m versions.
- The encoding grows quadratically in the size of the group

Constraint 3: Dependencies

If an instance is installed, one of the instances it requires must be too.

for each dependency $p.v$ requires $\{ p_1.v_1, \dots, p_m.v_m \} \in D :$

$$p.v \rightarrow p_1.v_1 \vee \dots \vee p_m.v_m$$

For our example:

```
(gcc.540 → (binutils.2261 ∨ binutils.2270 ∨ binutils.2271))  
∧ (gcc.540 → cpp.531)  
∧ (gcc.431 → binutils.2261)  
∧ (cpp.531 → (binutils.2270 ∨ binutils.2271))
```

Clause 4: The Request

Finally, the instances the user asked for must actually be installed.

```
for each requested version of a package p.v :  
    proposition p.v must be true
```

For our universe, we extend our formula with a clause saying this proposition must be true:

```
gcc.540
```

Summary: The Encoding

At least one version

one clause per package

$\wedge$

At most one version

one per pair of versions

$\wedge$

Dependencies

one per rule in D

$\wedge$

The request

one per requested instance

Hand the whole formula to the SAT solver and ask for a model back.

The Implementation

ocaml/pkg — about 150 lines, including the comments.

Implementing a Package Manager



Z3 is an industrial strength SAT (and SMT) solver

An incredible contribution to formal methods research and practice!

OCaml has an interface where you can create formulae as OCaml data structures

\$ opam install z3

Read about the interface here:

<https://z3prover.github.io/api/html/ml/Z3.html>

Implementing the Package Manager

defs.ml

The vocabulary: packages, versions, instances, rules, problems. Types only.

formula.ml (i)

A thin layer over Z3: build formulas, then check them.

encode.ml

The four clause groups, and reading an installation back out of a model.

problemA.ml / problemB.ml

Two example problems — one solvable, one not.

main.ml

Reads a -A / -B flag and runs the solver.

formula.mli: A Simpler Interface to the Z3 Solver

Z3's full API is a little more complicated than we need, so we supply the following wrapper

```
type formula

val bool_const : string -> formula (* create an atomic prop from a string name *)
val tru        : formula
val fls        : formula
val not        : formula -> formula
val ors        : formula list -> formula
val ands       : formula list -> formula
val ( &&& )     : formula -> formula -> formula
val ( ||| )    : formula -> formula -> formula
val ( ==> )    : formula -> formula -> formula
val ( <=> )    : formula -> formula -> formula

type model

type result = Satisfiable of model | Unsatisfiable | Unknown

val check      : formula list -> result
val satisfies  : model -> formula -> bool
val value_of_bool : model -> formula -> bool option
```

defs.ml: Defines a problem

```
type package = string
type version = int
type instance = package * version

(* list of packages and the versions that exist for each one *)
type universe = (package * version list) list

(* rule (i1, [i2; i3]) means i1 requires i2 or i3 *)
type rule = instance * instance list
type rules = rule list

(* instances that must be installed *)
type request = instance list

(* complete definition of a dependency problem to be solved *)
type problem = {
  universe : universe;
  rules : rules;
  request : request;
}
```

Corresponds with our formalization of the problem

- P and V become `universe`
- D becomes `rules`
- query becomes `request`

encode.ml: Generates our Formula

Each bullet from the notes becomes one function:

```
(* create a z3 proposition corresponding to package p and version v *)
let const ((p, v) : instance) : formula =
  bool_const (Printf.sprintf "%s.%d" p v)

let at_least_one_version (p, versions) =
  ors (List.map (fun v -> const (p, v)) versions)

let at_most_one_version (p, versions) =
  List.map
    (fun (vi, vj) -> not (const (p, vi)) ||| not (const (p, vj)))
    (pairs versions)

let dependency ((i, choices) : rule) =
  const i ==> ors (List.map const choices)
```

encode.ml: Putting It Together

The four groups, concatenated into one list of constraints:

```
let generate_constraints (p : problem) : formula list =  
  List.map at_least_one_version p.universe  
  @ List.concat_map at_most_one_version p.universe  
  @ List.map rule_constraint p.rules  
  @ List.map const p.request
```

Ask Z3 to solve and read the answer back:

```
let solve (problem : problem) : unit =  
  let cs = generate_constraints problem in  
  match check cs with  
  | Unsatisfiable -> print_endline "No consistent installation exists."  
  | Satisfiable model -> print_installation model problem.universe  
  | Unknown -> raise (Impossible "SAT is decidable")
```

problemA.ml: A Problem Instance

```
let universe : universe =  
  [ (gcc,      [ 0; 431; 540 ] );  
    (binutils, [ 0; 2261; 2270; 2271 ] );  
    (cpp,      [ 0; 531 ] ) ]  
  
let gcc_rule1 : rule = ((gcc, 540), [ (binutils, 2261);  
                                       (binutils, 2270);  
                                       (binutils, 2271) ] )  
  
let gcc_rule2 : rule = ((gcc, 540), [ (cpp, 531) ] )  
let gcc_rule3 : rule = ((gcc, 431), [ (binutils, 2261) ] )  
let cpp_rule1 : rule = ((cpp, 531), [ (binutils, 2270); (binutils, 2271) ] )  
  
let rules : rules = [ gcc_rule1; gcc_rule2; gcc_rule3; cpp_rule1 ]  
  
let request : instance list = [ (gcc, 540) ]  
  
let problem : problem = { universe=universe; rules=rules; request=request }
```

Running It

```
$ dune exec pkg -- -A
```

The program prints the constraints it built, then the answer:

```
(or gcc.0 gcc.431 gcc.540)
(or binutils.0 binutils.2261 binutils.2270 binutils.2271)
(or cpp.0 cpp.531)
(or (not gcc.0) (not gcc.431))
...
(=> gcc.540 (or binutils.2261 binutils.2270 binutils.2271))
(=> gcc.540 (or cpp.531))
(=> gcc.431 (or binutils.2261))
(=> cpp.531 (or binutils.2270 binutils.2271))
gcc.540
```

An installation that satisfies every rule:

```
gcc          version 540
binutils     version 2270
cpp          version 531
```

We asked only for gcc 540. The other two versions were forced by the rules.

Unsatisfiable Formula

Problem B keeps the same universe and rules, and changes the request:

```
let request : instance list = [ (gcc, 431); (cpp, 531) ]
```

- `gcc.431` requires `binutils.2261`.
- `cpp.531` requires `binutils.2270` or `binutils.2271`.
- At most one version of `binutils` may be installed.

```
$ dune exec pkg -- -B → No consistent installation exists.
```

Neither requested instance conflicts with the other directly.

The conflict is one dependency step away due to `binutils` requirements

Summary

The problem Choose one version per package so that every dependency of every chosen version is met, and the request is honored.

The encoding One variable per instance; four groups of constraints:

- (1) at least one version
- (2) at most one version
- (3) dependencies
- (4) request.

NP-hard problems can be solved in practice!

Reduce tough problems to SAT and let thirty years of solver engineering do the work!

Solving Planning Problems with SAT by Unrolling Time

A Different Kind of Problem

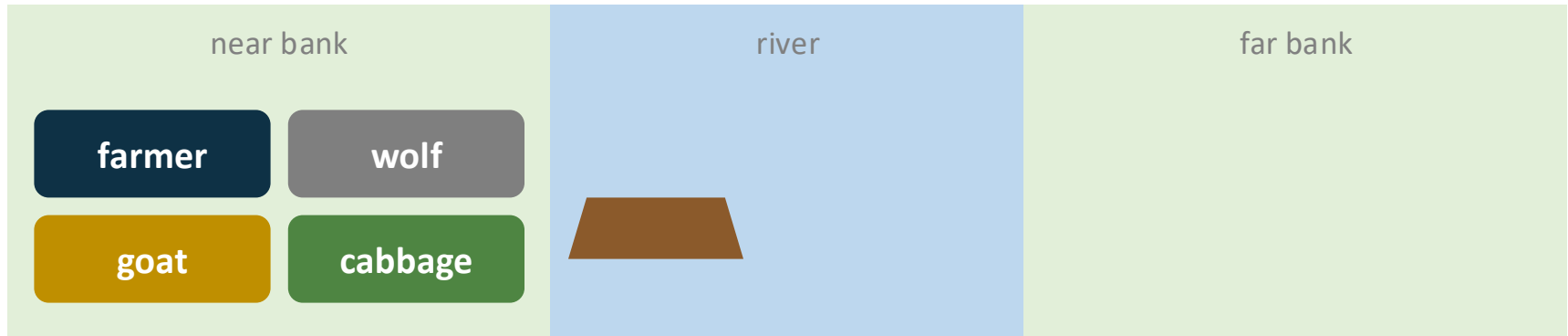
Package manager

- We care about one “**state**” of the system:
 - that state tells us which version of each package is installed
- We know the bits in advance: one per instance
- Solved with one SAT query

River crossing

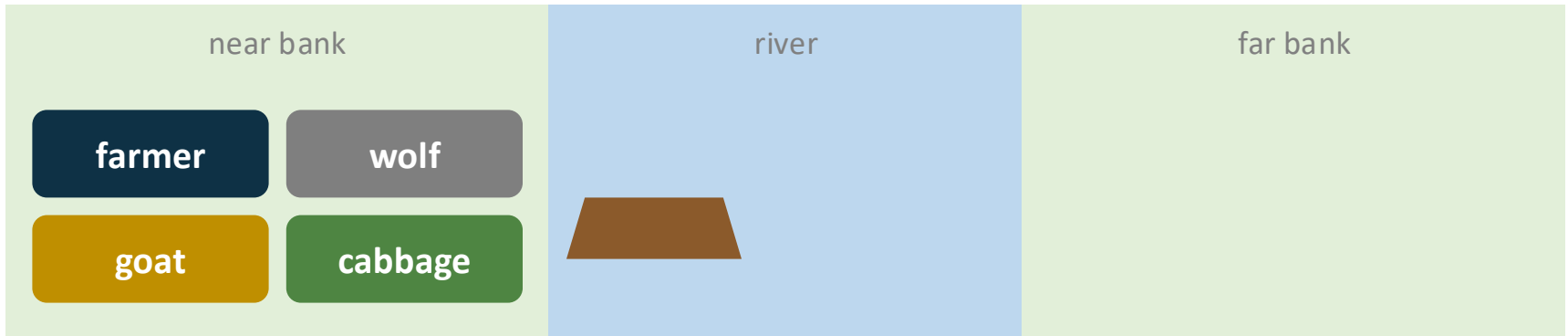
- Must accomplish a multi-step task.
- Each step generates a new state
- Each state has a finite representation
- But we do not know how many steps we need!
- So, up front, it seems like we may need an unlimited number of bits
- Solution:
 - Guess a steps.
 - Allocate that many bits (propositions)
 - Check for solution
 - If fail, allocate more bits

The Farmer, the Wolf, the Goat and the Cabbage



- A farmer is travelling with a wolf, a goat and a cabbage and comes to a river with a boat
 - initially, farmer and all companions are on the near bank together
- The boat holds the farmer and at most one other item.
 - ie: If the farmer is on one of the banks, they may choose to move to the other bank with at most one other item
 - Such a move generates a new state of the system
 - ie: different items are on different sides of the river
- Left without the farmer, **the wolf eats the goat**, and **the goat eats the cabbage**.
- **Is there a sequence of crossings that gets everyone across? How many?**

The Farmer, the Wolf, the Goat and the Cabbage



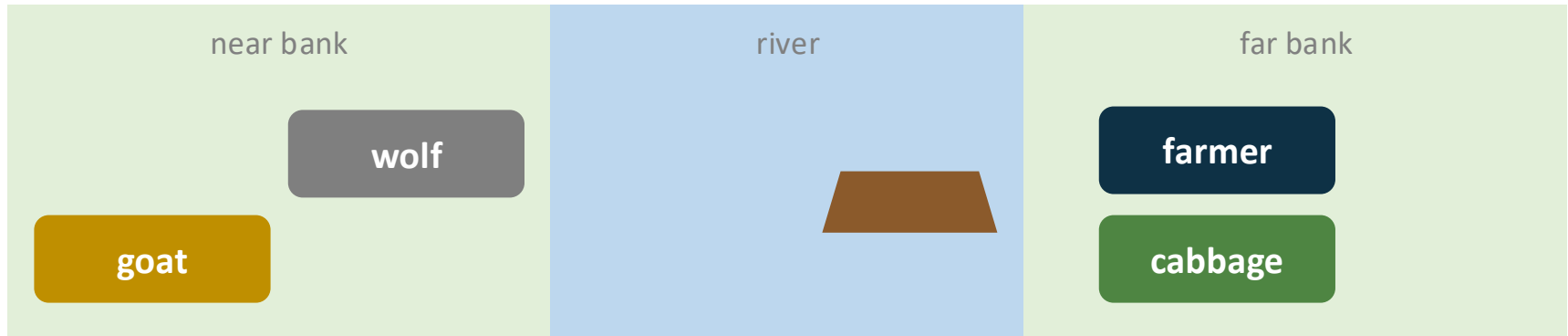
A **state** of the system may be represented as a collection of propositions p :

- **farmer, wolf, goat, cabbage**
 - p is false when p is on the near bank
 - p is true when p is on the far bank

Picture above can be represented like this:

```
[ farmer = false;  
  wolf = false;  
  goat = false;  
  cabbage = false ]
```

The Farmer, the Wolf, the Goat and the Cabbage



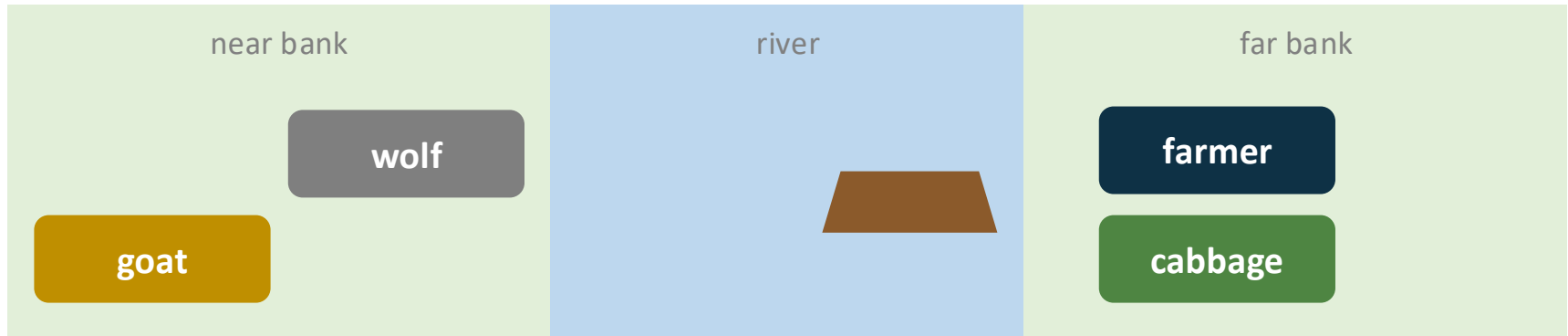
Now the picture is represented as follows:

```
[farmer = true;  
  wolf = false;  
  goat = false;  
  cabbage = true]
```

(Not such a great state since the wolf will eat the goat, which is what we are trying to avoid)

(The boat is always with the farmer so it doesn't need a separate proposition)

The Farmer, the Wolf, the Goat and the Cabbage



With 4 propositions (wolf, goat, farmer, cabbage), we can represent any state we want.

However, the problem asks us to find a *sequence of states*.

The first state in the sequence is the initial state with everyone on the near bank.

The goal state in the sequence is the state with everyone on the far bank.

We need more than 4 propositions. We need 4 propositions for each state!

Suppose we know that we need k states, we could create $4*k$ propositions.

eg: wolf@0, wolf@1, ..., wolf@ $k-1$), goat@0, goat@1, ...

A Sequence of States is a Plan

One propositional variable per item and time step:

`goat@3` holds iff after 3 crossings, the goat is on the far bank

The 7-crossing plan, as a model (1 = far bank):

t =	0	1	2	3	4	5	6	7
farmer	0	1	0	1	0	1	0	1
wolf	0	0	0	1	1	1	1	1
goat	0	1	1	1	0	0	0	1
cabbage	0	0	0	0	0	1	1	1

Each column is one state.

k crossings need $4(k + 1)$ variables: 32 for $k = 7$.

But we only learn that $k = 7$ works by solving the problem.

Solve with SAT by guessing a k , formulating a k -step problem, and checking if there is a solution. Might start with small k , and grow k each time you fail.

Constraints: Start, Goal

For a fixed k (eg: k is 3):

Start

everyone begins on the near bank

$\neg \text{farmer}@0 \wedge \neg \text{wolf}@0 \wedge \neg \text{goat}@0 \wedge \neg \text{cabbage}@0$

Goal

after k crossings, everyone is on the far bank

$\text{farmer}@k \wedge \text{wolf}@k \wedge \text{goat}@k \wedge \text{cabbage}@k$

Constraints: Safety

Pro Tip: When “programming” with logical formula, create higher-level abstractions to simplify construction of complex formula

```
let together (x, y, t) = x@t <=> y@t
```

Safety Constraints:

For each $t = 0 \dots k$, conjoin these formula:

never leave two enemies together without the farmer

```
¬( together(wolf, goat, t) ∧  
    ¬ together(wolf, farmer, t) )
```

```
¬( together(goat, cabbage, t) ∧  
    ¬ together(goat, farmer, t) )
```

Constraints for each Crossing

Another useful abstraction:

```
(* x changes bank *)  
let moves(x, t) =  $\neg(x@t \iff x@t+1)$ 
```

for $t = 0 \dots k$:

The farmer moves on every time step

$\text{moves}(\text{farmer}, t)$

An item crosses only with the farmer

$\text{moves}(x, t) \rightarrow (x@t \iff \text{farmer}@t)$ where $x \in \{\text{wolf}, \text{goat}, \text{cabbage}\}$

The boat holds at most one item

$\neg(\text{moves}(x, t) \wedge \text{moves}(y, t))$

for each x, y such
 $x \neq y$ (and x, y
not the farmer)

Unroll, and Unroll, and Unroll

```
let shortest () =  
  let rec search k =  
    match find_plan k with  
    | Some plan -> Some (k, plan)  
    | None -> search (k + 1)  
  in  
  search 0
```

```
$ dune exec river  
k = 0    UNSAT  
k = 1    UNSAT  
k = 2    UNSAT  
...  
k = 6    UNSAT  
k = 7    SAT
```

Each round adds **4 more variables** and one more copy of the safety and crossing rules.

The first SAT is the **shortest** plan.

UNSAT for every smaller k proves nothing shorter exists.

Running It

```
$ dune exec river
```

The model, printed one crossing per line:

```
      trip      near bank      far bank
0  start  farmer wolf goat cabbage |
1  -> goat  wolf cabbage      | farmer goat
2  <- alone  farmer wolf cabbage | goat
3  -> wolf  cabbage      | farmer wolf goat
4  <- goat  farmer goat cabbage | wolf
5  -> cabbage goat      | farmer wolf cabbage
6  <- alone  farmer goat      | wolf cabbage
7  -> goat      | farmer wolf goat cabbage
```

Summary: Unrolling Time

One state vs. a run:

- The package manager chooses one state, with a known number of bits
- In contrast, a plan is a sequence of states.

Brute Force Solution: Copy the state:

- One copy per step: $x@0, x@1, \dots, x@k$. $4(k + 1)$ variables for k crossings.

Four groups of constraints:

- start
- safe at every t
- a crossing from each t to $t + 1$
- goal at k .

Don't know k ? Try $k = 0, 1, 2, \dots$ until SAT, or you've exhausted the state space.

Finding Bugs in Programs by Unrolling

Program Verification

From Farmers to Programs:

- Before: propositional variables represented whether a farmer is on the near or far bank
- instead: propositional variable represents a **state of a program**.
 - eg: the local variable $x = 0$ or 1 , etc

t =	0	1	2	3	4	5	6	7
program state 0	0	1	0	1	0	1	0	1
program state 1	0	0	0	1	1	1	1	1
...	0	1	1	1	0	0	0	1
program state n	0	0	0	0	0	1	1	1



how the state changes
when you run the next
instruction in the program

A Lock Built from Flags

Two processes, **0** and **1**, run the same program. For process i , the other one is $j = 1 - i$.

```
0: wait until not flag[j]
1: flag[i] := true
2: flag[i] := false;
   goto 0
```

- Shared variables `flag[0]` and `flag[1]`, initially false.
- At each step a **scheduler** picks one process, which runs one instruction.
 - Any order is possible
 - Instruction execution assumed to be atomic
- **Goal: *mutual exclusion*:** the two processes are never both at instruction 2
 - idea: at instruction 2, some delicate operation occurs (eg: withdraw from bank)
- **This kind of program is incredibly difficult for humans to analyze**
 - must consider all possible schedules of P0, P0, P1, P1, P1, ...
 - 2^n schedules of length n

An Example Run

```
0: wait until not flag[j]
1: flag[i] := true
2: flag[i] := false;
   goto 0
```

State of the system:

- the program counter for each process (pc0 and pc1, which can be 0, 1, 2)
- the flag for each process

t	moves	what it does	pc0	pc1	flag[0]	flag[1]
0		start: both at 0, flags down	0	0	0	0
1	P0	flag[1] is false: go to 1	1	0	0	0
2	P0	flag[0] := true	2	0	1	0
3	P1	flag[0] is true: wait	2	0	1	0
4	P0	flag[0] := false; goto 0	0	0	0	0
5	P1	flag[0] is false: go to 1	0	1	0	0
6	P1	flag[1] := true	0	2	0	1

At most one process is ever at instruction 2.

At step 3, P1 waits because P0 holds the lock.

An Incorrect Run

Now the scheduler picks **P0, P1, P0, P1**:

```
0: wait until not flag[j]
1: flag[i] := true
2: flag[i] := false;
   goto 0
```

t	moves	what it does	pc0	pc1	flag[0]	flag[1]
0		start: both at 0, flags down	0	0	0	0
1	P0	flag[1] is false: go to 1	1	0	0	0
2	P1	flag[0] is false: go to 1	1	1	0	0
3	P0	flag[0] := true	2	1	1	0
4	P1	flag[1] := true	2	2	1	1

The state to avoid: **pc0 = 2** and **pc1 = 2**, both processes at instruction 2.

Bug: Each process checked the other's flag concurrently before signaling they were about to proceed.

Encoding the state as propositions

Program counters

where each process is: instruction 0, 1 or 2

<code>pc0=0@t</code>	<code>pc0=1@t</code>	<code>pc0=2@t</code>	exactly one true
<code>pc1=0@t</code>	<code>pc1=1@t</code>	<code>pc1=2@t</code>	exactly one true

Shared flags

state of the flags

`flag0@t` `flag1@t`

Scheduler, one bit per step

not state: a choice the solver makes at each step.

This part of the model records the trace.

Allows us to see how we arrived at a state for debugging

`sched@t` `true: process 1 moves from t to t+1`

8 bits per state, plus 1 per step.

Bounded Model Checking

Is a bad state reachable within k steps?

$$I(s_0) \wedge T(s_0, s_1) \wedge \dots \wedge T(s_{k-1}, s_k) \wedge (B(s_0) \vee \dots \vee B(s_k))$$

I: The initial state

both processes at instruction 0, both flags down

T: A formula that relates state s_i to s_{i+1} , describing one step

scheduled process runs an instruction. this formula implements the program

B: A formula describing the condition that indicates a bug

$pc0=2@t \wedge pc1=2@t$

The river crossing had the same shape, with the goal in place of B.

Here a satisfying assignment indicates a bug

T: Process i Runs One Instruction

```
0: wait until not flag[j]
1: flag[i] := true
2: flag[i] := false;
   goto 0
```

Defining the transition formula (ie: the program) if process i makes a move at time t

```
let step i t =
  let j = 1 - i and t' = t + 1 in
  [ (pc t i 0 &&& not (flag t j)) ==> pc t' i 1;          (* may proceed *)
    (pc t i 0 &&& flag t j)           ==> pc t' i 0;        (* keep waiting *)
    pc t i 1                         ==> (pc t' i 2 &&& flag t' i);
    pc t i 2                         ==> (pc t' i 0 &&& not (flag t' i));
    pc t i 0                         ==> unchanged (fun t -> flag t i) t ]
```

- $pc\ t\ i\ v$ is the atomic proposition indicating that:
 - the program counter of process i at time t is at instruction v
- $flag\ t\ i$ is the atomic proposition indicating
 - the value of the flag for process i at time t
- $unchanged\ x\ t$ is defined as $x\ t \iff x\ (t+1)$.

The Fix: Peterson's Lock

```
0: flag[i] := true
1: turn := j
2: wait until not flag[j] or turn = i
3: critical section!  flag[i] := false;  goto 0
```

```
$ dune exec mutex -- peterson
  k = 0    UNSAT
    ...
  k = 20   UNSAT
```

Is that a proof?

No: UNSAT up to k says only that there is no bug within k steps.

- Here there are only $4 \cdot 4 \cdot 2 \cdot 2 \cdot 2 = 128$ states, so UNSAT up to **$k = 127$** is a proof (`mutex peterson prove: 9 seconds`).

https://en.wikipedia.org/wiki/Peterson%27s_algorithm

Bounded Model Checking in Practice

1999: the idea

Biere, Cimatti, Clarke and Zhu replace the BDDs of earlier model checkers with a SAT solver, and unroll the system k steps. Most model checkers now have a SAT solver inside.

Biere et al., Symbolic Model Checking without BDDs, TACAS 1999 (ETAPS Test of Time Award, 2017)

Compaq: bugs in an Alpha microprocessor

Deep bugs in the memory subsystem of a next-generation Alpha, in a part already well explored by simulation. Failing traces of 25 to over 100 cycles; days with prior work, minutes with BMC.

Bjesse, Leonard and Mokkedem, Finding Bugs in an Alpha Microprocessor Using Satisfiability Solvers, CAV 2001

Intel: the Pentium 4

SAT-based BMC run on real Pentium 4 designs with over 1000 model variables, with no manual tuning.

Coty et al., Benefits of Bounded Model Checking at an Industrial Setting, CAV 2001

Amazon: the Firecracker virtual machine monitor

Kani, a bounded model checker for Rust, found a rounding error that let a guest do up to 0.01% more I/O than configured and could cause a crash

Monteiro and Roy, Using Kani to Validate Security Boundaries in AWS Firecracker, Kani blog, Aug 2023

Summary: Bounded Model Checking

The recipe $I(s_0) \wedge T(s_0, s_1) \wedge \dots \wedge T(s_{k-1}, s_k) \wedge (B(s_0) \vee \dots \vee B(s_k))$.

SAT a counterexample trace: a real bug, if T is right. Replay it to be sure.

UNSAT no bug within k steps. Not a proof, unless k reaches a bound.

Unknown depth we don't know how deep the bug is, so we unroll, and unroll, and unroll until we have decided we've tried hard enough. Involves many SAT calls, which can certainly get expensive.

There are many variations and optimizations of this basic idea.