BOUNDED MODEL CHECKING

Satisfiability of Boolean Formulae (SAT)

- SAT is a NP-complete problem
  - No sub-exponential algorithm (in worst case) is known
- However, recently «fast» tools have tackled SAT, showing that satisfiability problems can be solved efficiently in many practical cases
- «SAT solvers» use a standard format (DIMACS-CNF) and have great performance
  - They can routinely solve formulae with tens of thousands of variables and millions of constraints
- Zchaf, Minisat, etc.
- Various kinds of algorithms (not studied here)

DPLL algorithm

- SAT solvers accepts a standard format (DIMACS) for boolean formulae in Conjunctive Normal Form (CNF)

  $(x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z)$ becomes (3 variables, 2 clauses, literals $x, y, z, \neg y, \neg z$)

  p cnf 3 2

  1 2 3 0

  1 -2 -3 0

- Most algorithms are (very sophisticated) variations of a procedure called **Davis-Putnam-Logemann-Loveland** (DPLL), based on backtracking.

```
\\ F formula in CNF, T is the set of true literals (initially empty)
boolean DPLL(F,T)  {
  if (F evaluted over T is true) return true
  if (F evaluted over T is false) return false
  if (a clause of F has only one literal L ) return  DPLL(F(L/true), T ∪ {L})
  if (a literal L only appears in F as L=x or L =not x but not both)
                 return  DPLL(F(L/true), T)
  Choose a literal L;
 return DPLL(F(L/true), T ∪ {L}) || DPLL(F(L/false), T ∪ {!L}) ;
}
```

Unitary clause

Pure literal

Backtracking

Idea of BMC

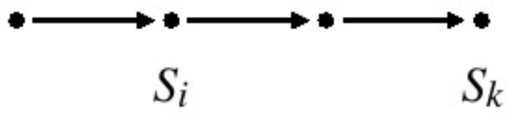
- «Counterexamples» to a LTL property P of a TS have finite length
  - since number of states is finite, a cycle (going through final states) occurs within a path of bounded length (at most #states+1);
  - The bound is called *diameter or completeness threshold.*
- Given any k>0, we can encode the «unfolding up to k steps» of the transition relation of TS into a boolean formula
- Adding suitable constraints (based on P and on cycle detection), we can build a formula Φ_k such that

Φ_k *is satisfiable iff there is a counterexample to P on TS of length at most k*

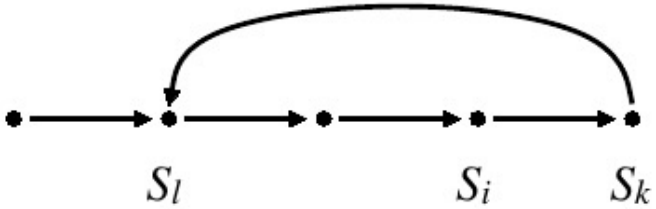
Back Loops

A prefix of length k of a path is finite, but it still might represent an infinite path if there is a back loop from the last state of the prefix to any of the previous states.

If there is no such back loop then the prefix does not say anything about the infinite behavior of the path



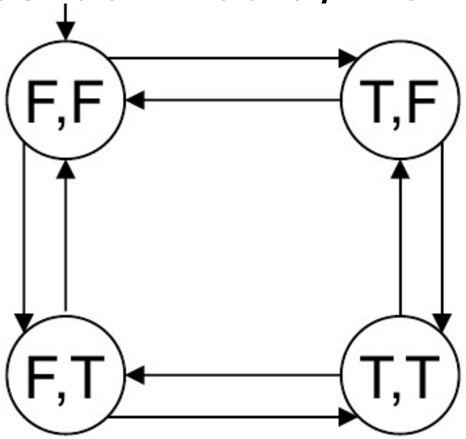
(a) no loop



(b) (k, l)-loop

Ex: transition relation as a boolean formula

- AP={x, y}. Consider x,y as boolean variables.
- Four states identified by their labels {∅, {y},{x},{x,y}}:.



States: {FF,FT,TF,TT}.

Initial condition can be expressed as $I \equiv \neg x \wedge \neg y$

- Introducing x',y' to denote variables in the next state, define the transition relation as the following boolean formula T(s,s'):
- $T((x,y),(x',y')) \equiv (x'=\neg x \wedge y'=y) \vee x'=x \wedge y'=\neg y$

Ex: unfolding of the transition relation

- Unroll (unfold) the transition relation a fixed number of times starting from the initial states
- For each unfolding, create a new set of variables:
  - Initial states: characterized with the variables x0 and y0
  - States after executing one transition: characterized with the variables x1 and y1
  - States after executing two transitions: characterized with the variables x2 and y2
  - etc., up to k

Example of unfolding

0) Initial states: $I(x0,y0) \equiv \neg x0 \land \neg y0$

1) Unfolding the transition relation once (bound k=1): $I(x0,y0) \land T(x0,y0,x1,y1)$:

$\neg x0 \land \neg y0 \land (x1=\neg x0 \land y1=y0 \lor x1=x0 \land y1=\neg y0)$

2) Unfolding the transition relation twice (bound k=2): $I(x0,y0) \land T(x0,y0,x1,y1) \land T(x1,y1,x2,y2)$:

$\neg x0 \land \neg y0 \land (x1=\neg x0 \land y1=y0 \lor x1=x0 \land y1=\neg y0) \land (x2=\neg x1 \land y2=y1 \lor x2=x1 \land y2=\neg y1)$

3) Unfolding the transition relation thrice (bound k=3):

$I(x0,y0) \land T(x0,y0,x1,y1) \land T(x1,y1,x2,y2) \land T(x2,y2,x3,y3)$:

$\neg x0 \land \neg y0 \land (x1=\neg x0 \land y1=y0 \lor x1=x0 \land y1=\neg y0)$

$\land (x2=\neg x1 \land y2=y1 \lor x2=x1 \land y2=\neg y1)$

$\land (x3 =\neg x2 \land y3=y2 \lor x3=x2 \land y3 =\neg y2)$

Unfolding the transition relation

1. The unfolding k times of the transition relation T of a transition system M is defined by a propositional formula $|[M]|_k$

NB: The notation |[X]| denotes the Boolean variables introduced in the encoding to represent some entity X

– The states s_i are represented as **bit vectors S_i.**

- The k-times unfolding of the transition relation represents all the finite paths of length k:

$$|[M_S]|_k \longleftrightarrow I(S_0) \wedge \bigwedge_{0 \leq i < k} T(S_i, S_{i+1})$$

Loop variables

- New variables and related constraints are introduced to denote the position of a loop (if any)
- Add k+1 *loop selector variables* $l_0, l_1, \ldots, l_k$
  - l_h describes the existence of a loop at position h; at most one of these variables can be true
  - If l_h is true then state $S_{h-1} = S_k$, i.e., the bit vector representing the state S_{h-1} is identical to the one for state S_k

- k+1 propositional variables, $InLoop_i$ (0<=i<= k) ("position i is inside a loop")
- Boolean var. *LoopExists* ("a loop does actually exist in the structure").

Existence of a loop at position i

- Position i is inside a loop:

Loop constraints:

Base	$\neg l_0 \wedge \neg \mathrm{InLoop}_0$
$1 \leq i \leq k$	$(l_i \rightarrow S_{i-1} = S_k) \wedge (\mathrm{InLoop}_i \longleftrightarrow \mathrm{InLoop}_{i-1} \vee l_i)$ $(\mathrm{InLoop}_{i-1} \rightarrow \neg l_i) \wedge (\mathrm{LoopExists} \longleftrightarrow \mathrm{InLoop}_k)$

- Truth of all subformulae is identical at position k+1 and i: (if there is no loop everything is false after k, else k+1 is just like i)

Last state constraints:

Base	$\neg \mathrm{LoopExists} \rightarrow \neg |[\phi]|_{k+1}$
$1 \leq i \leq k$	$l_i \rightarrow (|[\phi]|_{k+1} \longleftrightarrow |[\phi]|_i)$

Formula variables

- The semantics of a LTL formula Φ in positive normal form is given as a set of Boolean constraints over new *formula variables*.
  - There is a propositional variable $|[\varphi]|_i$ for each subformula of Φ and for each instant $0<=i<=k+1$
- For instance, for a formula A *U* B we introduce:

$$|[A]|_i,\ |[B]|_i,\ |[A\ U\ B]|_i,\ |[\neg A]|_i,\ |[\neg B]|_i,\ |[\neg(A\ U\ B)]|_i$$

- NB: instant k+1 is a fictitious one to simplify encoding

Propositional constraints

φ	$0 \leq i \leq k$
p	$\|[p]\|_i \longleftrightarrow p \in S_i$
$\neg p$	$\|[\neg p]\|_i \longleftrightarrow p \notin S_i$
$\phi_1 \wedge \phi_2$	$\|[\phi_1 \wedge \phi_2]\|_i \longleftrightarrow \|[\phi_1]\|_i \wedge \|[\phi_2]\|_i$
$\phi_1 \vee \phi_2$	$\|[\phi_1 \vee \phi_2]\|_i \longleftrightarrow \|[\phi_1]\|_i \vee \|[\phi_2]\|_i$

As a simple example, consider the formula $A \wedge B \vee \neg C$. For each $0 \leq i \leq k$, we introduce the propositional formulae:

$||[A \wedge B \vee \neg C]||_i \longleftrightarrow ||[A \wedge B]||_i \vee ||[\neg C]||_i$, and $||[A \wedge B]||_i \longleftrightarrow ||[A]||_i \wedge ||[B]||_i$.

Temporal Operators: weak versions

φ	$0 \leq i \leq k$
$\circ\phi_1$	$\|[\circ\phi_1]\|_i \longleftrightarrow \|[\phi_1]\|_{i+1}$
$\phi_1\mathcal{U}\phi_2$	$\|[\phi_1\mathcal{U}\phi_2]\|_i \longleftrightarrow \|[\phi_2]\|_i \vee (\|[\phi_1]\|_i \wedge \|[\phi_1\mathcal{U}\phi_2]\|_{i+1})$
$\phi_1\mathcal{R}\phi_2$	$\|[\phi_1\mathcal{R}\phi_2]\|_i \longleftrightarrow \|[\phi_2]\|_i \wedge (\|[\phi_1]\|_i \vee \|[\phi_1\mathcal{R}\phi_2]\|_{i+1})$

Above is just a *weak until* $\boldsymbol{\mathcal{W}}$. Definition is based on fixed point:
A $\boldsymbol{\mathcal{W}}$ B is equal to A $\vee \bigcirc$ (A$\boldsymbol{\mathcal{W}}$ B)

To encode ”strong” Until we need to deal with “eventualities”:
A $\boldsymbol{\mathcal{U}}$ B is equal to (A $\boldsymbol{\mathcal{W}}$ B) $\wedge \lozenge$ B

Temporal Operators: eventualities

- Correct treatment of "strong" until requires new propositional letters $\langle\langle\Diamond\phi_2\rangle\rangle_i$ for each subformula $\phi_1\mathcal{U}\phi_2$, (the Release operator is dual).
- There is an additional complication since eventualities can occur inside a loop.

Eventuality constraints:

φ	Base
$\phi_1\mathcal{U}\phi_2$	$\neg\langle\langle\Diamond\phi_2\rangle\rangle_0 \wedge \big(\mathrm{LoopExists} \rightarrow \big(\ \lvert[\phi_1\mathcal{U}\phi_2]\rvert_k \rightarrow \langle\langle\Diamond\phi_2\rangle\rangle_k\ \big)\big)$
$\phi_1\mathcal{R}\phi_2$	$\langle\langle\Box\phi_2\rangle\rangle_0 \wedge \big(\mathrm{LoopExists} \rightarrow \big(\ \lvert[\phi_1\mathcal{R}\phi_2]\rvert_k \leftarrow \langle\langle\Box\phi_2\rangle\rangle_k\ \big)\big)$

φ	$1 \leq i \leq k$
$\phi_1\mathcal{U}\phi_2$	$\langle\langle\Diamond\phi_2\rangle\rangle_i \longleftrightarrow \langle\langle\Diamond\phi_2\rangle\rangle_{i-1} \vee (\mathrm{InLoop}_i \wedge \lvert[\phi_2]\rvert_i)$
$\phi_1\mathcal{R}\phi_2$	$\langle\langle\Box\phi_2\rangle\rangle_i \longleftrightarrow \langle\langle\Box\phi_2\rangle\rangle_{i-1} \wedge (\neg\mathrm{InLoop}_i \vee \lvert[\phi_2]\rvert_i)$

Complete Encoding Φ_k

- The complete encoding of Φ is a boolean formula Φ_k consisting of the logical conjunction of the components:
  - loops, propositional connectives, temporal operators, and eventualities
- with $|[\Phi]|_0$ (i.e. Φ is evaluated only at instant 0).

Summary: Complete translation

1. Unfold the transition relation k times, defining a propositional formula $|[M]|_k$
2. New variables and related constraints are introduced to denote the existence and the position of a loop (if any)
3. The semantics of a LTL formula Φ, in release-positive normal form, is defined by a set of Boolean constraints over new *formula variables*.
   - There is a propositional variable $|[\varphi]|_i$ for each subformula and for each instant $0 \le i \le k+1$

BMC Procedure

1. Choose k «large enough but not too large»
2. Build Φ_k
3. With a SAT solver check Φ_k
4. If Φ_k is SAT: the solver returns a (nonspurious) counterexample to P of length <=k
5. If Φ_k is UNSAT then *maybe* the property is verified, but we may look for longer and longer counterexamples by incrementing the bound k and going to step 2.

The procedure is complete, because there is always a value CT, called *Completeness Threshold of the TS*, such that every counterexample has length less than CT:

if k>=CT and formula is UNSAT then property P holds.

However, the value of CT is usually unknown and hard (not impossible!) to compute.

(It is possible to verify with BMC itself a formula stating that k>=CT, but verification may be computationally too expensive)

BMC is very fast

- Since BMC produces a boolean formula, we can use a SAT-solver
- BMC is exceedingly faster than «traditional» model cheking in finding counterexamples
  - Order of magnitudes

- However, the answer «unsat» for a value k<CT cannot rule out that the property P is actually violated for a larger k….

Use of BMC in practice

- BMC is used mainly for fast «counterexample detection»
- Counterexamples are used to debug the model and/or the property
- After «debugging» is over we can be confident enough to have found all significant errors…
- …or we can then use more costly techniques to prove/check the property over the model.

Tool and Bibliography

- Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. (TACAS'99). Springer (1999) 193–207.
  - Defined in NuSMV
- Our own work in Zot (adding metric and past operators):

M. Pradella, A. Morzenti, P. San Pietro, Bounded satisfiability checking of metric temporal logic specifications. ACM Transactions on Software Engineering and Methodology (TOSEM) 22 (3), 2013

MODEL CHECKING IN THE SOFTWARE LIFE CYCLE

HW vs. SW verification

- Verification very successful in HW and protocol verification, where significant components can be verified.
  - In HW, need to check equivalence between Register Transfer Level description (e.g., a design defined in VHDL) and its netlist implementation (description of the connectivity of the electronic circuit).
  - *Equivalence checking* based on techniques like SAT or BDD, very close to model checking.
  - Commercial tools exist
- What about software engineering?
- Data-intensive software is not amenable to exhaustive verification of a complete system, because of the state explosion problem
- What can be done?

Software verification

- There are cases where an existing software component may be analyzed respect to some properties
  - SPIN helped NASA in developing Mars Science Laboratory, Deep Space 1, Cassini, the Mars Exploration Rovers, Deep Impact, by verifying software subsystems for potential race conditions and deadlocks.
  - *Java Pathfinder* (JPF), an open source tool by NASA to verify Java programs (Java bytecode), based on Spin.
    - Designed for detecting defects (e.g., deadlocks) in concurrent programs, but also used for distributed applications, embedded systems, GUI applications, spacecraft controllers and realtime OS.
- Still, in many cases starting from the code is computationally too intensive.
- Typically, some *abstraction* techniques are applied to reduce the state space so that MC can be applied and give meaningful results

MC for early testing/prototyping.

- Often, a model can be built for some of the most critical or difficult-to-design/test components
  - concurrent code, mission-critical components, complex GUI, communication protocols…
- MC strongly helps finding and fixing bugs in the model
  - Without MC those errors could propagate down to the implementation, becoming costly and difficult to find and debug
    - Errors found late may cost 100 times to fix than errors found early!
    - It requires to build a finite-state model and to express its properties
  - Unfortunately, even when having the right model, the final implementation may contain other bugs… that's why *this application* of MC is more akin to testing than to verification technique

Example of application of abstraction

- SLAM toolkit based on «boolean abstraction»
  - An approximation of the original program preserving reachability properties
  - A predicate P (one for each variable) partitions the state space in two classes: those verifying P and those not verifying P
  - Predicates are found «automatically»
  - It is then possible to build a boolean program, whose reachability is decidable by a symbolic model checker
- Developed by Microsoft Research to check 3rd party device drivers
- It is now part of the Microsoft Windows Driver Foundation development kit as the Static Driver Verifier (SDV).