

Program verification

Generalities about software Verification – Model Checking

Laure Gonnord David Monniaux

September 20, 2016



The teaching staff

- ▶ Laure GONNORD, associate professor, LIP laboratory, University of Lyon
Laure.Gonnord@ens-lyon.fr
- ▶ David MONNIAUX, CNRS senior researcher, VERIMAG laboratory, Grenoble
David.Monniaux@imag.fr

Schedule

Software, bugs, and proofs

States, reachability, and all that

Explicit (finite state) Forward Model Checking



Safe software ?

- ▶ safety-critical software (control of vehicles e.g. airplanes and cars, surgical robots, radiation therapy. . .)
- ▶ less critical software (flight management systems, financial transactions, unmanned spacecraft. . .)
- ▶ general-purpose software?
- ▶ (hot topic) software facing the Internet, newer security and privacy issues (e.g. 0-day vulnerabilities sold to intelligence services; are lives at stake ?)

Software engineering

Considers the means of production of software:

- ▶ documentation imperatives
- ▶ organization of software development teams
- ▶ good programming practices
- ▶ use of appropriate programming languages
- ▶ software development environments

Try to reduce the number of errors at the source.

“Software metrics”

Use of `lint`-like tools or more advanced

Not covered in this course



Proving properties of software

- ▶ Basic idea: software has **mathematically defined behaviour**
- ▶ Possible to do mathematical **proofs** on software
- ▶ Possible to **automate these proofs**



Program proofs

Proving that software truly does what it is meant to do.

behaviours \subseteq acceptable behaviours

- ▶ What does software do?
- ▶ What is it meant to do?
- ▶ What is a proof?



Semantics

A **precise definition** of what a program does — given for all programs within a programming language.

Very difficult for a full industrial language e.g. C++

vs

A definition in \pm vague **natural language** (e.g. ISO C and C++ standards, programming language manuals. . .)

Imprecise, fuzzy, sometimes contradictory definitions.

Language lawyers. Endless discussions on what a program should or should not be doing, on what a compiler has the right to do or not.



Specification

What software should do

- ▶ informal definition in natural language
- ▶ formal mathematical definition

Is the specification consistent?

Difficulties in writing specifications:

- ▶ Are all requirements taken into account?
- ▶ Redundancy with implementation



Specification example: sort

Unix command `sort`

(Without the options) Simple informal specification: “sort the lines in a file”

In more detail: complicated — e.g.

- ▶ what is the sorting order wrt non-ASCII characters?
- ▶ how are equivalent lines sorted (e.g. numeric ordering)

Mathematical definition possible, but long.



Difficulty

behaviours \subseteq acceptable behaviours

Both sets are not well defined in general.

May need to fix

- ▶ language definition
- ▶ target compilation environment (evaluation order, size of basic types, alignment. . .)
- ▶ precise specification

How about proofs?



The Halting Problem

Simple language: integers (\mathbb{Z}), tests, loops

There is no algorithm that says, given a program, whether this program halts. (Turing)



The Halting Problem, proof - Program Version.

Suppose we have a “magical analyzer” A : answer $A(P, X) = 1$ if “program P terminates eventually on input X ” $A(P, X) = 0$ otherwise.

```
int B(Program x) {
    if (A(x,x)==0) {
        return 1;
    } else {
        while(true) {}
    }
}
```

What is $B(B)$? (B applied to its own source code)



The Halting Problem, contradiction

```
int B(Program x) {  
    if (A(x,x)==0) {  
        return 1;  
    } else {  
        while(true) {}  
    }  
}
```

If $B(B) = 1$ then $A(B, B) = 0$ “program B does not terminate on input B ”. Absurd!

If $B(B)$ loops then $A(B, B) = 1$ “program B terminates on input B ”. Absurd!

There is no magical static analyser.



Workarounds

What is impossible is to check reachability

1. automatically
2. without false positives
3. without false negatives
4. on systems of unbounded state
5. with unbounded execution time

Lifting restrictions opens possibilities!



Course plan

Fully automatic decision procedures.

1. Course 1 : Finite state or infinite state model checking
2. Course 2 & 3 : Abstract interpretation (no false negative)
3. 2 Labs → one project.
4. Course 4 & 5 : Applications to safety-critical software verification, termination, worst-case execution time. . .
5. Paper reading → talks.



Schedule

Software, bugs, and proofs

States, reachability, and all that

Explicit (finite state) Forward Model Checking



Starting states + transitions

State of the program / of the machine = values of variables, registers, memories. . . within Σ .

Par exemple :

- ▶ if system state = 17 Booleans, then $\Sigma = \{0, 1\}^{17}$;
- ▶ if system state = 3 unbounded integers, then $\Sigma = \mathbb{Z}^3$;
- ▶ if finite automaton, Σ is the set of states;
- ▶ if stack automaton, state = pair (automaton state, stack contents), so $\Sigma = \Sigma_S \times \Sigma_P^*$.

Transition relation $\rightarrow : x \rightarrow y =$ “if I’m at x I can go to y at the next step”



Safety properties - Simple case

Show that a program cannot reach a “bad state” (crash, out-of-specification).

Set W of bad states.

Show that there is no $n \geq 0$ and $\sigma_0 \rightarrow \sigma_1 \rightarrow \dots \rightarrow \sigma_n$, σ_0 initial state (= reset), $\sigma_n \in W$ (trace of n steps leading to a bad state).

Otherwise said: $\sigma_0 \rightarrow^* \sigma_n \in W$. \rightarrow^* **reflexive transitive closure** of \rightarrow .



Reachable states

Let $\Sigma_0 \subseteq \Sigma$ be the initial states.

The set A of **reachable states** is the set of states σ such that

$$\exists \sigma_0 \in \Sigma_0 \sigma_0 \rightarrow^* \sigma \quad (1)$$

We want to show that $A \cap W = \emptyset$.

Bounding the state space

Restrict to a finite number of variables of a finite type.

Finite state space \implies “it’s just a big finite automaton!”

Everything is decidable!



Schedule

Software, bugs, and proofs

States, reachability, and all that

Explicit (finite state) Forward Model Checking



Explicit-state model checking

Given a transition relation τ

- ▶ Set $R := \{\text{initialstate}\}$
- ▶ For each state x in R , add all x' such that $(x, x') \models \tau$
- ▶ Do it until R is saturated (no new states are added)
- ▶ Then R is the set of **reachable states**.

Then test whether R contains undesirable states.

Implementation issues

If state = n Boolean variables, 2^n possible states.

Memory usage linear in number of reachable states.

Store states in hash table.

Store states in distributed hash table.

Tool example: CADP (INRIA Grenoble)



Explicit state model checking, a weakness

Representation expensive even if the set of reachable states is “simple”.

e.g. $\{0, 1\}^n$ “everything reachable” needs $\Theta(2^n)$ memory

Try to compress sets of states by **symbolic** representation.



Reachable states as a limit

X_n is the set of states reachable within n steps of \rightarrow : $X_0 = \Sigma_0$,
 $X_1 = \Sigma_0 \cup R(\Sigma_0)$, $X_2 = \Sigma_0 \cup R(\Sigma_0) \cup R(R(\Sigma_0))$, etc.

with $R(X) = \{y \in \Sigma \mid \exists x \in X \ x \rightarrow y\}$.

X_k grows wrt \subseteq .

Its limit (= union of all terms) is the **set of reachable states**.



Iterative computation

Remark $X_{n+1} = \Sigma_0 \cup R(X_n)$.

Intuition: to reach in at most $n + 1$ steps

- ▶ either in 0 steps = initial states Σ_0
- ▶ either in $0 < k \leq n + 1$ steps, thus in at most n steps (X_n) followed by another step

But how to efficiently represent the X_n and compute over them?

The problem

Representing **compactly** sets of Boolean states

A set of vector n Booleans = a function from $\{0, 1\}^n$ into $\{0, 1\}$.

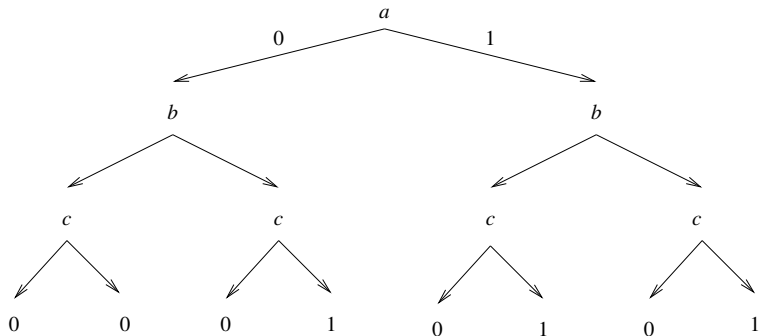
Example: $\{(0, 0, 0), (1, 1, 0)\}$ represented by $(0, 0, 0) \mapsto 1$, $(1, 1, 0) \mapsto 1$ and 0 elsewhere.



Expanded BDD

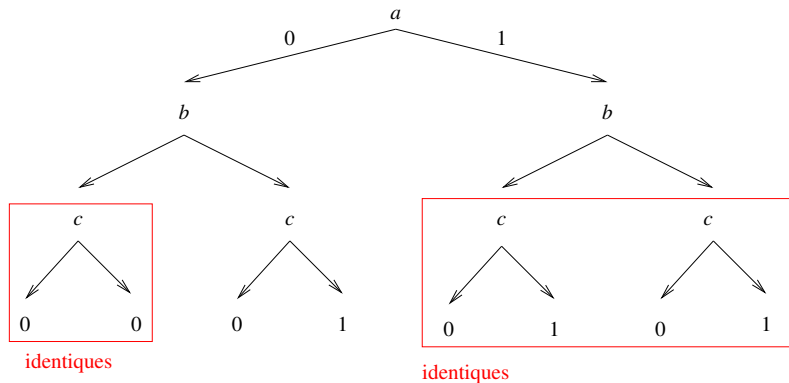
Binary decision diagrams

Given ordered Boolean variables (a, b, c) , represent $(a \wedge c) \vee (b \wedge c)$:

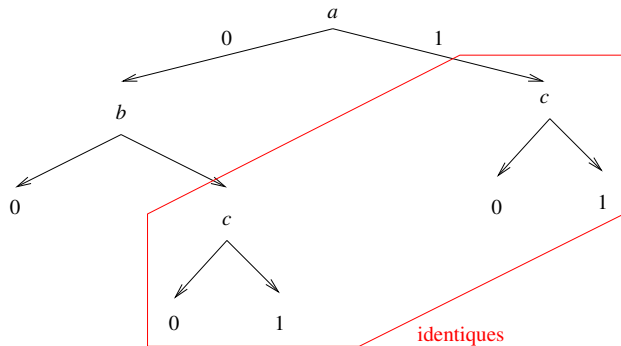


Removing useless nodes

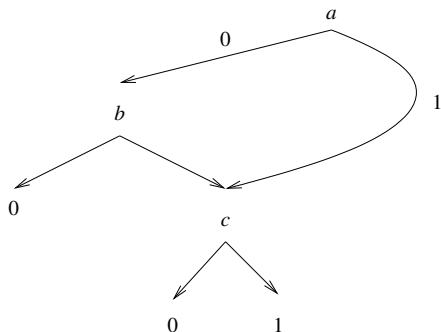
Silly to keep two identical subtrees:



Compression



Reduced BDD



Idea: turn the original tree into a DAG with **maximal sharing**.

Two different but isomorphic subtrees are never created.

Canonicity: a given example is always encoded by the same DAG.

Implementation: hash-consing

Important: implementation technique that you may use in other contexts

“Consing” from “constructor” (cf Lisp : cons).

In computer science, particularly in functional programming, hash consing is a technique used to share values that are structurally equal. [...] An interesting property of hash consing is that two structures can be tested for equality in constant time, which in turn can improve efficiency of divide and conquer algorithms when data sets contain overlapping blocks.

https://en.wikipedia.org/wiki/Hash_consing



Implementation: hash-consing 2

Keep a **hash table** of all nodes created, with hashcode $H(x)$ computed quickly.

If node = (v, b_0, b_1) compute H from v and unique identifiers of b_0 and b_1

Unique identifier = address (if unmovable) or serial number

If an object matching (v, b_0, b_1) already exists in the table, return it

How to collect garbage nodes? (unreachable)



Garbage collection in hash consing

Needs **weak pointers**: the pointer from the hash table should be ignored by the GC when it computes reachable objects

- ▶ Java WeakHashMap
- ▶ OCaml Weak



Garbage collection in hash consing

Needs **weak pointers**: the pointer from the hash table should be ignored by the GC when it computes reachable objects

- ▶ Java WeakHashMap
- ▶ OCaml Weak

(Other use of weak pointers: caching recent computations.)



Hash-consing is magical

Ensures:

- ▶ **maximal sharing**: never two identical objects in two \neq locations in memory
- ▶ ultra-fast equality test: sufficient to **compare pointers** (or unique identifiers)

And once we have it, BDDs are easy.



BDD operations

Once a variable ordering is chosen:

- ▶ Create BDD false, true(1-node constants).
- ▶ Create BDD for v , for v any variable.
- ▶ Operations \wedge , \vee , etc.



Binary BDD operations

Operations \wedge , \vee : recursive descent on both subtrees, with **memoizing**:

- ▶ store values of $f(a, b)$ already computed in a hash table
- ▶ index the table by the unique identifiers of a and b

Complexity with and without dynamic programming?



Binary BDD operations

Operations \wedge , \vee : recursive descent on both subtrees, with **memoizing**:

- ▶ store values of $f(a, b)$ already computed in a hash table
- ▶ index the table by the unique identifiers of a and b

Complexity with and without dynamic programming?

- ▶ without dynamic programming: unfolds DAG into tree \Rightarrow exponential
- ▶ with dynamic programming $O(|a|.|b|)$ where $|x|$ the size of DAG x



Quantifiers

BDD for formula F over variables x, y, z .

Want a BDD for formula $\exists x F$ over variables y et z .

$[\exists x F](y, z) \equiv F(0, y, z) \vee F(1, y, z)$: compute $F[0/x] \vee F[1/x]$ ($F[b/x]$ is F where x has been replaced by b).

Same for \forall but with \wedge .

Otherwise said **quantifier elimination**.



Back to transition systems

- ▶ The set Σ_0 of initial states is defined by a formula over $x_1, \dots, x_n \Rightarrow$ a BDD over n variables.
- ▶ The transition relation T over Boolean variables $x_1, \dots, x_n, x'_1, \dots, x'_n$ (x' = updated x) \Rightarrow a BDD over $2n$ variables.

Recall $\phi(X) = \Sigma_0 \cup R(X)$, in formulas:

$$\phi(X) = \Sigma_0 \vee (\exists x_1, \dots, x_n (X \wedge T))[x'_1/x_1, \dots, x'_n/x_n] \quad (2)$$

All operations doable on BDDs!



Iterative computations over BDDs

Compute sequence X_0, \dots with $X_0 = \Sigma_0$ and $X_{n+1} = \phi(X_n)$,
stop when $X_n = X_{n+1}$ (recall: ultra-fast equality test!)

(Or stop when X_i intersects bad states.)

Sounds very simple

but many possible optimizations and variants (e.g. **signed BDDs**), much work needed

In practice, need other operators (e.g. “constrain”,
“restrict” . . .)



Industrial use: hardware

Clocked hardware \simeq reset state + transition relation

Checking properties of circuits during conception (building prototypes is very expensive)

Tools such as Cadence-SMV



Bounded model checking

BDDs are too costly (worst-case exponential time and space)

Unbounded reachability in Boolean circuits is
PSPACE-complete

Idea: limit search to n steps, “only” NP-complete
Coming next : Bounded MC

