SAT-based bounded model checking

[Automated Reasoning, 2015/2016 1b — Lecture 6]

Doina Bucur

d.bucur@rug.nl

Dec 2015

In this lecture...

SAT and SAT solving

Modelling software. Unwinding transitions

- Unwinding transitions. Modelling real data
- Briefly about coding standards for embedded software

SAT-based bounded model checking (BMC)

- Basics of bounded model checking
- CBMC: Bounded Model Checker for ANSI-C

Assignment

This lecture focuses on **model checking for critical software**. We see (1) a precise way to translate software data types into a model, and (2) two practical ways to address state-space explosion: *bounded model checking*, which unrolls any loops in the system model only a bounded number of times, and *symbolic model checking*, which avoids constructing the state-space of a Kripke structure, and instead represents it by a formula in propositional logic.

You see these methods combined in the industrial-strength model checker **CBMC** (C Bounded Model Checker).

Introduction: The Boolean satisfiability problem (SAT)

The satisfiability (SAT) problem is that of taking any Boolean formula and determining whether the Boolean variables can be assigned to values so that the formula evaluates to **true**[1].

Examples (with a, b, c Boolean variables):

$$(a \vee b \vee c) \wedge (\overline{a} \vee b \vee \overline{c})$$

is satisfied for b equal to 1, and

$$(a \leftrightarrow 1) \wedge (b \leftrightarrow a)$$

(where $\leftrightarrow$ is the double implication[2]) is satisfied for a and b equal to 1.

[1] `http://en.wikipedia.org/wiki/Satisfiability_of_boolean_expressions`.

[2] Simple implication: $a \rightarrow b \equiv \overline{a} \vee b$

SAT complexity

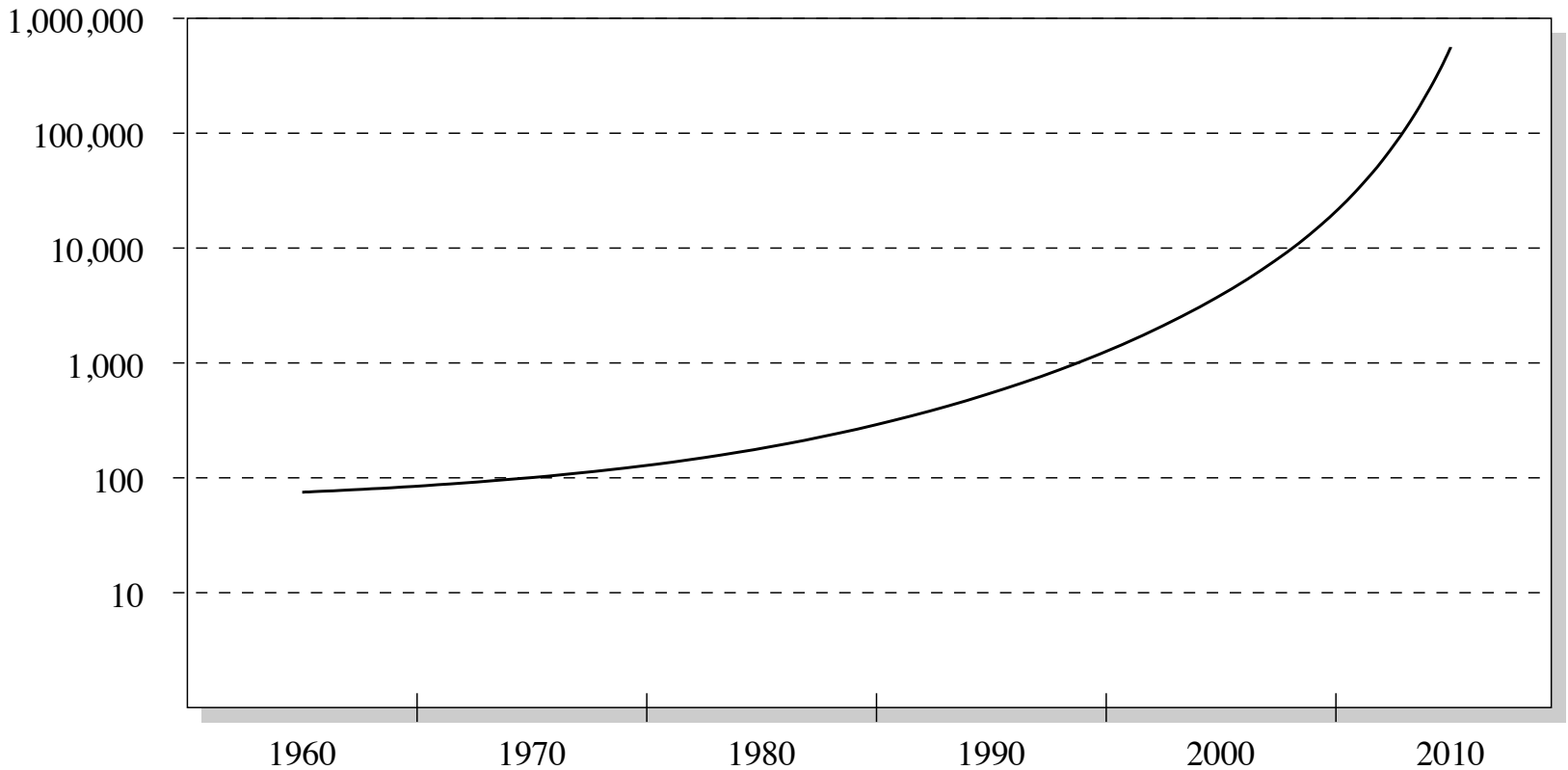
Solving SAT was the first problem proven to be NP-complete in the worst case: there can be no algorithm to efficiently solve all SAT instances.

However,

- SAT instances following certain patterns are solvable in polynomial time, and
- **SAT solvers** implement heuristics which work efficiently in practice for most instances.

SAT solvers (1)

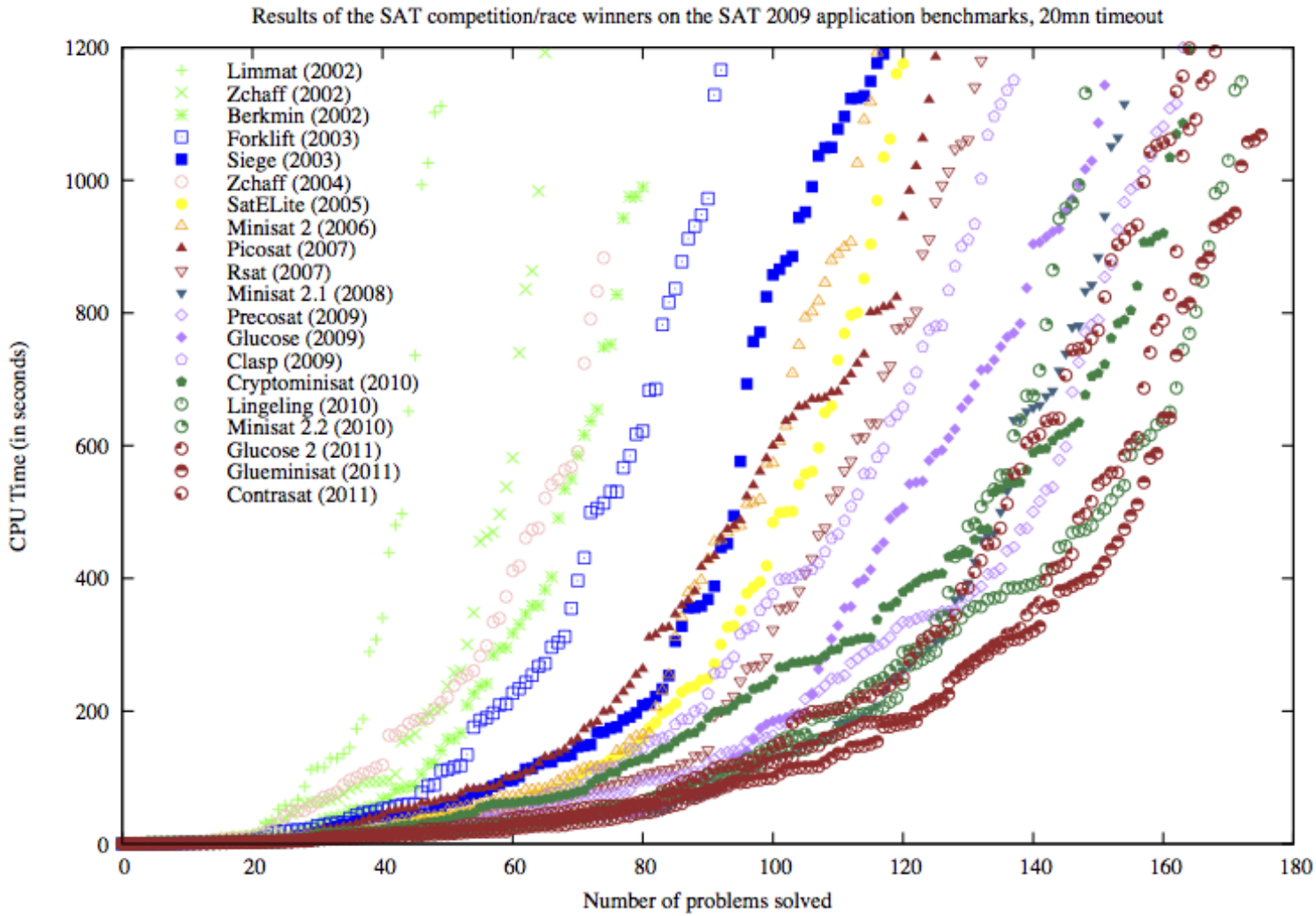
SAT solvers made progress (number of variables in solvable formula, per year)[3]:



[3] A qualitative graph. D. Kroening, 2008.

SAT solvers (2)

SAT solvers made progress (runtime per benchmark)[4]:



[4] http://www.satcompetition.org/

Modelling software (reminder)

A computer running software has:

- **data**, partitioned into **variables**, each taking value from a finite set; this includes
  - a program counter per thread,
  - variables on stack,
  - variables on heap,
  - of which some are pointers or data structures;
- **instructions** (sequential or concurrent) transforming data over time.

We already formalized (basic) data types and instructions using states and transitions in Kripke structures. Here, we add bit-level accuracy in the model.

Unwinding transitions in software (1)

In Lecture 2, you have seen a precise way to extract a transition relation out of software (we ignore the program counter for simplicity):

```
/* x, y start at 0 */
while (true)
    atomic { x = (x+1) mod 3, y = x+1; }
```

Write **initial state** s_0 in predicate-logic form:
$s_0 := (x = 0) \wedge (y = 0)$.

Write T, the **transition relation** for the loop iteration, in predicate-logic form, where primed variables are next-state variables:

$T := (x' = (x + 1) \text{mod } 3) \wedge (y' = x' + 1)$

Unwinding transitions in software (2)

To show how the infinite loop evolves in the next time step, you would **conjunct** s_0 and T, i.e. unwind the transition relation once:

(this is s_0)
$(x_0 = 0) \wedge (y_0 = 0)$

(the following is T)
$T := (x' = (x + 1) \text{mod } 3) \wedge (y' = x' + 1)$

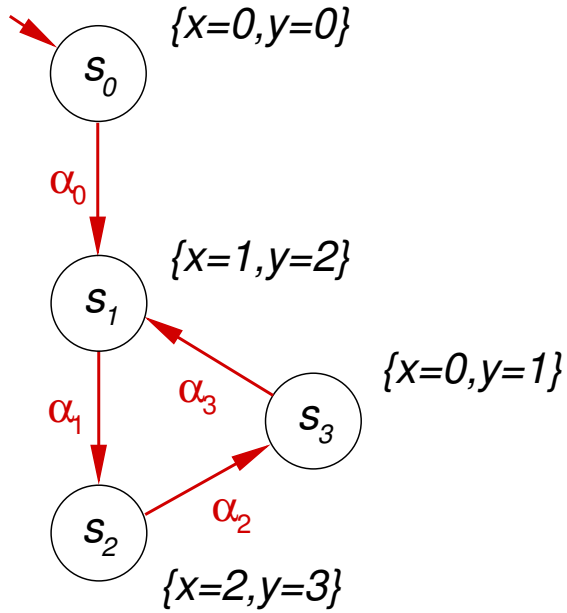
(by the conjunction $s_0 \wedge T$, we effectively model s_1:)
$(x_0 = 0) \wedge (y_0 = 0) \wedge$
$(x_1 = (x_0 + 1) \text{mod } 3) \wedge (y_1 = x_1 + 1)$

(if again conjuncted with T, we get s_2:)
$(x_0 = 0) \wedge (y_0 = 0) \wedge$
$(x_1 = (x_0 + 1) \text{mod } 3) \wedge (y_1 = x_1 + 1) \wedge$
$(x_2 = (x_1 + 1) \text{mod } 3) \wedge (y_2 = x_2 + 1)$

(and so on, forever! since I cannot see when I reached a preexisting state)

Unwinding transitions in software (3)

On the previous slide we have not computed the state space in the **Kripke structure** as we did in Lecture 2 (if we did, we'd have gotten the structure on the right). The model checking method which creates a Kripke structure is called *explicit-state model checking*.



In this lecture, we show a different model checking algorithm which only needs as model the initial state of the memory, and all program transitions, all in a single **logic formula**. This is called *symbolic model checking*.

Extracting models from software: data (1)

On computers, over byte-sized variables, $200 + 100 \neq 300$ (as we expect over the $\mathbb{N}, \mathbb{R}$ domains), but 44.

...because the **C-11**[5] **standard** dictates facts such as:

- When a computation involves unsigned operands, on overflow the result should be reduced modulo $2^{\text{type width}}$.
- For signed integer types, arithmetic is defined to use two's complement representation with silent wrap-around on overflow.

To model all data accurately, it is usual to model all constants and variables by bit vectors:

$$200 = \langle 11001000 \rangle$$

[5] `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`

Extracting models from software: data (2)

The result of arithmetics over bit-vectors depends on

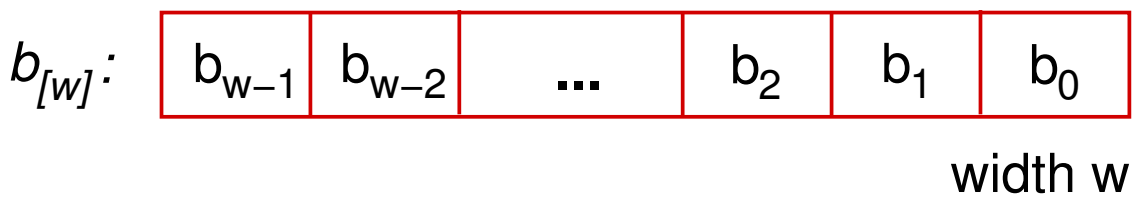
- **width**: 8, 16, 32, 64, etc;
- **encoding**: signed, unsigned, fixed point, floating point, etc.

Definition (Bit-vector)

A bit-vector $b_{[w]}$ is a vector of Boolean variables of length w,

$$b_{[w]} : \{0, \ldots, w-1\} \to \{0, 1\}$$

or



Translating real-world data types (of different widths and encodings) into bit-vectors is called bit blasting.

Extracting models from software: data (3)

We need to also carefully remodel **arithmetic operators** over bit-vectors of different widths and encodings. This is called flattening.

What should

$$==, <, >, \leq, \geq, ?:, |, \&, ||, \&\&, +, -, *, /, \%$$

be translated to, over vectors of Boolean variables $b_{[w]}$?

When the variable x is a bit vector, instead of a simple transition of the form $T := (x' = x + 1)$, we have much larger transition relations, but over elementary (boolean) variables.

Extracting models from software: data (4)

Take the bit vectors $A_{[8]}$, $B_{[8]}$, $C_{[8]}$ and $S_{[8]}$ corresponding to integer program variables A, B, C, S.

Assignment. The C program statement { `A = B;` } is now the transition:

$$[A_0 \leftrightarrow B_0] \wedge \ldots \wedge [A_7 \leftrightarrow B_7]$$

Boolean OR on unsigned integers. The C program statement { `A = B | C;` } is now the transition:

$$[A_0 \leftrightarrow (B_0 \vee C_0)] \wedge \ldots \wedge [A_7 \leftrightarrow (B_7 \vee C_7)]$$

Conditional on unsigned integers. The C program statement {`(S==0)?A=B:A=C;`} is now the transition:

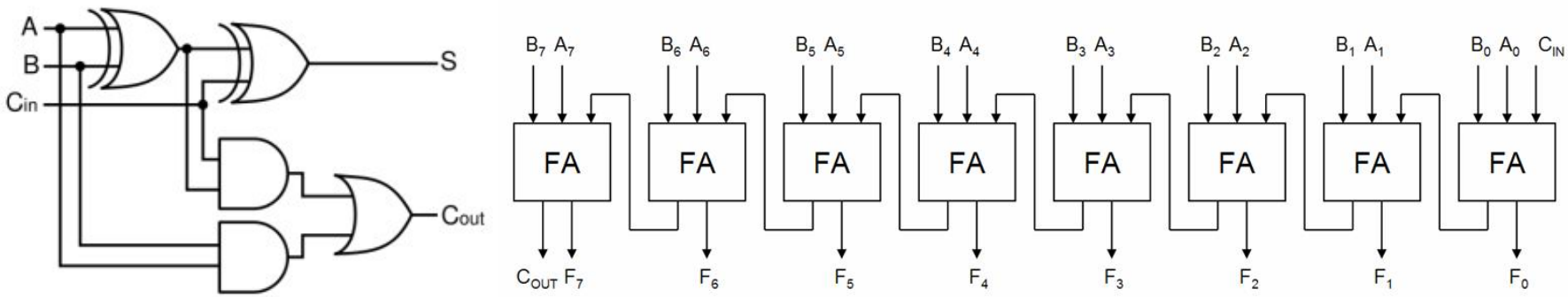
$$\left[\left(\bigwedge_i S_i \leftrightarrow 0\right) \wedge \left(\bigwedge_i A_i \leftrightarrow B_i\right)\right] \vee \left[\neg\left(\bigwedge_i S_i \leftrightarrow 0\right) \wedge \left(\bigwedge_i A_i \leftrightarrow C_i\right)\right]$$

Extracting models from software: data (5)

Arithmetic + on unsigned integers. The C program statement { `S = A + B;` } is now the transition:

$$(S_0 \leftrightarrow A_0 \oplus B_0) \wedge [S_1 \leftrightarrow (A_1 \oplus B_1) \oplus (A_0 \wedge B_0)] \wedge \ldots$$

Essentially, this formula mimics an 8-bit full adder:



Multiplication * results in very hard formulas.

Flattening for other data types requires knowledge of the data encoding, e.g., the IEEE 754 Standard for Floating-Point Arithmetic.

Motivation: Critical software

In the first lecture, I listed examples of **embedded** or **industrial systems** (microprocessors, spacecraft, railway systems, etc). Most of these are programmed in C. For such software:

- Reliability is central.
- The software development process and tools aid in achieving reliability by
  - imposing coding standards, and
  - using analysis, testing and formal verification tools.

An example of software development standard for ISO C is **MISRA C**, developed by the *Motor Industry Software Reliability Association*[6]. This standard is now a widely accepted best practice by developers in aerospace, telecom, medical devices, defense, railway, etc.

[6] `http://en.wikipedia.org/wiki/MISRA_C` (for an overview); Google for partial lists of the coding rules

Software standards: MISRA C

From the MISRA C, some of the rules related to **control flow**:

- The `goto` statement should not be used. Labels should not be used (except in `switch` statements).
- Functions shall not call themselves, either directly or indirectly.
- Numeric variables being used in a `for` loop for iteration counting should not be modified in the loop body.

These rules impose the use of only straightforward program **loops and recursion**.

As a consequence, model checkers targeted at embedded software can treat loops specially.

Bounded model checking

It is the most successful formal-analysis technique in the hardware industry, and also for embedded software (C-like languages).

Idea:

- Check realistically large code.
- Only check for safety properties. This means we need to exhaustively check for the reachability of "unsafe" states in the system (and not for the existence of ω-runs).
- Take all loops in the system model (recursive function calls, `while` loops, etc). For each loop, only search for unsafe states up to a certain depth bound k.

  Disadvantages:
  - Possible incomplete checking.

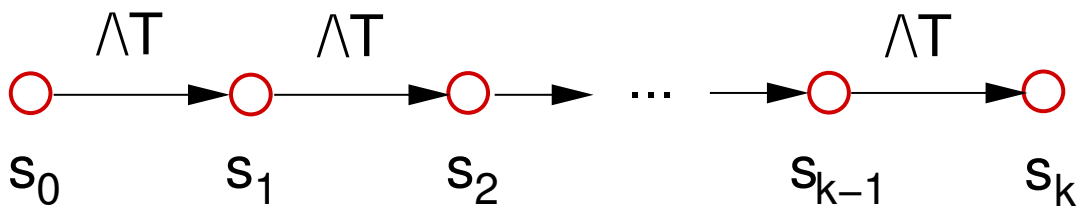
  Advantages:
  - Many subtle bugs were found in practice.
  - Combined with SAT solvers, extremely efficient and robust.

Unwinding of a loop

To only construct any transition system up to a given loop depth k, we simply concatenate the loop transition T to the initial state s_0 (both as propositional-logic formulae):

$s_0 \wedge T$ (for $k = 1$)
$\quad \wedge T$ (for $k = 2$, etc)

or:



or as formula in propositional logic:

$$C := s_0 \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

Bounded unwinding of loops

```
while (condition) {
    body
}
```

is unwound, for $k = 5$:

```
if (condition) {
    body
    if (condition) {
        body
        if (condition) {
            body
            if (condition) {
                body
                if (condition) {
                    body
                }
            }
        }
    }
}
```

Putting all together (1)

Below, an **uninitialized** program (left) is translated into a formula C on the right (the "constraint" upon the values of x and y)[7] via a rewriting showing each variable valuation (center):

```
x=x+y;
if(x!=1)
  x=2;
else
  x++;

assert(x<=3);
```

$\rightarrow$

```
x1=x0+y0;
if(x1!=1)
  x2=2;
else
  x3=x1+1;

x4=(x1!=1)?x2:x3;
assert(x4<=3);
```

$\rightarrow$

$C :=$ `x1=x0+y0` $\wedge$
`x2=2` $\wedge$
`x3=x1+1` $\wedge$
`x4=(x1!=1)?x2:x3`

$P :=$ `x4` ≤ 3

[7] This is from the CBMC tool paper, `http://www.cs.cmu.edu/~svc/papers/view-publications-ckl2004.html`. You can see the terms in the long Boolean conjunction that is this formula, with CBMC, if you type the program in `file.c` and run `cbmc --show-vcc file.c`.

Putting all together (2)

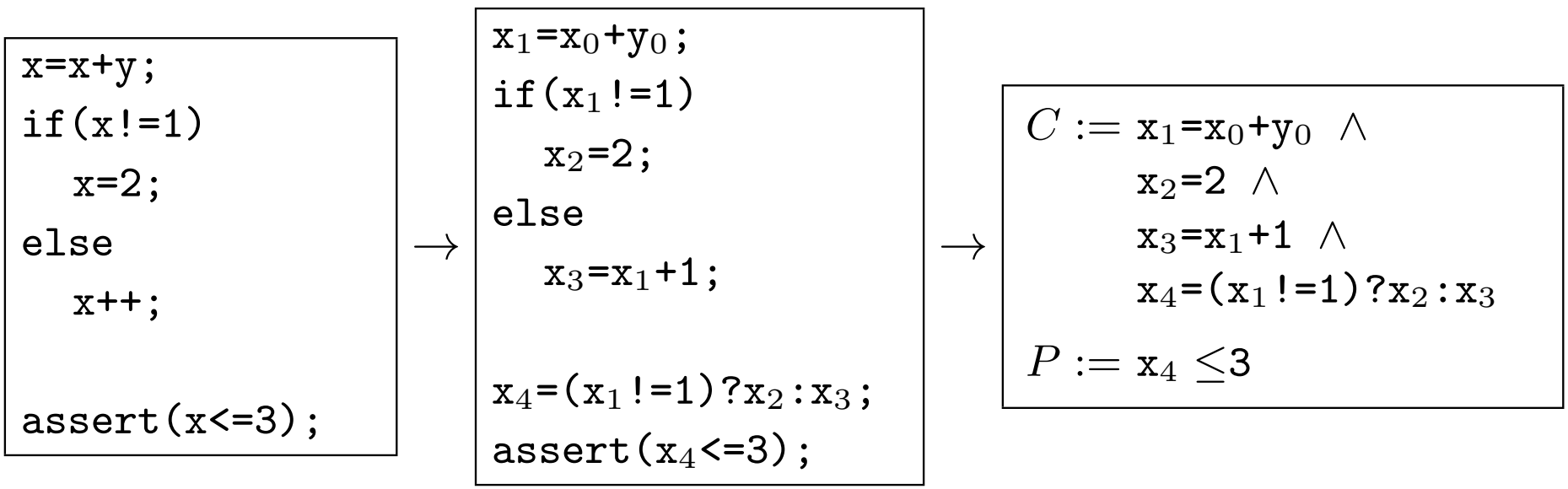
The resulting transition system is a boolean formula (but over standard, non-boolean variables, and including standard arithmetic operators: $==, <, >, \leq, \geq, |, \&, ||, \&\&, +, -, *, /, \%$).

Next step: **bit-blast** the variables, **flatten** the arithmetic operators, and obtain a Boolean formula over Boolean variables. A model checker like CBMC will do this expansion internally.

Next step: "add" the **specification** to the formula above (we see this in what follows).

Adding the specification: some intuition

Below, an assertion is written at the end of the code; it is also translated to an appropriate formula P on the right:



Adding the specification, more formally

We want to check a safety property, say, an assertion `assert(x<=3);` placed somewhere in the code. The model checker rewrites the expression $P := (x \leq 3)$ to use the correct valuation of x (on the previous slide, $P := (x_4 \leq 3)$).

Then, the model checker will search for a counterexample. For a counterexample to exist to property P, the negated property $\neg P$ has to hold in that state, i.e.

$$\text{if } C \wedge \neg P \text{ holds, there exists a counterexample}$$

A SAT solver is called over $C \wedge \neg P$. Done.

Completeness

Look again at the bounded loop unwinding. CBMC adds an unwinding assertion to (dis)prove that we have done enough unwinding:

```
if (condition) {
    body
    if (condition) {
        body
        if (condition) {
            body
            assert(!condition);
        }
    }
}
```

Some final statements:

- For finite loops, there always exists a completeness cutoff k, after which there need to be no more unwinding.
- However, deciding on the value of k is as difficult as doing model checking without unwinding bounds.
- In practice, these thresholds are overapproximated.

CBMC: Bounded Model Checker for ANSI-C

Home page: `http://www.cprover.org/cbmc/`.

Installation: `http://www.cprover.org/cprover-manual/installation-cbmc.shtml`

Manual: `http://www.cprover.org/cprover-manual/`; you will need section 1, **Introduction**, section 3, **CBMC**, and section 5, **Modelling**.

Publication: `http://www.cs.cmu.edu/~svc/papers/view-publications-ckl2004.html`. This describes the tool in brief.

The developer of CBMC, Prof. D. Kroening (Oxford University), received an award from the software testing and verification research community for "[promoting] the industrial adoption of formal software verification more than any other tool in existence"[8].

[8] `http://www.research.ibm.com/haifa/conferences/hvc2011/award.shtml`

Bibliographical notes

SAT-based bounded model checking is not covered in the three books you've seen. However, it is a fairly modern flavour of model checking, particularly well applied in the hardware and software industry, and one of E. M. Clarke's ongoing research interests. At the time of writing the *Model Checking* book (E. M. Clarke, O. Grumberg, D. A. Peled), the topic was not mainstream; there will be a chapter on SAT in the second edition of this book.

Have a look instead at the original paper introducing this model-checking technique: `http://dl.acm.org/citation.cfm?doid=309847.309942`, *Symbolic model checking using SAT procedures instead of BDDs*, Bierre, Cimatti, Clarke, Fujita and Zhu, in DAC '99, Proceedings of the 36th annual ACM/IEEE Design Automation Conference.

D. Kroening has a book on **checking algorithms over bit-vectors**: `http://www.decision-procedures.org/`.

Assignment (introduction)

This assignment is entirely practical, and has you play with CBMC, the model checker for C which implements the BMC technique with SAT-solving internally. This has the purpose of introducing you to new types of **software errors**, and to software model checking, also known as *software verification*.

Assignment (CBMC manual)

See slide 26; install yourselves the CBMC binary (the source code, C++, is open and available on the same page). Type `cbmc --help` for a start. Go through the manual:

- Section 1 is a brief introduction, both to CBMC and to a 'sister' model checking tool, SATABS (which you can ignore). You see in this section that CBMC can take in input not only ANSI-C, but also other languages.
- Section 5 tells you how to write specifications (with `assert`), assign nondeterministic values to program variables (with `nondet***` functions), and restrict nondeterministic values (with `assume`).
- Section 3 overviews CBMC with practical examples of use. You see here that CBMC generates assertions (called *properties*) **automatically** for **standard programming errors** (array out-of-bounds access, null-pointer dereference, over- and underflows, division by zero), if instructed by a flag. You can list these generated assertions with, e.g., `cbmc a.c --bounds-check --show-properties` (for out-of-bounds access), and can verify one with `cbmc file.c --bounds-check --property your-property-id`.

Assignment (software bugs)

(72 ) For each of the types of programming scenarios below[9], write (or find) a small C or C++ program, add the required specification against the error (only if CBMC doesn't already do it for you), and use CBMC to get a counterexample showing the error. Name the type of software bug. Say whether each verification run is *complete* (slide 25). Say what the effect of *executing* each buggy program would be.

1. The program writes into a local string (allocated on the stack) one character more than expected.
2. The same for a string allocated on the heap (with `malloc` or `new`).
3. The program tries to read the memory pointed by a global, uninitialized pointer variable.
4. A `float` variable is divided by a variable whose value is currently zero.
5. A 16-bit signed integer overflow.
6. A float underflow[10].

[9] For an overview of programming bugs, see also http://en.wikipedia.org/wiki/Software_bug, section "Common types of computer bugs".

[10] One note: in the current version, CBMC "forgets" to add the required assertion for underflow when you give it `--signed-overflow-check`; add it yourselves manually.

Assignment (software bugs)

(38 ) Integer overflow can happen in the well-known binary search algorithm. See the blog post: `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.

Use CBMC to get a counterexample showing that error.

You may be surprised at how little computation time is needed by the tool, even if you use the general integer type, and nondeterministic initialization of most variables, including the array contents.

If you are curious: the webpage `http://sv-comp.sosy-lab.org/2013/benchmarks.php` (see the first link on the page) has many real-world benchmarks for CBMC.

Assignment (applications of C model checking)

(50 ) Choose any one of the following papers to read:

- The USENIX workshop on Systems Software Verification paper *Comparing Model Checking and Static Program Analysis: A Case Study in Error Detection Approaches*[11]. This compares benchmarks of CBMC and a static-analysis tool with regard to bug finding, on large code bases.
- Any of the many practical applications of CBMC at `http://www.cprover.org/cbmc/applications.shtml`. If in doubt, choose from the section on *Further Practical Applications and Experience Reports*.

For the paper chosen, write a half-page (critical) summary.

[11] `http://www.usenix.org/events/ssv10/tech/full_papers/Vorobyov.pdf`

Assignment administration

Handing in:

- in lab session or electronically;
- either way, before the end of Mon Jan 11.

Grading:

- this assignment adds up to 160  , or 160 grade centipoints.