TECHNISCHE
UNIVERSITÄT
WIEN

Software Verification

From programs to complex systems

Luca Di Stefano

University of Parma, 9 December 2024

$ whoami

2016–2020	PhD at GSSI, L'Aquila, Italy
	• Modelling collective adaptive systems • Formal verification of CAS models
2020–2022	Postdoc at INRIA Grenoble, France
	• Compositional verification of CAS • Support more expressive properties
2022–2024	Postdoc at GU/Chalmers, Göteborg, Sweden
	• Model-checking agents with reconfiguration • Reactive synthesis over infinite-state arenas
now	Postdoc at TU Wien, Austria
	• All of the above ☺

Get in touch: `luca dot di dot stefano at tuwien.ac.at`

https://lucadistefano.eu

Overview (1/3)

Verification: Rigorous assessment of the correctness of a system

Software: The system is a program

```c
1 int divBy2(int n) {
2   return n/2
3 }
4
5 int main() {
6   int x
7   int y = divBy2(x)
8   assert(y * 2 == x)
9 }
```

`x` stores an arbitrary integer value (ISO C std. 6.2.4.5)

Can the program reach line 8 and violate `y*2==x`?

- No: return PASS
- Yes: return FAIL + a sequence of steps leading to the violation (counterexample)

Overview (2/3)

```
CBMC version 6.4.0 (cbmc-6.4.0) 64-bit arm64 macos
[...]
** Results:
divBy2.c function main
[main.assertion.1] line 8 assertion y * 2 == x: FAILURE

Trace for main.assertion.1:

State 16 file divBy2.c function main line 6 thread 0
----------------------------------------------------
  return_value_nondet=3 (00000000 00000000 00000000 00000011)
[...]
State 26 file divBy2.c function main line 7 thread 0
----------------------------------------------------
  y=1 (00000000 00000000 00000000 00000001)

Violated property:
  file divBy2.c function main line 8 thread 0
  assertion y * 2 == x
  y * 2 == x
```

Overview (3/3)

```
1  int divBy2(int x) {
2    return x/2
3  }
4
5  int main() {
6    int x = *
7    assume(x % 2 == 0)
8    int y = divBy2(x)
9    assert(y * 2 == x)
10 }
```

`assume(cond)` restricts analysis to executions where `cond != 0`

Useful to prune out unwanted counterexamples, model the environment in which the code will run, etc.

```
CBMC version 6.4.0 (cbmc-6.4.0) 64-bit arm64 macos
Type-checking divby2.safe
[...]
** Results:
divby2.safe.c function main
[main.assertion.1] line 9 assertion y * 2 == x: SUCCESS
```

Model checking

One way to implement formal verification

Given formal representations of the system *and* of what makes it correct, exhaustively explore the former and look for violations

Essentially, it's proof by lack of counterexamples

- ☺ Fully automated
- ☺ Can be applied in many domains (HW, SW, protocols, . . .)
- ☺ Works well with concurrency
- ☹ Mainly scalability (we'll see)
- ☹ Some expertise required

There are other ways. . .

Testing

- ☺ Very widespread
- ☺ Can be surprisingly effective (e.g., fuzzing)

- ☹ Cannot prove correctness
- ☹ Concurrency bugs?

Theorem proving

- ☺ Can exploit sophisticated tactics
- ☺ High expressiveness

- ☹ Typically only semi-automated
- ☹ Requires expert knowledge

What about abstract interpretation?

Roots

AbsInt: collecting semantics and lattice theory

ModChk: operational semantics and modal logic

Goals

AbsInt: building static analysers

ModChk: proving properties

In practice, they are routinely used together
(more on that later)

Model Checking in a nutshell

Input:

1. A Kripke structure $\mathcal{M}$
2. A property φ describing “good” computations

Checks whether $\mathcal{M} \models \varphi$

> “φ holds in $\mathcal{M}$”
> “$\mathcal{M}$ models/is a model for φ” (hence the name)

Output: PASS, or FAIL + counterexample

Caution

The term “model” creates lots of confusion. . .

Kripke structure

Assume you have a (finite) set AP of atomic propositions. Each $a \in AP$ represents a basic "fact", e.g., "we are at line 8" or "the value of `x` is 0".

Then a Kripke structure is $\langle S, I, R, L \rangle$

- S: States (finite)
- $I \subseteq S$: Initial states
- $R \subseteq S \times S$: Transition relation (total[1])
- $L : S \to 2^{AP}$: Labelling function: $L(s)$ tells you which APs hold in state s

[1] I.e., every state has at least one outgoing transition

(Linear) Temporal Properties

Consider paths through $\mathcal{M}$ rooted in an initial state

R is total $\Rightarrow$ Infinite-length paths $\pi = s_1 s_2 s_3 \ldots$
with $s_i \, R \, s_{i+1}$ for every i (aka $s_i \rightarrow s_{i+1}$)

A property describes how good paths should be.
Model checking = look for bad paths

Paths are just ordered sequences of states, hence "linear" and "temporal"

(Not the only logical framework)

LTL in a nutshell (1/2)

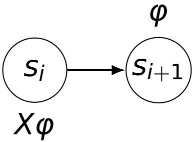
A logic for linear temporal properties φ

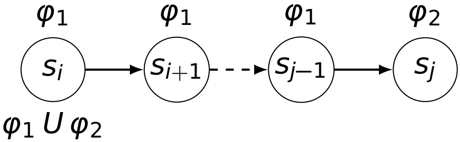
When does a state s_i in a path satisfy φ? ($s_i \models \varphi$)

$true$	always
a	iff $a \in L(s_i)$
$\neg\varphi$	iff $s_i \not\models \varphi$
$\varphi_1 \land \varphi_2$	iff $s_i \models \varphi_1$ and $\land\ s_i \models \varphi_2$

Ok, but where is the temporal part?

$$
s_i \models \begin{cases}
X\varphi & \text{iff } s_{i+1} \models \varphi & \textbf{[next]} \\
\varphi_1 \, U \, \varphi_2 & \text{iff } \exists j \geq i . s_j \models \varphi_2 \wedge & \\
 & \wedge \; \forall k . i \leq k < j \Rightarrow s_k \models \varphi_1 & \textbf{[until]} \\
F\varphi & \text{same as } \mathit{true} \, U \, \varphi & \textbf{[finally]} \\
G\varphi & \text{same as } \neg F \neg \varphi & \textbf{[globally]}
\end{cases}
$$





Path $\pi = s_1 s_2 \ldots$ satisfies φ iff $s_1 \models \varphi$

Explicit-state model checking 1/2

$\varphi = G(\texttt{lineIs8} \Rightarrow \texttt{yTimes2Eqx})$

```
1   int divBy2(int x) {
2     return x/2
3   }
4
5   int main() {
6     int x;
7     int y = divBy2(x)
8     assert(y * 2 == x)
9   }
```

Initial steps:

1. Turn program into a Kripke Structure $\mathcal{M}$
2. Negate the property: $F(\texttt{lineIs8} \wedge \neg\texttt{yTimes2Eqx})$

3. Now turn the negated property into a Büchi automaton $\mathcal{A}$. These are automata that recognize infinite words (*ω-regular*). A word is accepted if it makes $\mathcal{A}$ visit an accepting state infinitely many times.

Explicit-state model checking 2/2

4. Explore the synchronous product $\mathcal{M} \otimes \mathcal{A}$ (Intuitively, this captures how $\mathcal{A}$ evolves when fed paths over $\mathcal{M}$)
5. If you find a path in $\mathcal{M} \otimes \mathcal{A}$ that loops through an accepting state, it represents a path in $\mathcal{M}$ that violates φ (counterexample). Thus, $\mathcal{M} \not\models \varphi$
6. Otherwise, $\mathcal{M} \models \varphi$

Explicit-state = Direct representation of $\mathcal{M}$

Complexity of LTL model checking

$$\mathcal{O}(|\mathcal{M}| \cdot 2^{|\varphi|})$$

☹ The automaton construction is exponential

☹ $|\mathcal{M}|$ ~doubles for each added *AP*
(state space explosion problem)

Many attempts at mitigation

- On-the-fly MC: only keep portions of $\mathcal{M}$ in memory
- Compositional MC: split $\mathcal{M}$, solve smaller problems, compose these together
- Symmetry reductions

Symbolic Model Checking

A way to overcome state space explosion

Define your system/program as:

- A vector of n (finite-state) variables $\mathbf{x} = x_1, \ldots, x_n$
- A predicate $init(\mathbf{x})$ that describes the initial states
- A set of n functions $next(x_i) = f_i(\mathbf{x})$ describing how x_i changes from one state to the next

Explicit-state MC = enumerate all initial states, use *next* to compute successors, construct $\mathcal{M}$,...

Symbolic MC = directly manipulate *init*, *next*

Symbolic Model Checking

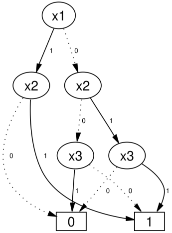
For simplicity let $\mathbf{x}$ a vector of Booleans

- *init*($\mathbf{x}$) is already a Boolean function
- We can always express the system $next(x_1), \ldots, next(x_n)$ as

$$R(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$$

such that $\mathbf{x}'$ is a successor of $\mathbf{x}$ iff $R(\mathbf{x}, \mathbf{x}') = true$

We can store/manipulate these (and any Boolean function) with efficient data structures called Binary Decision Diagrams (BDDs)



Picture credit: Wikipedia

Model-checking $Gp(\mathbf{x})$, symbolically

Intuitively: first compute BDD for all reachable states, then intersect with negated p

```
states = BDD(false) // BDD for an empty set
frontier = BDD(init)
tr = BDD(R)
notP = BDD(¬p)
do {
  // Update visited states
  states = states ∨ frontier
  // Update frontier
  frontier = Image(states, tr) ∧¬states
} while (frontier not empty)
return PASS if (states∧notP is empty) else FAIL
```

Image(`states`, `tr`) is the (BDD for the) set of successors of `states` according to `tr`

(Checking BDDs for emptiness is easy)

Final notes on BDDs

- ☺ Can be generalized to all of LTL[a]
  Intuitively, fixed-point computation is guided by the "shape" of the property
- ☺ Impressive advance in hardware domain:
  "10^{20} States and Beyond" in 1990 (!)
- ☹ BDDs also become cumbersome
- ☹ Ordered BDDs mitigate this but:
  1. Finding a good variable ordering is hard
  2. Some functions always yield a BDD of exponential size
- ☹ Still finite-state!

[a] Actually, "standard" algorithms are based on branching-time logics that are a superset of LTL

Bounded model checking

States reachable within k steps:

$$Reach_k = init(\mathbf{x}^{(1)}) \land R(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) \land \ldots \land R(\mathbf{x}^{(k-1)}, \mathbf{x}^{(k)})$$

where each $\mathbf{x}^{(1)}$ is a vector of Boolean variables

To verify safety ($Gp(x)$):

1. Consider $P = p(\mathbf{x}^{(1)}) \land \ldots \land p(\mathbf{x}^{(k)})$
2. Solve $Reach_k \land \neg P \ldots$ Using a SAT solver!

   SAT Counterexample found (a reachable state where $\neg p$), system is unsafe

   UNSAT Bounded system is safe, cannot say anything about the whole system (underapproximation)

```
x=x+y;
if(x!=1)
  x=2;
else
  x++;
assert(x<=3);
```

$\longrightarrow$

```
x1=x0+y0;
if(x1 != 1)
  x2=2;
else
  x3=x1+1;
x4=(x1!=1)?x2:x3;
assert(x4<=3);
```

$\longrightarrow$

$$
\begin{aligned}
C :=\ & \mathrm{x}_1 = \mathrm{x}_0 + \mathrm{y}_0 \land \\
& \mathrm{x}_3 = \mathrm{x}_1 + 1 \land \\
& (\mathrm{x}_1 \neq 1) \Rightarrow \mathrm{x}_4 = \mathrm{x}_2 \land \\
& \neg(\mathrm{x}_1 \neq 1) \Rightarrow \mathrm{x}_4 = \mathrm{x}_3 \\
P :=\ & x_4 \leq 3
\end{aligned}
$$

1. C code (+ assertions)
2. Static single assignment (SSA) pass
3. SAT formula ($C \land \neg P$)
   (Encode +,-,$\star$... as Boolean circuits)

---

[2] Adapted from Clarke et al. 2004

From C to SAT: Reduction

- Function calls are inlined
- Loops are unwound: apply k times

$$\texttt{while}(e)\ \{P\} \Rightarrow \texttt{if}(e)\ \{P\};\ \texttt{while}(e)\ \{P\}$$

  (ignore last `while`)
- Similar approach for recursive function calls & backwards `goto`s
- During unwinding, pointer dereferences (`&p`) are substituted with their variables

```
int a, b, *p;
if(x) p=&a; else p=&b;
*p=1;
```

→

```
int a, b, *p;
if(x) p=&a; else p=&b;
if(x) a=1;  else b=1;
```

Pros and cons of BMC (so far)

- ☺ Rapid progress in SAT ⇒ Very efficient
- ☺ Can scale to the complexity of real software
- ☺ Minimal, precise counterexamples
- ☹ Bounded analysis
- ☹ Tailored for safety checking
  (All of LTL may be reduced to safety-checking an appropriate automaton, but this has a cost)
- ☹ Still finite state

Why infinite-state matters

. . . Aren't all computers finite-state? Yes, but unbounded things are not uncommon in software

- "bignum" types, strings, recursive structs. . .
- Dynamic memory management (`malloc`, `free`)
- Process/thread creation and destruction (`fork`, `pthread_create`)

Treating these unknowns as ranging over ∞ domains might be more elegant and potentially more efficient

However, need formal tools able to handle these domains

SMT-based BMC

SMT = Satisfiability Modulo Theories

- Solver is not limited to Booleans
- Can reason about variables of certain types for which a suitable theory exists (= formal description of operators on these variables)
- Example: LIA = theory of integers with linear arithmetic (+, −, but no multiplication)

☺ We can implement BMC tools that encode `int`s as integers, `float`s as reals... and then use an SMT solver ⇒ Verification over infinite state spaces

☹ Many interesting theories are undecidable

Predicate abstraction

Another way to handle large/infinite state spaces

- Define a set of predicates $p_1, \ldots p_n$ over $\mathbf{x}$
- These induce (at most) 2^n abstract states $s_i^\sharp$, i.e., from $(\neg p_1, \ldots, \neg p_n)$ to $(p_1, \ldots, p_n)$
- We add a transition $s_i^\sharp \to s_j^\sharp$ whenever a concrete state $s \in s_i^\sharp$ can transition into $s' \in s_j^\sharp$
- Similarly abstract *init* and φ
- Use a procedure for finite-state MC

☺ Sound (it is an abstract interpretation after all!)

☹ Overapproximation. What if the MC step FAILs?

CEGAR

(Counterexample-Guided Abstraction Refinement)

1. Build initial abstraction $\mathcal{M}^{\sharp 0}, \varphi^{\sharp 0}$
2. Check if abstract system $\mathcal{M}^{\sharp 0}$ satisfies $\varphi^{\sharp 0}$
3. If SAFE, exit (SAFE).
4. If FAIL with counterexample $\pi^{\sharp 0}$:

   If it can be concretised, exit (FAIL).

   Otherwise (spurious):

   a. Find at what step $\pi^{\sharp 0}$ becomes spurious
   b. Extract new predicates with this information
   c. Compute a new abstractions $\mathcal{M}^{\sharp 1}, \varphi^{\sharp 1}$
   d. Go back to square 2.

- ☺ Fully automated (we can extract pr. from $\mathcal{M}, \varphi$)
- ☹ Sensitive to which predicates are used
- ☹ Some properties may need ∞ refinements

Applications to CAS

(Collective Adaptive Systems)

Collections of concurrent agents that interact with each other and adapt to changes

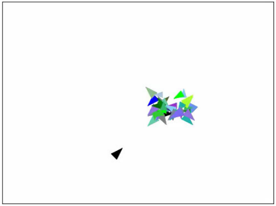
1. Collective behaviour emerges from local choices
2. Their evolution is hard to predict and reason about
3. Most modelling tools only focus on simulation
4. Can tools from SW verification help?

Our approach

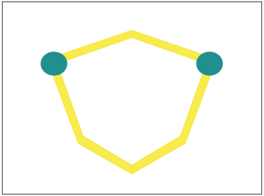
- Describe system in a high-level DSL
- Attribute-based interaction: agents observe and react to other agents' exposed variables
- Structural encoding of the system as a C program
- Sequentialization: concurrent system → sequential program (+ additional nondeterminism)

Takeaways

- Relatively low-effort
- Not limited to our language
- Can benefit from progress in SW verification
- Also suitable for simulation
  - Use a dummy assertion that fails after *k* steps
  - Give program to a SAT-based BMC
  - Randomize the behaviour of the SAT solver to get different traces



Flocking behaviour after disruption by a bird of prey



Ant colony determining the shortest path to a food source

(Some) Resources

Model checkers for C

CBMC	https://github.com/diffclue/cbmc
CPAchecker	https://cpachecker.sosy-lab.org/
ESBMC	https://github.com/esbmc/esbmc
UAutomizer	https://ultimate-pa.org/automizer

SAT/SMT solvers

KissSAT	https://github.com/arminbiere/kissat/
Z3	https://github.com/Z3Prover/z3
MathSAT	https://mathsat.fbk.eu/
CVC5	https://cvc5.github.io/

Competitions

SV-COMP	https://sv-comp.sosy-lab.org/
SAT	https://satcompetition.github.io/
SMT-COMP	https://smt-comp.github.io/2024/

Conclusions

- Long history of MC success in the HW domain
- Increasingly able to tackle real-world SW

  MS Windows Driver Foundation (early 2000s)
  NASA Mars rovers (2004)
  Boot code in AWS data centres (2018)
- Advantages from mixing multiple formal methods
- We can use SW MCs as backends (I know I do ☺)
- Every ☹ is a topic of active research

References I

Christel Baier and Joost-Pieter Katoen. Principles of Model Checking. MIT Press, 2008.

Dirk Beyer and Andreas Podelski. “Software Model Checking: 20 Years and Beyond”. In: Principles of Systems Design. 2022. DOI: 10.1007/978-3-031-22337-2_27.

Armin Biere et al. “Symbolic Model Checking without BDDs”. In: TACAS. 1999. DOI: 10.1007/3-540-49059-0_14.

Guillaume P. Brat et al. “Experimental Evaluation of Verification and Validation Tools on Martian Rover Software”. In: Formal Methods Syst. Design 25.2-3 (2004), pp. 167–198. DOI: 10.1023/B:FORM.0000040027.28662.a4.

Jerry R. Burch et al. “Symbolic Model Checking: 10^{20} States and Beyond”. In: LICS. 1990. DOI: 10.1109/LICS.1990.113767.

Edmund Clarke, Daniel Kroening, and Flavio Lerda. “A Tool for Checking ANSI-C Programs”. In: TACAS. 2004. DOI: 10.1007/978-3-540-24730-2_15.

Edmund M. Clarke. “The Birth of Model Checking”. In: 25 Years of Model Checking. 2008. DOI: 10.1007/978-3-540-69850-0_1.

References II

Edmund M. Clarke, Daniel Kroening, and Karen Yorav. “Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking”. In: DAC. 2003. DOI: 10.1145/775832.775928.

Edmund M. Clarke et al. “Counterexample-Guided Abstraction Refinement for Symbolic Model Checking”. In: J. ACM 5 (2003). DOI: 10.1145/876638.876643.

Edmund M. Clarke et al., eds. Handbook of Model Checking. 2018. DOI: 10.1007/978-3-319-10575-8.

Rocco De Nicola et al. “Modelling Flocks of Birds and Colonies of Ants from the Bottom Up”. In: STTT 25 (2023). DOI: 10.1007/s10009-023-00731-0.

Luca Di Stefano, Rocco De Nicola, and Omar Inverso. “Verification of Distributed Systems via Sequential Emulation”. In: TOSEM 3 (2022). DOI: 10.1145/3490387.

Susanne Graf and Bernhard Steffen. “Compositional Minimization of Finite State Systems”. In: CAV. 1990. DOI: 10.1007/BFb0023732.

References III

Ranjit Jhala and Rupak Majumdar. “Software Model Checking”. In: ACM Comput. Surveys 41.4 (2009), 21:1–21:54. DOI: 10.1145/1592434.1592438.

Viktor Schuppan and Armin Biere. “Efficient Reduction of Finite State Model Checking to Reachability Analysis”. In: 5 (2004). DOI: 10.1007/s10009-003-0121-x.

Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. “Checking Safety Properties Using Induction and a SAT-Solver”. In: FMCAD. 2000. DOI: 10.1007/3-540-40922-X_8.