Model Checking of Software

Patrice Godefroid

Bell Laboratories, Lucent Technologies
A Brief History of Model Checking

- Prehistory: transformational programs and theorem proving
- Early 80’s: foundations (the pioneers)
- Late 80’s: first tools… (the first champions)
- Early 90’s: reality check: state spaces explode! (the engineers)
- Late 90’s: the boom… then reality check… (the entrepreneurs)
- Next: can model checking be applied to software?
  - Challenges and approaches currently being investigated…
Preliminaries: Formal Verification + Disclaimer

- What is Verification? 4 elements define a verification framework:

  ![Verification Diagram]

  **Verification**: to check if **all** possible behaviors of the implementation are compatible with the specification

- While testing can only find errors, verification can also prove their absence (=exhaustive testing).

- Disclaimer: emphasis on technical ideas, not references…
Prehistory (70’s and before)

• “Transformational” programs:
  – Most early computer programs were designed to compute something.
  – Examples: accounting, scientific computing, etc.
  – Transformation from initial to final state.
  – Specification= pre-condition/post-condition

• Formal verification:
  – Paper and pencil.
  – Using Theorem Proving (first CAV tools)
Theorem Proving

• Goal: automate mathematical (logical) reasoning.

• Verification using theorem proving:
  – Implementation represented by a logic formula I (ex: Hoare’s logic).
  – Specification represented by a logic formula S.
  – Does “I implies S” hold?
  – Proof is carried out at syntactic level.

• This approach is very general.
  – Many programs and properties can be checked this way.

• However, most proofs are not fully automatic.
  – A theorem prover is rather a proof assistant and a proof checker.
Early 80’s: Foundations (the pioneers)

- From transformational programs to **reactive programs**:
  - A transformational program *computes* something.
  - A reactive program *controls* something.
  - Examples: telephone switch, airplane, ATM, power plant, pacemaker, etc.

- A reactive program continually interacts with its environment.

- Viewed as a FSM (automaton), called the *state space*.

- Behavior described in terms of sequences of states/transitions.

- Language for temporal properties: **Temporal Logic** [Pnueli,...]
Temporal Logic Model Checking

• Example: Linear-time Temporal Logic (LTL)
  – Specify properties of infinite sequences of states (or transitions).
  – Temporal operators include: G (always), F (eventually) and X (next).
  – Example: G(p -> Fq)
    \[ \begin{array}{c}
    s_0 \\
p \\
q \\
\end{array} \]

• “Does M satisfy f ?” = model checking [Clarke, Emerson, Sifakis,…]
  – For f in LTL, do all infinite computations of M satisfy f?
  – For f in BTL, does the computation tree of M satisfy f?

• Algorithmic issues: efficient decision procedures exist…
  – Proof can be carried out at semantic level, via state-space exploration.
  – Ex: for CTL, SAT is EXPTIME-complete, but model checking is linear!
First Model-Checking Frameworks

- 4 components define a model-checking framework:
  - Implementation (program) = an FSM.
  - Specification (property) = a temporal logic formula.
  - Comparison Criteria = defined by semantics of the temporal logic.
  - Algorithm = evaluates the formula against the FSM (“model-checking algorithm”)

- Model-Checking Research in the 80’s:
  - Various temporal logics: linear-time, branching-time,…
  - Relationship between temporal logics and classes of automata (LTL and word automata; BTL and tree automata…)
  - Classes of temporal properties (safety, liveness,…)
  - Etc.

- Model checking is automatic but (essentially) restricted to finite-state systems.
- Many reactive systems can be modeled by FSMs! Let’s build tools!
Late 80’s: First tools… (the first champions)

• Examples: CAESAR, COSPAN, CWB, MURPHI, SPIN, etc.
  – Differ by specification language, implementation language, comparison criterion, and/or verification algorithms,
  – but all based on systematic state-space exploration.

• NOTE: using a temporal logic is not mandatory.
  – Many “model-checking” tools do not support a full temporal logic.
  – From now on, no distinction here between model checking and systematic state-space exploration.
  – Logic is a powerful theoretical tool (characterizes classes of properties).
  – Logic can be very useful in practice too (concise and expressive).

• First success stories in analyzing circuit designs, communication protocols, distributed algorithms!
Formal Verification vs. Testing

• Experiments with these tools show that model checking can be very useful!
  – Main strength: model checking can detect subtle design errors.

• In practice, formal verification is actually testing because of approximations:
  – when modeling the system,
  – when modeling the environment,
  – when specifying properties,
  – when performing the verification.

• Therefore “bug hunting” is really the name of the game!
  – Main goal: find errors that would be hard to find otherwise.
Early 90s: First Reality Check… (the engineers)

• Model Checking is limited by the state explosion problem.

  \[ X_1 = 1 \quad X_2 = 1 \quad \ldots \quad X_n = 1 \quad 2^n \text{ states!} \]
  \[ n! \text{ interleavings!} \]

  – FSM (=state space) can itself be the product of smaller FSMs…
  – Model checking is usually linear in the size of the state space,
  – but the size of the state space is usually exponential (or worse) in the system description (program).

• State-space exploration is fundamentally hard (NP, PSPACE or worse).

• Engineering challenge: how to make model checking scalable?
Dealing with State Explosion

• Divide-and-conquer approaches:
  – abstraction: hide/approximate details.
  – compositionality: check first local properties of individual components, then combine these to prove correctness of the whole system.

• Algorithmic approaches:
  – “symbolic verification”: represent state space differently (BDDs,…).
  – state-space pruning techniques: avoid exploring parts of the state space (partial-order methods, symmetry methods,…).
  – Techniques to tackle the effects of state explosion (bit-state hashing, state-space compression, caching, etc.).
  – Etc.

• Several order of magnitudes gained! We are in business!
Late 90’s: the boom… then reality check…

• “Industrial” model-checking tools are developed and gain acceptance in industry…
  – Become routinely used for some applications in some companies.

• Mostly hardware: IBM, Intel, Lucent, Motorola, etc.

• Software designs too (with SDL (Telelogic), VFSM (Lucent),…)

• Several start-ups are trying to emerge!…
  – FormalCheck (now Cadence), Verisity, Verysys, Mentor Graphics, 0-in,…

• Making money selling model-checking tools is hard!
  – Scalability issues (state explosion…)
  – Usability issues (requires training for specification and verification)
Applications: Hardware

• **Hardware verification** is an important application of model checking and related techniques.
  
  – The finite-state assumption is not unrealistic for hardware.
  
  – The cost of errors can be enormous (e.g., Pentium bug).
  
  – The complexity of designs is increasing very rapidly (system on a chip).

• However, model checking still does not scale very well.
  
  – Many designs and implementations are too big and complex.
  
  – Hardware description languages (Verilog, VHDL,…) are very expressive.
  
  – Using model checking properly requires experienced staff.

• Quid for Software?
Applications: Software Models

- Analysis of software models: (e.g., SPIN)
  - Analysis of communication protocols, distributed algorithms.
  - Models specified in extended FSM notation.
  - Restricted to design.

- Analysis of software models that can be compiled: (e.g., SDL, VFISM)
  - Same as above except that FSM can be compiled to generate the core of the implementation.
  - More popular with software developers since reuse of “model” is possible.
  - Analysis still restricted to “FSM part” of the implementation.
Model Checking of Software

- Challenge: how to apply model checking to analyze software?
  - “Real” programming languages (e.g., C, C++, Java),
  - “Real” size (e.g., 100,000’s lines of code).

- Two main approaches for software model checking:

  Modeling languages → state-space exploration → Model checking

  Programming languages → state-space exploration → Systematic testing

  (VeriSoft)

  (Bandera, Feaver, JPF, SLAM, ...)

  abstraction
Approach: Systematic Testing (VeriSoft)

- Control and observe the execution of concurrent processes of the system under test by intercepting system calls (communication, “VS_toss(n)”, assertion violations, etc.).

- Systematically drive the system along all the paths (=scenarios) in its state space (=automatically generate, execute and evaluate many scenarios).

- From a given initial state, one can always guarantee a complete coverage of the state space up to some depth.
VeriSoft State-Space Search

• Automatically searches for: (safety properties only!)
  – deadlocks,
  – assertion violations,
  – divergences (a process does not communicate with the rest of the system during more than x seconds),
  – livelocks (a process is blocked during x successive transitions).

• A scenario (path) is reported for each error found.

• How to efficiently explore state spaces without storing any state?
  – States of arbitrary (OS) processes are too complex to be represented explicitly (no hash-tables, BDDs,…).
  – For concurrent systems, need partial-order algorithms! [Godefroid,…]
The VeriSoft Simulator
VeriSoft Project Status

• Development of research prototype started in 1996.

• Examples of applications in Lucent Technologies:
  – 4ESS Heart-Beat Monitor debugging and unit testing (Switching, switch maint.)
  – WaveStar 40G R4 integration and system testing (Optical, network management)
  – CDMA Call Processing Library testing (Wireless, call processing)

• VeriSoft 2.0 available outside Lucent since January 1999:
  – 100’s of non-commercial licenses in 25+ countries; 10’s of commercial licenses; several industrial users (Lucent, Cisco, Motorola, Philips,…)

• Examples of related research issues:
  – How to automatically close open reactive programs? [Colby, Godefroid, Jagadeesan,…]
  – How to analyze effectively partial state-spaces? [Bruns, Godefroid,…]
  – How to apply and optimize this approach to multi-threaded programs? [Stoll,...]
Approach: Automatic Abstraction

- **Main ideas and issues:**
  - 1. Abstract: extract a model out of concrete program via static analysis.
    - Which programming languages are supported? ((subset of) C, Java, Ada, DSL?)
    - Additional assumptions? (Pointers? Recursion? Concurrency?…)
    - What is the target modeling language? ((C)(E)FSMs, PDAs,…)
    - Can/must the abstraction be guided by the user? How?
  - 2. Model check the abstraction.
    - What properties can be checked? (Safety? Liveness?,…)
    - How to model the environment? (Closed or open system?…)
    - Which model-checking algorithm? (New algorithms for PDAs, HSMs,…)
    - Is the abstraction “conservative”? 
  - 3. Map abstract counter-examples back to code.
    - Behaviors violating the property may have been introduced during Step 1.
    - Hence, need to map scenarios leading to errors back to the code. HOW?
Active Area of Research...

- Examples of tools:
  - Bandera [Dwyer, Hatcliff,…]: Java to SPIN/SMV/* using user-guided abstraction mapping and slicing/abstract-interpretation/*
  - SLAM [Ball, Rajamani,…]: C to “Boolean programs” (=CFG+boolean variables); automatic abstraction refinement using predicate abstraction…
  - JavaPathFinder [Havelund, Penix, Visser,…]: Java model-checking using special JVM and model-checker…
  - Feaver [Holzmann,…]: C to SPIN using user-specified abstraction mapping…
  - Etc! (Tools for Ada, static analysis of concurrent programs,…)

- Examples of frameworks: (automatic abstraction refinement)
  - [Graf,Saidi,…], [Clarke,Grumberg,Jha,…], [Ball,Rajamani,Podelski,…], [Dill,Das,…], [Khurshan,Namjoshi,…], [Dwyer, Pasareanu, Visser,…], [Bruns,Godefroid,Huth,Jagadeesan, Schmidt,…], [Henzinger,…], etc.
Summary: Two (Complementary) Approaches

- **Systematic software testing:**
  - Idea: control the execution of concurrent processes by intercepting systems calls related to communication, and automatically drive the entire system through many scenarios.
  - Flexible and scalable approach (code independent).
  - Counterexamples arise from code execution (sound).
  - Provide complete state-space coverage up to some depth only (incomplete).

- **Static analysis for automatic model extraction:**
  - Idea: parse code to generate an abstract model which is then analyzed by model-checker; abstraction may/must be guided by the user.
  - Coverage can be exhaustive (can be complete).
  - Abstraction may cause spurious counterexamples (unsound)…
  - Technology less mature, active area of research.
Conclusions

• Model Checking is a very successful research area:
  – New: original approach to check the correctness of reactive systems.
  – Non-obvious: rich theory behind it.
  – Useful: many applications and success stories, including in industry.

• From a business perspective, the success is mixed…(!?!)

• Can model checking be applied to software?
  – Hard problem (model checking + program analysis).
  – Good for research!
  – Bad for business?