Design and Application of Formal Methods for Component Systems

František Plášil, Jiří Adámek

Institute of Computer Science
Academy of Sciences of the Czech Republic

Faculty of Mathematics and Physics
Charles University, Prague
Projects

• Realistic Application of Formal Methods in Component Systems

• Component Reliability Extensions to the Fractal Component Model
  ▪ France Telecom Research and Development
Reliability: Striking stories

• http://wwwzenger.informatik.tu-muenchen.de/persons/huckle/bugse.html
Reliability: Striking stories

- Shutdown of USS Yorktown in 1998
  - A sailor mistakenly typed 0 in a field of the kitchen inventory application.
    - Subsequent division by this field caused an arithmetic exception
    - Propagated through the system lead to power shutdown for about 3 hours
Achieving system reliability

• Experimental methods
  ▪ Testing
    • (applied to the system itself)
  ▪ Simulation
    • (experimenting with a model of a system)

• Formal methods
  ▪ Deductive verification
    • Theorem proving
  ▪ Equivalence checking
    • Comparing two specifications
  ▪ Model checking
    • Checking a particular property
  ▪ Static (code) analysis
Formal Methods

• Full verification
  ▪ Prove that the behavior of the system is equivalent to a given behavior
    • Equivalence checking

• Partial verification
  ▪ Prove that the behavior of the system satisfies certain conditions
    • Model checking, theorem proving, static analysis
Equivalence Checking

- Check whether two behavior specifications are equivalent
  - Consider a hierarchical system:
Equivalence Checking

- Check whether two behavior specifications are equivalent
  - Behavior specified as a labeled transition system (LTS)
Equivalence Checking: Characteristics

- Exhaustive
  - All errors are found and no false negatives are generated
- Suitable for hierarchical structures of behavior descriptions
  - Hardware
  - Hierarchical (software) component systems
  - Model driven development (OMG MDA)

- Even checking of an infinite state transition system against finite state one is **decidable**
  - Finite state specification vs. infinite state implementation (e.g. code)

- Not suitable for “flat” (non-component) software architectures
- State explosion problem
  - The size of the model grows exponentially with the number of components at a level
Model Checking

- Check whether a behavior specification satisfies a given temporal property
  - Behavior specified as a Kripke structure
  - Temporal property specified by a formula in a temporal logic or a finite automaton

Property to be checked:

\[(EG E[p U q]) \& EX r\]
Model Checking: Characteristics

- Exhaustive
  - All errors are found (and no false negatives are generated)
- Appropriate for finite-state systems
  - Hardware
  - Communication protocols
  - ... 
  - Algorithms used in OS kernels (e.g. schedulers, cache managers)

- Undecidable for general infinite state Kripke structures
  - Generally a program code ~ infinite state Kripke structures
    - Heap (assumed unlimited), ...

- State explosion problem
  - The size of the model grows exponentially with the number of components at a level
Theorem Proving

- Check whether a system (e.g. program) satisfies a given property
  - The property specified as a formula in a predicate logic
  - The behavior specified in a functional programming language
    or as a logic formula

\[ \forall x: [a]( \text{length}(x) = \text{length}(\text{reverse}(x)) ) \]

- \text{length} [[]] = 0
- \text{length} x::xs = 1 + \text{length}(xs)
- \text{reverse} [[]] = []
- \text{reverse} x::xs = \text{conc}(\text{reverse}(xs), [x])

...
Theorem Proving: Characteristics

• Exhaustive
  ▪ All errors are found and no false negatives are generated

• It is undecidable in general
  ▪ Human assistance needed to finish a proof

• Not very suitable for proving properties of imperative programs
  ▪ Hard to capture global state and side effects
Theorem Proving: Application

• Proof-carrying code
  ▪ For a given program, a formula describing its correctness is generated
    • with respect to a given set of security policies
  ▪ The formula is proven
    • either automatically or with human assistance
  ▪ The proof is distributed along with the code
    • as a security certificate
  ▪ The receiver of the code
    • checks the proof
Static Analysis

• Check whether a given program satisfies a set of predefined constraints
  ▪ Program in an imperative programming language (C/C++, Java, …)
  ▪ Constraints, such as: no
    • Non-initialized variables used
    • Out-of-bound index used to access an array
    • Null-pointer is dereferenced
    • ….
Static Analysis: Characteristics

• The problem is undecidable in general
• Most of the methods check the properties locally
  ▪ Only the context of one function / method is considered
  ▪ Only one thread is considered
  ➔ Non-exhaustive
    • Not all property violations are found
    • False negatives may be reported
Reality in SW: Combined Methods

• Automated Code Verification
  ▪ Model Checking + Theorem Proving + Static Analysis
    • Java PathFinder, NASA
    • SLAM/SDV Microsoft (driver checking)
    • Bandera
Reality in SW: Combined Methods

- Behavior protocols
  - Plasil, Adamek, Visnovsky, Kofron,…
    - Equivalence Checking + Model Checking
    - Applied to SW components

- Our desire (project “Information Society”)
  - Behavior protocols + Automated Code Verification of SW components
frame protocol:

!da.Open;

( ?d.Insert { !tr.Begin;
    !da.Insert; !lg.LogEvent;
    (!tr.Commit + tr.Abort) }
  +
  ?d.Delete { !tr.Begin;
    !da.Delete; !lg.LogEvent;
    (tr.Commit + tr.Abort) }
  +
  ?d.Query { !da.Query }
  ) * ;

!da.Close
• **Goal**
  - To check whether code of a component is compliant with its behavior specification
    - behavior protocol

• **Benefits of component granularity**
  - Analyzing code of each component separately reduces state explosion
Project: Component Reliability Extensions to the Fractal Component Model

• Partner
  ▪ France Telecom Research and Development

• Goals
  ▪ To extend the Fractal Component Model with Behavior Protocols
    • Behavior protocols were originally developed for the SOFA component model
  ▪ To implement static and dynamic checking of behavior compliance
    • for Fractal components
Project: Realistic Application of Formal Methods in Component Systems

• Partners
  ▪ Charles University, Faculty of Mathematics and Physics
  ▪ Masaryk University in Brno, Faculty of Informatics

• Goals
  ▪ Platform supporting automated formal verification of component-based applications
    • To develop/enhance methods of automated program verification
    • To identify approaches making the existing verification tools more efficient, especially in a distributed environment