Model Checking of Programs: Software Component Perspective

Jiří Adámek, František Plášil

DISTRIBUTED SYSTEMS RESEARCH GROUP
http://nenya.ms.mff.cuni.cz

CHARLES UNIVERSITY PRAGUE
Faculty of Mathematics and Physics
Outline

• “Classic” Model checking
• Model checking of programs
  - Problems and approaches
• Project: Component Reliability Extensions for Fractal Component Model
  - Supported by France Telecom
  - The role of the model checking of programs in the project
“Classic” Model Checking

Behavior Specification ~ Model → Model Checker → Properties

YES/NO + Counter-example
“Classic” Model Checking

• An automatic method
• Counterexample: helps to identify the problem
• Successfully used for verification of:
  - Hardware
  - Communication protocols
  - Distributed algorithms
  - Algorithms used in OS kernels
    • Schedulers, synchronization algorithms
  - Cryptographic protocols
“Classic” Model Checking

• Behavior specification ~ Model
  ▪ Modeling languages
    • Petri nets
    • Process algebras
    • C-like languages (e.g. Promela / SPIN)
    • Explicit definition of variable value changes (e.g. SMV)
  ▪ Common mathematical abstraction
    • Kripke structure
    • Labeled transition system
    • State transition system
  ▪ Finite state models

• Property specification
  ▪ Temporal logic
    • State based
      ▪ LTL, CTL, μ-calculus, …
    • Transition based
      ▪ Hennessy-Milner logic, Action-based EF logic, …
  ▪ Finite automata
State transition system:

Result: NO
Counter-example: infinite-loop ( receive process wait )

Property (CTL):
AG (received => AF sent)
“Classic” Model Checking

• Fighting the state-explosion problem
  ▪ Binary Decision Diagrams
  ▪ Model Checking Using Representatives
    • Also called Partial Order Reduction
  ▪ Counter-example guided abstraction refinement
    • Safety properties only
Counter-example guided abstraction refinement

Create an initial abstract model

Refine the abstract model

Abstract Model

Is it false-negative?

Model Checker

Result: NO + CE

Result: YES

Model

Properties

NO

YES
Model Checking of Programs

• Examples of the properties to check:
  ▪ Are all variables initialized before the first usage?
  ▪ Is there a possibility of a null-pointer dereferencing?
  ▪ Is there a possibility of a deadlock?

• All those properties are safety properties
How to check a property of a program? (1)

“Classic” approach:

1. Program
2. Write a model (manually)
3. Model
4. Properties
5. “Classic” Model Checker
6. YES/NO + Counter-example
How to check property of a program? (2)

Direct model checking of the program:

- Program
- Program Model Checker
- Properties
- YES/NO + Counter-example
Direct model checking of programs

• Advantages
  ▪ No need for manual behavior specification
  ▪ Programming languages are more powerful than behavior specification languages
    • More complex behavior can be modeled

• Disadvantages
  ▪ Programming languages are more powerful than behavior specification languages
    • Model checking of programs is undecidable in general
      ➔ For complex programs a model checker may either stop without any answer or run forever
Direct model checking of programs

• Tools
  - SLAM / SDV (Microsoft Research)
  - Bogor (Kansas State University)
  - Java PathFinder (NASA)
The SLAM Model Checker

• All properties we want to verify are safety properties
  ⇒ Counter-example guided abstraction refinement can be used

• A problem: how to implement
  ▪ Counter-example analysis
  ▪ Abstraction refinement
  ⇒ Theorem proving
The SLAM Model Checker

Create an initial abstract model

Refine the abstract model

Abstract Model

Properties

Model

Is it false-negative?

Model Checker

Result: NO + Counter-example

Result: YES

YES

NO

NO

+ CE
Our project

• Component Reliability Extensions for Fractal Component Model
  ▪ A joint project of the Institute of Computer Science and France Telecom

• Context
  ▪ Hierarchical component models – SOFA, Fractal
  ▪ Behavior protocols

• Goal
  ▪ To develop tools for verification of Fractal components
    • including model checking of the component implementation
Our project

?IDhcpServerLifetimeController.Start ;
(
  !IDhcpCallback.IpAddressInvalidated* | !IDhcpCallback.IpAddressInvalidated* |
  (?IManagement.UsePermanentIpDatabase^ ;
   ( !IIpMacPermanentDb.GetIpAddress* |
     ( !IManagement.UsePermanentIpDatabase$ ;
       ?IManagement.StopUsingPermanentIpDatabase^  
     )
   )
  )
)*
)
Our project

• Verification tasks
  ▪ Compliance checking for composite components
    • Is the aggregate behavior of a composite component compliant with the specified behavior?
  ▪ Detection of composition errors
    • Do the components on a single level of nesting communicate in a correct way?

► Both performed by the BP-Checker previously developed within the SOFA project
► Within this project we adapted the BP-Checker for the Fractal component model
Our project

• Verification tasks
  ▪ Compliance checking for primitive components
    • Is the implementation (code) of a primitive component compliant with the specified behavior?
    • The task can be seen as a special kind of automata-based model checking of a program
      ▪ A behavior protocol naturally specifies a finite state machine

➤ A new tool was developed for the task
  • It is based on a combination of the Java PathFinder and the BP-Checker
Our project

• Case study
  - An application providing wireless internet access on airports
    - Network management
    - Payment by credit cards
    - Discounts for first class/business class passengers
    - Discounts for frequent-flyer card owners
      - Airlines alliances also considered
  - Complexity of the architecture
    - ~ 20 components
    - 3 levels of nesting
  - Complexity of behavior protocols
    - ~ 60 events in a component
    - ~ 500 lines of spec all together
    - Typical architecture protocol state space $10^7 – 10^8$
  - Checking time
    - Composition errors and compliance of composite components: 50 minutes
    - Compliance for primitive components: 1 hour