Module-based Software Security

Eun-Young Lee

Dongduk Women's University

3rd ROSAEC Workshop

We're doing well….

- Analyze programs statically, line by line
- Find out errors of bad programs (innocent or evil)

Software systems are getting bigger...

- Black box approach
- Object-oriented programming
- Component-based software architecture
  - Off-the-self components
- Service-oriented software architecture

- Programmers cannot help using software components, which are not written by themselves.
- Concerns
  - How can you trust the correctness of foreign components?
  - Are you sure that you are using foreign components correctly?

Component Shopping

- Concerns
  - How can you trust the correctness of foreign components?
  - Are you sure that you are using foreign components correctly?

- Useful shopping guidelines
  - Type Checking
  - Interface Checking

Research Motivation

- There should be ways
  - to specify the pre-/or post-conditions of software components
  - to verify that a software component is correctly used
- Component Constraint Specification
  - OCL (Object Constraint Language)
  - Behavioral Subtyping
- Component Constraint Verification
  - JML/ESC

Component Constraint Specification

OCL

Object Constraint Language (OCL)

- Object Constraint Language (OCL) is part of UML
- OCL was developed at IBM by Jos Warmer as a language for business modeling within IBM
- OCL specification is available here:

  http://www.omg.org/technology/documents/formal/ocl.htm
- Recommended text:
  - "The Objection Constraint Language: Precise Modeling with UML", by Jos Warmer and Anneke Kleppe

OCL Constraints

- OCL constraints are declarative
  - They specify what must be true, not what must be done
- OCL constraints have no side effects
  - Evaluating an OCL expression does not change the state of the system
- OCL constraints have formal syntax and semantics
  - their interpretation is unambiguous

Types and Instances

- OCL types are divided into following groups
  - Pre-defined types
    - Basic types: String, Integer, Real, Boolean
    - Collection types: Collection, Set, Bag, Sequence
  - User-defined model types
    - User defined classes such as Customer, Date, LoyaltyProgram

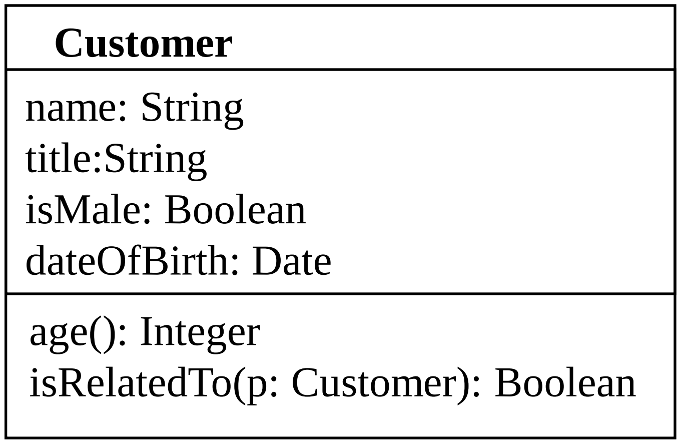
OCL expressions and constraints

- Each OCL expression has a result
  - the value that results by evaluating the expression
- The type of an OCL expression is the type of the result value
  - either a predefined type or a model type
- An OCL constraint is an OCL expression of type Boolean

OCL Constraints

- OCL supports Design-by-Contract style constraints
  - Invariants
  - Pre-conditions
  - Post-conditions

- Example of OCL Constraints



```
Customer
age() >= 18

Customer
self.isRelatedTo(self) = true

Customer
self.name='Joe Senior' implies
                   self.age() > 21
```

OCL Tools

- Octopus: *OCL Tool* for Precise Uml Specifications
  - Plug-in for Eclipse
  - an IDE for *OCL* (Object Constraint Language) and UML.

- OSLO Project - Open Source Library for *OCL*

- MDT-OCLTools
  - Plug-in for Eclipse
  - Model Development Tools (MDT) Project
  - first-class support to modelers working with specifications containing expressions written in OCL

Component Constraint Verification

JML AND ESC/JAVA

Java Modeling Language (JML)

- JML is a behavioral interface specification language
- The Application Programming Interface (API) in a typical programming language (for example consider the API of a set of Java classes) provides very little information
  - The method names and return types, argument names and types
- This type of API information is not sufficient for figuring out what a component does
- JML is a specification language that allows specification of the behavior of an API
  - not just its syntax, but its semantics
- JML specifications are written as annotations
  - As far as Java compiler is concerned they are comments but a JML compiler can interpret them

JML Project(s) and Materials

- Information about JML and JML based projects are available at Gary Leavens' website:
  - http://www.cs.ucf.edu/~leavens/JML/

- Recommended Reference:
  - Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212-232, June 2005

JML

- One goal of JML is to make it easily understandable and usable by Java programmers, so it stays close to the Java syntax and semantics whenever possible

- JML supports design by contract style specifications with
  - Pre-conditions
  - Post-conditions
  - Class invariants

- JML supports quantification (`\forall`, `\exists`), and specification-only fields and methods
  - Due to these features JML specifications are more expressive than Eiffel contracts and can be made more precise and complete compared to Eiffel contracts

JML Annotations

- JML assertions are added as comments to the Java source code
  - either between `/*@ . . . @*/`
  - or after `//@`
    - These are ***annotations*** and they are ignored by the Java compiler

- In JML properties are specified as Java boolean expressions
  - JML provides operators to support design by contract style specifications such as `\old` and `\result`
  - JML also provides quantification operators (`\forall`, `\exists`)

- JML also has additional keywords such as
  - `requires, ensures, signals, assignable, pure, invariant, non null, . . .`

Design by Contract in JML

- In JML, contracts:
  - Pre-conditions are written as a `requires` clauses
  - Post-conditions are written as `ensures` clauses
  - Invariants are written as `invariant` clauses

JML Tools

- There are tools for parsing and type-checking Java programs and their JML annotations
  - JML compiler (**`jmlc`**)

- There are tools for supporting documentation with JML
  - HTML generator (**jmldoc**)

- There are tools for runtime assertion checking:
  - Test for violations of assertions (pre-, post-conditions, invariants) during execution
  - Tool: **jmlrac**

- There are testing tools based on JML
  - JML/JUnit unit test tool: **jmlunit**

- Extended static checking:
  - Automatically prove that contracts are never violated at any execution
  - Automatic verification is done statically (i.e., at compile time).
  - Tool: **ESC/Java**

Extended Static Checking

- Extended Static Checking (ESC) is a static analysis technique that relies on automated theorem proving

- Goals of Extended Static Checking:
  - prevent common errors in programs
  - make it easier to write dependable programs
  - increase programmer productivity

Extended Static Checking

- Help programmer in writing dependable programs by generating warning messages pointing out potential errors at compile time

- The programmer annotates the code using design-by-contract style assertions written in Java Modeling Language (JML)
  - Assertions can be added to clarify the contract of a method based on the warnings generated by the ESC tool

- Big difference between ESC and Dynamic design-by-contract monitoring tools:
  - ESC checks errors at compile time, before running the program, for ***all*** possible executions of the program!

Extended Static Checking

- ESC/Java also restricts the types of properties that it checks

- Verification of general properties of programs may require a lot of input from the programmer
  - Programmer may need to enter a lot of assertions (such as loop invariants, etc.) to remove the false positives.

- ESC/Java tries to achieve a balance between the properties it covers and the effort required by the programmer
  - In this sense it is different than the more ambitious program verification tools based on Hoare Logic and automated theorem provers

Types of Errors

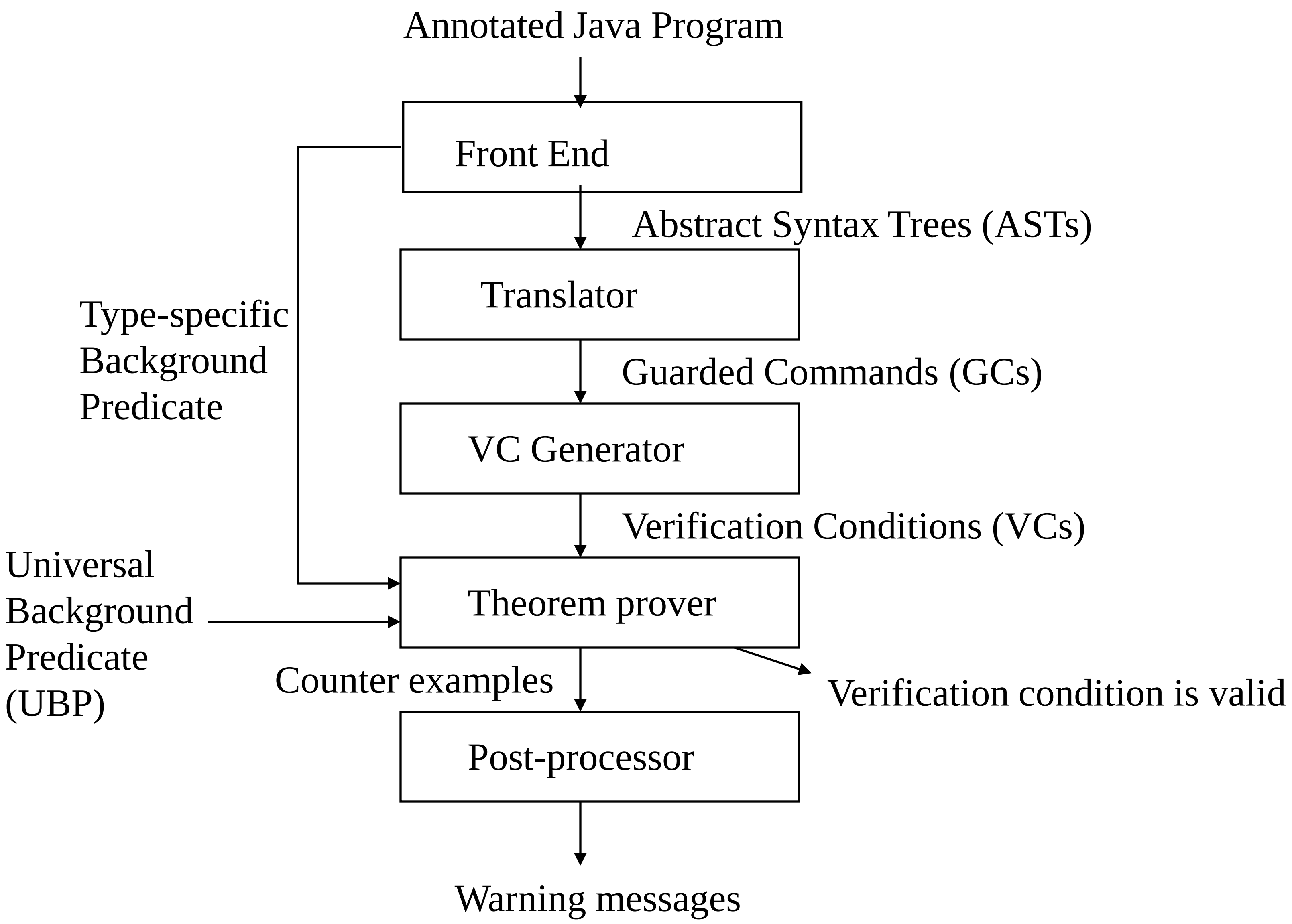
- ESC/Java checks three types of errors

  1. Common programming errors such as null dereferences, array index bound errors, type-cast errors, division by zero, etc.

  2. Common synchronization errors such as race conditions and deadlocks

  3. **Violations of program annotations, i.e., static checking of contracts such as pre-conditions, post-conditions or invariants**

How does ESC/Java Work?


Annotated Java Program
Front End
Abstract Syntax Trees (ASTs)
Translator
Guarded Commands (GCs)
VC Generator
Verification Conditions (VCs)
Type-specific Background Predicate
Universal Background Predicate (UBP)
Theorem prover
Counter examples
Verification condition is valid
Post-processor
Warning messages

Extended Static Checking

- Given a program, ESC tool generates a logical formula, called a ***verification condition***, that is valid if and only if the program is free of the classes of errors under consideration

- An automated theorem prover is used to check if the negation of the verification condition is satisfiable
  - Any satisfying assignment to the negation of the verification condition is a ***counter-example*** behavior that demonstrates a bug

Modular Verification

- ESC/Java uses a modular verification strategy.
- ESC/Java uses Design-by-contract style specifications to achieve this.
- When ESC/Java produces the guarded command for a method *M*, it translates each method call in *M* according to the specification (of its contract) rather than the implementation of the called method.
- Hence, the resulting non-deterministic guarded command *G* may be able to go wrong in ways involving behaviors of called routines that are permitted by their specification but can never occur with the actual implementations.
- Note that this is a sign of incompleteness, i.e., ESC/Java can generate false positives.

Modular Verification

- Modular verification modularizes and hopefully simplifies the verification task
  - specifications are likely to be simpler than the implementations

- Another nice side effect of the modular verification is that the methods are verified against the specifications of the methods that they are using
  - In the future if the implementation of a method that is called by method M changes but if its specification remains the same we do not have to verify M again since the verification is based on the specification not the implementation of the called methods

Verification Condition Generator

- Verification Condition (VC) Generator generates verification conditions from the guarded commands

- The output of the VC generator for a guarded command *G* is a predicate in first-order logic that holds for precisely those programs states from which no execution of the command *G* can go wrong

- VC generator in ESC/Java is an efficient weakest-precondition generator

Extended Static Checking References

- Suggested reading:
  - ``Extended static checking for Java." Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. In Proc. of PLDI 2002.
  - ``An overview of JML tools and applications." Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. In International Journal on Software Tools for Technology Transfer, 7(3):212-232, June 2005.
- Other References
  - ``Applications of extended static checking." K.R.M. Leino (In proceedings of SAS'01, ed. Patrick Cousot, volume 2126 of Springer LNCS, 2001.)
  - ``Extended Static Checking: a Ten-Year Perspective." K.R.M. Leino (In proceedings of the Schloss Dagstuhl tenth-anniversary conference, ed. Reinhard Wilhelm, volume 2000 of Springer LNCS, 2001.)
  - ``ESC/Java User's Manual.'' K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Technical Note 2000-002, Compaq Systems Research Center, October 2000.

Research Challenges

- How to specify the constraints (i.e., verification conditions)?
  - Various languages are suggested
  - Automatic constraint generation would be beneficiary
- How to verify the constraints at component composition time?
  - Smart programmers can write all the proofs , stating that their programs satisfy the constraints.
  - For lazy programmers, a smart compiler can prove (or help to prove) that a user program satisfies the constraints.

Research Challenges

- What to be verified at component composition time?
  - Security Policies
    - This program never accesses the un-authorized area of memory.
    - This program never exposes the private information to outside.
  - Smart programmers can state a security policy in the form of component constraints (e.g., in OCL, in JML)
  - Some smart compilers generate constraints (or verification conditions) for lazy programmers
    - ESC/Java
  - For a given security policy, constraints (pre-condition and post-condition) for a component may differ from one host program to another host program.