Theme 5. Support technologies

Programming Languages Technologies and Paradigms (LTP)

DSIC, ETSInf

Main goals

- Basic knowledge about semantics-based techniques for program analysis, testing and verification
- Understanding the impact of support technologies in *software* development
- Identifying the basic ingredients in the automation of *software* processes

Developing good *software*

- Using tools and technologies for automatic support of some processes is **essential** to guarantee the quality of developed software
- Two main strategies can be considered:
  - *Prevention*: invest in a good design; use analysis, testing, and debugging techniques from the beginning, etc.
  - *Correction*: errors and malfunctions are detected and then corrected by using formal methods and analysis, verification, testing, and debugging techniques.

Developing good *software*

- Using tools and technologies for automatic support of some processes is **essential** to guarantee the quality of developed software
- Two main strategies can be considered:
  - *Prevention*: invest in a good design; use analysis, testing, and debugging techniques from the beginning, etc.
  - *Correction*: errors and malfunctions are detected and then corrected by using formal methods and analysis, verification, testing, and debugging techniques.

Learn more...

ISW (3A) and/or the Software Engineering branch (MFI, AVD, . . .)

Support technologies

Support tools and technologies for software development:

- Rely on the **semantics** of the programming languages
- Can be *static* (compilation time) or *dynamic*
- Can be formal methods or not
- Can be automatic or semiautomatic

Support technologies

Applications

- Check the system for correctness (it fulfils user's requirements),
- Guarantee the absence of unexpected runtime errors
- Analyze system's efficiency (execution time, resource consumption, etc.),
- Automatic code optimization,
- ...

Support technologies

Examples (*Software testing*)

Software testing

- Main goal: detect possible bugs in code
  - The absence of failures is not guaranteed
- The program is executed on a test set
  - It is not exhaustive
- The challenge: test set design
  - there are methods for the automatic generation of test sets, for instance by using symbolic execution

Support technologies

Examples (*Software testing*)

Software testing step-by-step

- Test set design
- Execution of the program on the test sets
- Evaluating the result
  - If an error is found: correct and test again
  - No error detected: more tests required?
- Managing test sets

- Test set design is the costly step

Support technologies

Examples (*Static analysis*)

Static program analysis

Techniques to predict *in compilation-time* the dynamic (i.e., runtime) behavior of the program.

- Undecidable problem: absolute precision is incompatible with effectiveness.
- Semantics-based

Used in

- Compiler optimization
- Verification
- Integrated Development Environments (IDEs)

Support technologies

Examples (*Static analysis*)

Static program analysis

- Example: constant propagation
- Goal: code optimization

```
y:=4;
x:=1;
while (y>x) do
         (z := y;
          x := y*y);
x:=z
```

Analysis:

For each instruction: have the variables used in this program point a constant value?

Support technologies

Examples (*Static analysis*)

Static program analysis

- Example: constant propagation
- Goal: code optimization

```
y:=4;
x:=1;
while (y>x) do
         (z := y;
          x := y*y);
x:=z
```

- variable `y` is constant in the loop
- variable `x` is NOT constant in the loop

Support technologies

Examples (*Static analysis*)

Static program analysis

- Example: constant propagation
- Goal: code optimization

```
y:=4;
x:=1;
while (4>x) do
        (z := 4;
         x := 4*4);
x:=z
```

- optimize the code to avoid memory accesses and wasteful computations

Support technologies

Examples (*Debugging*)

Debugging

- Main goal: bug detection
- Main approaches:
  - handcrafted: *tracers*, *print debugging*
  - directed: algorithmic debugging
  - automatic: assertions, *abstract debugging*

- The usual approach is handcrafted or semi-automatic debugging

Symbolic execution

Many support tools rely on *symbolic execution*, which is based on the ***small-step*** semantics of the language.

Idea

- Symbolic values are used as inputs so that

··· a tree is built to represent all possible program executions,

··· each execution *path* is given appropriate *conditions* to be fulfilled by the inputs in order to follow such a path

Symbolic *vs* concrete execution

Example

- Let $X1$ and $X2$ be input parameters for program

$$
\begin{array}{l}
\textbf{if } X1 > X2 \ \textbf{ then } X1 := X1 - X2 \\
\phantom{\textbf{if } X1 > X2 \ } \textbf{ else } X2 := X2 - X1
\end{array}
$$

- Concrete execution. Possible traces:

$$\langle \textbf{if } \ldots, \{X1 \mapsto 3, X2 \mapsto 6\}\rangle \to \ldots \to \langle \checkmark, \{X1 \mapsto 3, X2 \mapsto 3\}\rangle$$
$$\langle \textbf{if } \ldots, \{X1 \mapsto 3, X2 \mapsto 8\}\rangle \to \ldots \to \langle \checkmark, \{X1 \mapsto 3, X2 \mapsto 5\}\rangle$$
$$\langle \textbf{if } \ldots, \{X1 \mapsto 4, X2 \mapsto 2\}\rangle \to \ldots \to \langle \checkmark, \{X1 \mapsto 2, X2 \mapsto 2\}\rangle$$
$$\vdots$$

Symbolic *vs* concrete execution

Example

- Let $X1$ and $X2$ be input parameters for program

$$\textbf{if } X1 > X2 \textbf{ then } X1 := X1 - X2$$
$$\textbf{else } X2 := X2 - X1$$

- Symbolic execution: Execution tree:

Symbolic values

path condition

$$\langle \textbf{if } X1{>}X2 \ \ldots, \{X1 \mapsto ?X1, X2 \mapsto ?X2\}, \textit{true}\rangle$$
$$\downarrow$$
$$\langle X1 := X1 - X2, \{X1 \mapsto ?X1, X2 \mapsto ?X2\}, ?X1 > ?X2\rangle$$
$$\downarrow$$
$$\langle \checkmark, \{X1 \mapsto ?X1 - ?X2, X2 \mapsto ?X2\}, ?X1 > ?X2\rangle$$

Symbolic *vs* concrete execution

Example

- Let $X1$ and $X2$ be input parameters for program

$$\begin{array}{l} \textbf{if } X1 > X2 \ \textbf{ then } X1 := X1 - X2 \\ \qquad\qquad\qquad \textbf{else } X2 := X2 - X1 \end{array}$$

- Symbolic execution. Execution tree:

$\langle \textbf{if } X1 {>} X2 \ \ldots, \{X1 \mapsto ?X1, X2 \mapsto ?X2\}, \mathit{true} \rangle$

$\langle X1 := X1 - X2, \{X1 \mapsto ?X1, X2 \mapsto ?X2\}, ?X1 > ?X2 \rangle$

$\langle \checkmark, \{X1 \mapsto ?X1 - ?X2, X2 \mapsto ?X2\}, ?X1 > ?X2 \rangle$

$\langle X2 := X2 - X1, \{X1 \mapsto ?X1, X2 \mapsto ?X2\}, ?X1 \leq ?X2 \rangle$

$\langle \checkmark, \{X1 \mapsto ?X1, X2 \mapsto ?X2 - ?X1\}, ?X1 \leq ?X2 \rangle$

Symbolic *vs* concrete execution

Example

- Let $X1$ and $X2$ be input parameters for program

$$\begin{array}{ll}\textbf{if } X1 > X2 & \textbf{then } X1 := X1 - X2 \\ & \textbf{else } X2 := X2 - X1\end{array}$$

- Symbolic execution. Execution tree:

$$\langle \textbf{if } X1>X2 \ \ldots, \{X1 \mapsto ?X1, X2 \mapsto ?X2\}, \mathit{true}\rangle$$

$$\langle X2 := X2 - X1, \{X1 \mapsto ?X1, X2 \mapsto ?X2\}, ?X1 \leq ?X2\rangle$$

$$\langle X1 := X1 - X2, \{X1 \mapsto ?X1, X2 \mapsto ?X2\}, ?X1 > ?X2\rangle$$

$$\langle \checkmark, \{X1 \mapsto ?X1, X2 \mapsto ?X2 - ?X1\}, ?X1 \leq ?X2\rangle$$

$$\langle \checkmark, \{X1 \mapsto ?X1 - ?X2, X2 \mapsto ?X2\}, ?X1 > ?X2\rangle$$

final *path conditions*

Symbolic execution for SIMP

- A configuration of the concrete machine is a pair:

$$\langle instr, s \rangle$$

  - a state s is a mapping from variables to values, e.g., $\{X \mapsto 0, Y \mapsto 5\}$.

- A configuration of the symbolic machine is a triple:

$$\langle instr, ss, pc \rangle$$

  - a symbolic state ss is a mapping from variables to *symbolic expressions*, e.g., $\{X \mapsto ?X, Y \mapsto ?X + ?Y\}$.
  - A *path condition pc* is a boolean condition on the *initial symbolic values* of the input parameters.

Symbolic operational semantics for SIMP

- Similar to the *small-step* semantics for SIMP (Theme 2)
- Initial state: a **symbolic value** (beginning with '?') is given to each input variable
- When reaching a forking point (e.g., a conditional or loop), the guard is conjoined to the corresponding *path condition*

Symbolic operational semantics

Evaluation of symbolic expressions

- Evaluation of expressions:

  $\langle a, ss \rangle \Rightarrow$ *sexp* represents the symbolic evaluation of arithmetic expression *a*.

  The evaluation of subexpressions in *a* is attempted and their (symbolic) values are then used to obtain the symbolic value of *a*.

  Any concrete value is used to further simplify the expression.

Examples:

- $\langle X + Y, \{X \mapsto ?X, Y \mapsto 3\}\rangle \Rightarrow\ ?X + 3$
- $\langle X + Y, \{X \mapsto 5, Y \mapsto 3\}\rangle \Rightarrow\ 8$
- $\langle (X * X) + (Y * Y), \{X \mapsto ?X, Y \mapsto 3\}\rangle \Rightarrow\ (?X * ?X) + 9$

Symbolic operational semantics

Execution rules

- Sequence:

$$\frac{}{\langle \checkmark\, ; i_1, ss, p\rangle \;\rightarrow\; \langle i_1, ss, p\rangle}$$

$$\frac{\langle i_0, ss, p\rangle \;\rightarrow\; \langle i_0', ss', p'\rangle}{\langle i_0; i_1, ss, p\rangle \;\rightarrow\; \langle i_0'; i_1, ss', p'\rangle}$$

- Assignment:

$$\frac{\langle a, ss\rangle \;\Rightarrow\; sexp}{\langle X := a, ss, p\rangle \rightarrow \langle \checkmark, ss[X \mapsto sexp], p\rangle}$$

In general, the computed result in *sexp* is a symbolic expression rather than a value.

Symbolic operational semantics

Execution rules

- Conditional: A fork is introduced in the tree by using the following two rules:

$$\frac{\langle b, ss \rangle \Rightarrow sb}{\langle \textbf{if } b \textbf{ then } i_0 \textbf{ else } i_1, ss, p \rangle \rightarrow \langle i_0, ss, p \wedge sb \rangle}$$

$$\frac{\langle b, ss \rangle \Rightarrow sb}{\langle \textbf{if } b \textbf{ then } i_0 \textbf{ else } i_1, ss, p \rangle \rightarrow \langle i_1, ss, p \wedge \neg sb \rangle}$$

Symbolic operational semantics

Execution rules

- Conditional: A fork is introduced in the tree by using the following two rules:

$$\frac{\langle b, ss\rangle \Rightarrow sb}{\langle \textbf{if } b \textbf{ then } i_0 \textbf{ else } i_1, ss, p\rangle \rightarrow \langle i_0, ss, p \wedge sb\rangle}$$

$$\frac{\langle b, ss\rangle \Rightarrow sb}{\langle \textbf{if } b \textbf{ then } i_0 \textbf{ else } i_1, ss, p\rangle \rightarrow \langle i_1, ss, p \wedge \neg sb\rangle}$$

- We can use a *logical engine* to check $p \wedge sb$ (y $p \wedge \neg sb$) for satisfiability. In this way, we can *prune* unfeasible executions.

Symbolic operational semantics

Execution rules

- Loop: As for the conditional, the following rules are used to fork the tree.

$$\frac{\langle b, ss\rangle \Rightarrow \; sb}{\langle \textbf{while } b \textbf{ do } i, ss, p\rangle \rightarrow \; \langle \checkmark, ss, p \wedge \neg sb\rangle}$$

$$\frac{\langle b, ss\rangle \Rightarrow \; sb}{\langle \textbf{while } b \textbf{ do } i, ss, p\rangle \rightarrow \; \langle i; \textbf{while } b \textbf{ do } i, ss, p \wedge sb\rangle}$$

Symbolic operational semantics

Execution rules

- Loop: As for the conditional, the following rules are used to fork the tree.

$$\frac{\langle b, ss\rangle \Rightarrow sb}{\langle \textbf{while } b \textbf{ do } i, ss, p\rangle \rightarrow \langle \checkmark, ss, p \wedge \neg sb\rangle}$$

$$\frac{\langle b, ss\rangle \Rightarrow sb}{\langle \textbf{while } b \textbf{ do } i, ss, p\rangle \rightarrow \langle i; \textbf{while } b \textbf{ do } i, ss, p \wedge sb\rangle}$$

Warning!

The loop can make the symbolic execution tree infinite due to the possibility of infinitely many executions.

Exercise

Write the symbolic execution tree for this program:

$$\textbf{if } X > 3 \textbf{ then } (Y := 2; X := X - 2) \textbf{ else } Y = 6$$

Symbolic execution

Application: test set generation

- Symbolic execution can be used to automatically generate test sets for program testing

Idea

1. If the tree is infinite, we bound it. Two main approaches:
   - Put limits to the depth of the tree, or
   - Put limits to the iterations of the loop
2. The *path conditions* are obtained from the leaves of the tree; we call them **final** *path conditions*
3. Use a logical engine to define the *test sets* as *values for the variables that satisfy each path condition*

Symbolic execution

Application: test set generation

$$\langle \textbf{if } X1 > X2\ \ldots, \{X1 \mapsto ?X1, X2 \mapsto ?X2\}, \mathit{true}\rangle$$

$$\langle X2 := X2 - X1, \{X1 \mapsto ?X1, X2 \mapsto ?X2\}, ?X1 \leq ?X2\rangle$$

$$\langle X1 := X1 - X2, \{X1 \mapsto ?X1, X2 \mapsto ?X2\}, ?X1 > ?X2\rangle$$

$$\langle \checkmark, \{X1 \mapsto ?X1, X2 \mapsto ?X2 - ?X1\}, ?X1 \leq ?X2\rangle$$

$$\langle \checkmark, \{X1 \mapsto ?X1 - ?X2, X2 \mapsto ?X2\}, ?X1 > ?X2\rangle$$

- If values for $?X1$ and $?X2$ satisfying the *final path conditions* are found, we can use them to define test sets for the corresponding paths.

  Case 1: $?X1 = 2, ?X2 = 1$

  Case 2: $?X1 = 2, ?X2 = 2$

Symbolic execution

Application: test set generation

Automation is important:

- We save development time
- Defining the test sets can be costly
- This is the reason why there are tools to automatically generate test sets using symbolic execution

Example

Define test sets to cover all (feasible) execution paths

$$\begin{array}{l}\textbf{if } (X1 >= X2)\&\&(X3 < X2 + X1) \\ \quad \textbf{then if } (X1 < 256) \;\; \textbf{then } X1 := X2/(X1 - X2) \\ \quad \phantom{\textbf{then if } (X1 < 256) \;\;} \textbf{else } X1 := 7 \\ \quad \textbf{else } X1 := 0\end{array}$$

Testing support tools

Based on symbolic execution

- Klee: a project of the University of Illinois at Urbana-Champaign (UIUC)
  - test set generation
  - for C
  - `https://klee.github.io/`
  - installed in all computers at DSIC labs (linux version)
  - Demo available here:

    `https://poliformat.upv.es/access/content/group/GRA_11557_2015/TEORIA/Tema%205/media/klee-small.mp4`
- Java Pathfinder: a NASA project with many functionalities, in particular:
  - test set generation (JUnit format)
  - for Java
  - `http://babelfish.arc.nasa.gov/trac/jpf`
  - Demo available here:

    `https://poliformat.upv.es/access/content/group/GRA_11557_2015/TEORIA/Tema%205/media/jpf-small.mp4`

Other tools

Based on symbolic execution

- KeY: Analysis tool for Java
  - JML annotations are used
  - Deduction based on symbolic execution
  - Test set generation
  - Debugger based on symbolic execution
  - `http://www.key-project.org/`

Testing support tools

Relying on other technologies

- PEX: Microsoft's tool
  - Automatic test set generation
  - For .NET
  - `http://research.microsoft.com/en-us/projects/pex/`
- QuickCheck: Developed in Chalmers University, then exported to other languages and business models
  - Random generation based on property specifications
  - for Haskell, then adapted to other languages like C, Java, JavaScript, Erlang, etc. (see `https://en.wikipedia.org/wiki/QuickCheck`)
  - `https://hackage.haskell.org/package/QuickCheck`

Analysis of program properties

Specification and properties

Program analysis and verification always concern the specification of a *property*. Examples of properties:

- Hoare triples in axiomatic semantics establish properties on the basis of the pre- and postconditions
- for concurrent (and/or reactive) programs, properties like *deadlock freedom* or *the need for a machine to react on an event like pressing a button* can be specified
- methods to check specific properties (e.g., the absence of *null pointers* or *divisions-by-zero*) can be implemented

Analysis of program properties

Analysis techniques

There are several approaches to program analysis. Two important examples:

- *Model checking*: properties concerning the temporal behavior of programs are specified. For instance, *is it possible for a program to run into a deadlock?* The analysis checks whether one such property holds for the program.

- By adding *assertions* (i.e., formulas of some logic) in the code, we can use a number of tools to check in compilation (or execution) time whether the assertions are violated.

Analysis of program properties

Analysis techniques

There are several approaches to program analysis. Two important examples:

- *Model checking*: properties concerning the temporal behavior of programs are specified. For instance, *is it possible for a program to run into a deadlock?* The analysis checks whether one such property holds for the program.
  - Learn more: MFI (IS branch)
- By adding *assertions* (i.e., formulas of some logic) in the code, we can use a number of tools to check in compilation (or execution) time whether the assertions are violated.
  - See the two forthcoming examples

Analysis of program properties

Use of assertions

- Remind: analysis should rely on a formal semantics
- With an axiomatic semantics we can
  - analyze program properties
  - use information about these properties to improve other analysis, verification or testing techniques

JML: Java Modeling Language

- It is a notation that can be used in several tools. These tools can be
  - dynamic (*runtime assertion checking*):
    - *runtime* checking: `jmlc` (`http://www.dc.fi.udc.es/ai/tp/practica/jml/JML/docs/man/jmlc.html`)
    - test set generation: `jmlunit` (`http://www.eecs.ucf.edu/~leavens/JML-release/docs/man/jmlunit.html`)
  - static (*static verification*):
    - assertion checking: `ESC/Java2` (`http://kindsoftware.com/products/opensource/ESCJava2/`),
    - based on the weakest precondition calculus: `JACK` (`http://www-sop.inria.fr/everest/soft/Jack/jack.html`),
    - deductive verification: `Krakatoa` (`http://krakatoa.lri.fr/`)

JML: Java Modeling Language

- Properties related with concepts like *aliasing*, inheritance, *side effects*, etc. can be specified.

JML annotations are introduced as special comments in Java programs:

```
//@ <JML specification>
/*@ <JML specification> @*/
```

JML: Java Modeling Language

Notation

keyword	use
`requires`	precondition
`ensures`	postcondition
`assert`	assertion
`pure`	the method introduces no *side effect*
`invariant`	class invariant
`loop_invariant`	loop invariant
`signals`	postcondition when exception
`signals_only`	possible exceptions given a precondition
`assignable`	attributes that can be modified by methods
`also`	combine specifications
`spec_public`	makes a variable public (to the spec.)

JML: Java Modeling Language

Available expressions:

expression	meaning
`\result`	value returned by the method
`\old(<expression>)`	value of the expression when entering the method
`a ==> b`	implication
`a <== b`	`b` implies `a`
`a <==> b`	if and only if

Besides, universal and existential quantification:

expression
`(\forall <decl>; <range-exp>; <body-exp>;)`
`(\exists <decl>; <range-exp>; <body-exp>;)`

JML: Java Modeling Language

Example 1

```
public class TickTockClock {

  ...some code here ...

  //@ protected invariant 0<=second && second<=59;
  protected int second;

  ...some code here ...

  //@ ensures 0 <= \result;
  //@ ensures \result <= 59;
  public /*@ pure @*/ int getSecond() {
    return second;
    }
```

JML: Java Modeling Language

Example 2

```
public class BankingExample {

  public static final int MAX_BALANCE = 1000;
  private int balance;

  //@ private invariant balance>=0 && balance<=MAX_BALANCE;

  //@ ensures balance == 0;
  public BankingExample() { balance = 0; }

  //@ requires 0<amount && amount + balance<MAX_BALANCE;
  public void credit(int amount) { balance += amount;}

  //@ requires 0<amount && amount <= balance;
  public void debit(int amount) { balance -= amount;}
  }
```

Dafny: analysis within .NET

A JML-like notation + analysis

- Tool for the static analysis of programs
- A specification language for assertions, preconditions, postconditions,... is used
- Specifications are used for verification

Specific of Dafny

- It is a hybrid language: functional and object-oriented
- Executed as **part of the compiler**
- Programmers interact with the tool to modify the program
  - similar to type or error correction during an IDE interaction session
- web version available; there also is a standalone version to be used within Visual Studio

Dafny: analysis within .NET

Notation

Some constructs:

keyword	use
`requires`	precondition
`ensures`	postcondition
`modifies`	modifiable element
`assert`	assertion

expression	meaning
`multiset`	manages a set of values
`old(exp)`	initial value of `exp` in a method
`predicate...`	defines a predicate
`forall...`	universal quantification
`exists...`	existential quantification
`if [...then ...else]`	conditional

Dafny: analysis within .NET

Example

```
var x: int;
var y: int;
var tmp: int;

method Swap()
  modifies this;
  ensures x==old(y) && y==old(x);
  { tmp := x;
    x := y;
    y := tmp;
  }
```

References (1/2)

Software Testing:

- *Software Reliability Methods*. Doron A. Peled. Springer, 2001. (Capítulo 9).

Semantics:

- Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993 (Capítulo 2).

Test set generation based on symbolic execution:

- James C. King. *Symbolic execution and program testing*. Comm. of the ACM, 19(7):385-394, 1976.
- L.A. Clarke. *A System to Generate Test Data and Symbolically Execute Programs*. IEEE Transacions on Software Engineering. 2(3):215-222, 1976.

References (2/2)

Java Modeling Language (ESC/Java2):

- *Advanced Specification and Verification with JML and ESC/Java2*. P. Chalin, J. R. Kiniry, G. T.Leavens and E. Poll. Formal Methods for Components and Objects, 2006.

Dafny:

- *Using Dafny, an Automatic Program Verifier*. L. Herbert, K. Rustan, M. Leino and J. Quaresma. 2011.

Static analysis:

- *Principles of Program Analysis*. F. Nielson, H. R. Nielson and C. Hanking. Springer, 2004.