Software Engineering using Formal Methods

Lecture Notes by Thomas Schulz

Last Update: March 2, 2012 - 16:18

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Lecture held in WS 2011/12 by:
Prof. Dr. Reiner Hähnle
Dr. Richard Bubel

Contents

Disclaimer

This document is a summary of the lecture “Software Engineering using Formal Methods”. It was created for personal study and exam preaparation.

The author do not warrant or assume any legal responsibility for the accuracy, completeness, or usefulness of any information described in this document.

Motivation

Defects in Software can cause (financially) *severe* and *omnipresent* failures. Unfortunately, best practices known from other engineering disciplines are not adaptable to developing software (see Table 1).

Table 1: Hardware vs. Software

Best Practices for Hardware	**Why not for Software?**
Redundancy	Does not help against bugs!
Separation of Subsystems	Usually not (completely) possible!
Precise Calculation	Software is too complex!
Follow patterns	No mature methods in SE!
Robust Design	Local Errors often affect the whole system!

One possible approach is to test a software product, but this shows only the *presence* of errors, not their *absence*. Besides, testing is always incomplete, expensive and time consuming.

This motivates the topic of the lecture. Formal methods provide tools to verify correctness and completeness. The idea for both parts of this course is to provide a specification of a system, provide a specification of the requirements and (semi-)automatically check whether the specification meets the requirements. The first part discusses an approach for concurrent processes while the second part adresses object-oriented programs.

1 Modeling & Model Checking with PROMELA & SPIN

1.1 PROMELA Introduction

- put variable declarations at start
- non-initialized arithmetic variables are set to 0
- the values $\mathbb{B} = \{$`true,false`$\}$ are syntactic sugar for the bit values 1 and 0
- there is at most one `mtype` *(message type)* per program
- first statement after "`::`" is considered as the *guard*
- if more than one guard is true, then one is randomly chosen
- use "`->`" after command that starts with "`::`", not "`;`"
- *blocking* occurs if no guard is true
- feel free to declare constants with "`#define C val`"
- there are two possibilities to express a for-loop
  1. "`for (i : 1 .. 6)`" iterates over `i` from 1 to 6
  2. "`for (i in a)`" iterates over all indices `i` of array `a`

1.2 Verifying with Spin

1.3 Modeling Concurrency

1.4 Introduction to PROMELA/SPIN

1.5 Modeling Distribution

1.6 Propositional Logic & Temporal Logic (1)

1.7 Temporal Logic (2)

1.8 Channels & Linear Temporal Logic

1.9 Temporal Model Checking with Spin

2 Modeling & Verification with JML & KeY

2.1 First-Order Logic (Syntax and Semantics)

2.2 First-Order Logic – Calculus

2.3 JML (1)

2.4 JML (2)

2.5 Dynamic Logic 1

2.6 Dynamic Logic Calculus

2.7 Proof-Obligations

2.8 Loop Invariants

3 FAQ

3.1 [PROMELA] What are the exact semantics of "`atomic`" and "`d_step`"?

The keyword "`atomic`" describes a *weakly*, the keyword "`d_step`" a *strongly* atomic sequence. The difference lies in the interruption condition: The first can *only* be interrupted if a statement is not executable while the second cannot be interrupted *at all* (see slide 21 in "Concurrent Programming").

3.2 [General] What is the difference between "deadlock", "livelock" and "starvation"?

Processes are in a *deadlock*, if each process waits for an event that only other processes can trigger. A *livelock* is a concrete deadlock where two or more processes are not waiting, but are trapped in a loop and cannot complete their task. *Starvation* describes the state of a process that is waiting for an event that does not occur.

3.3 [PROMELA] How do the statement types relate to their executability?

Table 2 illustrates the answer (see slide 28 in "Distributed Programming").

Table 2: Executability of Statements

Statement Type	Executability
assignments	always
assertions	always
print statements	always
expression statements	iff value is $\neg 0 \vee \neg$`false`
`send ! msg`	iff message queue is not full, i.e. $n < cap$
`request ? msg`	iff `request` is not empty, i.e. $n > 0$

3.4 [LTL] What is the meaning of "$\Box\Diamond\phi$"? How is this justified?

(see slide 25 in "LTL (1)")

3.5 [LTL] How are "$\Box$" and "$\Diamond$" related? What is the intuition?

3.6 [JML] How can `(\forall τ x; a; b)` and `(\exists τ x; a; b)` be translated?

According to slide 34 in "JML (1)", the first statement is equivalent to an *implication*, `(\forall τ x; a ==> b)`, while the second expression has the same semantics as a *conjunction*, `(\exists τ x; a && b)`

3.7 [JML] Is "`/*@ non_null @*/`" always the opposite of "`/*@ nullable @*/`"?

No! If an array is specified as `/*@ nullable @*/`, then the array itself may be a null pointer or the *elements* of this array can be null. On the other hand, if an array is declared as `/*@ non_null @*/`, then neither the array nor its elements are null (see slide 30 in "JML (2)").

3.8 [JML] Is "`/*@ pure @*/`" and "`/*@ assignable \nothing @*/`" the same?

No! The scope of `/*@ assignable \nothing @*/` is *local* to each specification case of a method. Specifying a method as `/*@ pure @*/` is a *global* statement and also prohibits non-terminationand exceptions (see slide 8 in "Proof Obligations").

3.9 [DL-Calculus] Why there are three steps in the "loopInvariant"-rule?

First, the case *initially valid* proves that the loop invariant holds prior to the loop execution. Second, the step *preserved* ... . Third, the *use case* ... (see slides 7 ff. in "Loop Invariants").