Course Outline

- L1: Introduction
  - testing vs. verification, embedded systems, …
- L2: Specification formalisms
  - labeled transition systems, UML / OCL, {CSL | JML}, …
- L3: Test generation
  - Chinese postman algorithm, transition tours, coverage, …
- L4: Test execution
  - testing algorithms in detail, e.g. ParTeG, SpecExplorer
- L5: Test evaluation and assessment
  - oracle problem, mutation analysis
Testing and Fault Models

• Problem: What is a “good” test suite?
  ▪ Much work has been done in trying to give a formal answer

• Compare two LTSs A and B which are only “slightly” different: Test suite T is ”good” if it can discover (the existence of) a difference
  ▪ B is called a mutant of A, the difference is called a fault

• Several methods for generation of test suites which are good in comparing LTSs exist
  ▪ transition tour algorithm
  ▪ W method
  ▪ unique I/O method

Transition Tour

• Nait 81: For a given LTS S, a transition tour is a sequence which takes S from the initial state $s_0$, traverses every transition at least once, and returns to $s_0$

• Example:

\[ \begin{align*}
S_0 & \xrightarrow{t_1} S_1 \xrightarrow{t_2} S_2 \\
& \quad \xrightarrow{t_3} S_3 \xrightarrow{t_4} S_4 \\
\end{align*} \]

Transition tour
TT: $t_1, t_2, t_1, t_1, t_1, t_1$

TT (input/expected output): a/1.b/2.a/1.a/2.b/2.a/1.b/2.a/2.b/2

detects output faults in the SUT
How to Construct a Transition Tour

- Chinese postman algorithm (named after Mei Ko Kwan [1962])
  - postman must deliver letters alongside roads of a district
    (NP-complete optimization problem)
- Euler’s “Königsberger Brückenproblem”
  - solvable iff strongly connected and for all states, in-degree=out-degree
  - In order to make the LTS eulerian, we may have to traverse certain transitions several times
- Hierholzer’s algorithm
  - use depth-first-search until you hit a cycle
  - extend the cycle at the first junction

Example

Algorithm can be formulated by extending DFS for all-transitions coverage (homework)
Mutation Analysis

• Testing to prove equivalence (or, in fact, to prove any preorder relation between models) has not been very successful (personal opinion).
  ▪ Even a “complete” test suite may miss errors in the SUT
• However, we can use these ideas to assess the effectiveness of test generation algorithms
  ▪ How to convince your boss that you are using a “good” test case generator TeG?
  ▪ Assume we are given a model M
  ▪ Apply TeG to M and obtain test suite T(M)
  ▪ Inject a fault into M to get a slightly different model M’
  ▪ if T(M) fails on M’ this is an indication that TeG is effective

• In order to make this argument sound, we need to
  ▪ repeat this process with several models M₁,…,Mₙ
  ▪ select mutation operators which model “real” faults, i.e. Mᵢ’ could be a mistake made by an implementer
  ▪ make sure that the effect of a mutation is not masked, i.e. it must be visible to the outside
• Resulting statements are of the type: “For the models M₁,…,Mₙ and mutation operators op₁,…,opₖ, G₁ can detect an average of 90% of all mutants, whereas G₂ can detect 93%.”
• Such statements have proven to be useful!
An Example: Door Control Unit

- Built into the door of a car
- Used to control electric windows, electric seats, interior lights, and some other
- Original requirements document (well, almost original) available on the internet (in German)

- Here: modeling one axis of seat movement
- Motor, button, sensor, controller
**Button**

```
b_forward/GEN(bp_forward);
```

```
b_center/GEN(bp_release);
```

**Motor**

```
m_forward/itsEnv->GEN(env_forward);
m_stop/itsEnv->GEN(env_stop);
```

```
m_limit/itsEnv->GEN(env_limit);
m_stop/itsEnv->GEN(env_stop);
m_back/itsEnv->GEN(env_back);
m_limit/itsEnv->GEN(env_limit);
```
Position Sensor

Control
Complete Seat Control

Procedure for Test Generation

- Model physical dependencies by an environment model (e.g. motor movement and sensor value)
- In this case, no separate user model is necessary (user can press any button at any time)
- Separate interfaces into „required“ and „provided“
- Use some automatic test generator for the automatic construction of a test suite
- Clean up the result (i.e., purge non-visible events, remove duplicate and included test sequences etc.)
Object-Oriented Modelling

• Class Diagrams
  ▪ assumed to be known…
  ▪ class contains data fields and methods
  ▪ private and public components, associations
  ▪ each object belongs to a class

UML 2

• 13 diagram types
• Common meta-model
• Instances (objects) can occur in several diagrams, different views onto the same thing
• A structure diagram, e.g. a class, defines a collection of objects with similar properties, attributes and methods
  ▪ signature and structure
• A behavioural diagram, e.g. a statechart, defines a collection of behaviours of objects
  ▪ change of model in time
OCL

- Object constraint language
- (important) Part of UML
- Specifies constraints on model elements
  - “A constraint is a restriction on one or more values of (part of) an object-oriented model or system”
- Different kinds of constraints
  - **Invariant** - a constraint that must always be met by all instances of a class
  - **Precondition** of an operation - a constraint that must always be true before the execution of the operation
  - **Postcondition** of an operation - a constraint that must always be true after the execution of the operation
  - **Guard** of a transition – a constraint that must be met before a state transition fires

Connection UML-OCL

- Each OCL formula can mention objects defined in UML diagrams
- OCL semantics „relative“ to a certain UML model
- Expressions in OCL „add information“ to UML diagrams
  - not a „stand-alone“ specification language
  - OCL for constraints which cannot be expressed by diagrams
    - e.g. number of passengers in a flight is less or equal to the number of seats in the plane
    - fixed interpretation (includes arithmetics)
Standardisation

- early 1990s: Steve Cook and John Daniels, Syntropy design method
  - adaptation of Z to OOA
- 1996: OMG request for proposal; IBM and ObjectTime Ltd. submit joint proposal
- 1997: OCL 1.1
- 1.5.2006: OCL 2.0

A Simple Example

Informal Description
1. The age of a person is not negative.
2. Each person is younger than the parents.
3. After the birthday a person is one year older.
4. Each person has exactly two parents. If somebody becomes parent, the set of children is nonempty and the number is increased by one.
5. Only adults may own a car
6. The first registration date of a car is after its built

OCL-Constraint
- Context Person inv: self.alter >=0
- Context Person inv: self.Eltern->forall(e|e.Alter>=self.Alter)
- Context Person::habGeburtstag() post: self.Alter=self.Alter@pre+1
- Context Person inv: self.Eltern->size()='2
- Context Person inv: Alter<18 implies autos->size()='0
- Context Auto inv: Erstzulassung->>Baujahr
OCL Types

- Basic type "void": `void::oclIsUndefined`
- Boolean, Integer, Real, String, …
- Enumerations: `enum{val1, val2, val3}`
- Set, Bag, Sequence
  - Union: `Bag{2,2,3}, Bag{3,3}`
- Class types: each class name can be used as a type
  - Most general class/type: `OclAny`
- Strong typing rules, subtyping according to OO
  - Integer is subtype of Real
  - Each type conforms to each of its supertypes

OCL Contexts

- The context attaches a constraint to a particular modelling element
  - Context `<class name>::<operation>(<parameters>)(<Boolean OCL expression>)`
- Dot-notation allows access to other (visible) modelling elements or objects
  - `Meeting.start, Passagier.name`
- "self" always refers to the object identifier from which the constraint is evaluated
  - `context Meeting inv: self.end > self.start`
- Access to collections via `-`
  - `Passagier.buchung->size()`
**OCL Operators**

- **Boolean operators:** =, and, or, xor, not, implies, if-then-else, forall
  - sequential evaluation, i.e. \((true \ or \ undefined) = true\)
  - \(x \ implies \ y = (not \ x) \ or \ (x \ and \ y)\)

- **@pre-operator**
  - refers to the *previous* value of an object in a postcondition

- **select-operator**
  - \(collection->select(condition)\) is any element of the collection satisfying the condition
  - e.g. \(Passagier.buchung->select(datum=TODAY)\)

**Pre- and Postconditions**

- **Used to constrain methods**

  - **context Meeting :: confirm()**
    - **pre:** Calendar.freeTimeSlot(self.start, self.duration())
    - **post:** self.isConfirmed = true
  - **context Meeting :: duration(): Integer**
    - **post:** result = self.end - self.start
  - **context Meeting :: shift(d: Integer)**
    - **post:** start = start@pre + d and end = end@pre + d
Iterations

\( c \rightarrow \text{iterate}(x : T1;\ a : T2 = \text{exp0} \mid \text{exp}) \)
- \( c \) is of type \( \text{Collection}(T) \)
- \( x \) is a name for a variable (sometimes called the "cursor")
- \( x \) is of a type \( T1 \) which is conformant to \( T \)
- \( a \) is the name for a variable (sometimes called the "accumulator")
- \( a \) is of type \( T2 \)
- \( \text{exp0} \) is an OCL expression giving a value of type \( T2 \)
- \( \text{exp} \) is an OCL expression using the variables \( x \) and \( a \) and giving a value of type \( T2 \)
- The type of the whole "iterate" expression is of type \( T2 \)

Predefined iterations

- \( \text{c: Collection(T) - > size: Integer} \)
  \( \text{post: result = c->iterate(e: OclAny; a: Integer = 0 \mid a+1)} \)
- \( \text{c: Collection(T) - > isEmpty: Boolean} \)
  \( \text{post: result = c->size = 0} \)
- \( \text{c: Collection(T) - > forAll (expr: OclExpression): Boolean} \)
  \( \text{post: result = c->iterate(e: OclAny; a: Boolean = true \mid a and expr)} \)
- \( \text{c: Collection(T) - > exists (expr: OclExpression): Boolean} \)
  \( \text{post: result = c->iterate(e: OclAny; a: Boolean = false \mid a or expr)} \)
Model-based Design and Model-based Testing

- Often: same syntax, different pragmatics
  - e.g. test cases can be formulated in Java
  - e.g. system spec can be formulated with LTL