Systeme Hoher Qualität und Sicherheit
Vorlesung 6 vom 25.11.2013: Detailed Specification, Refinement & Implementation

Christoph Lüth & Christian Liguda

Universität Bremen

Wintersemester 2013/14

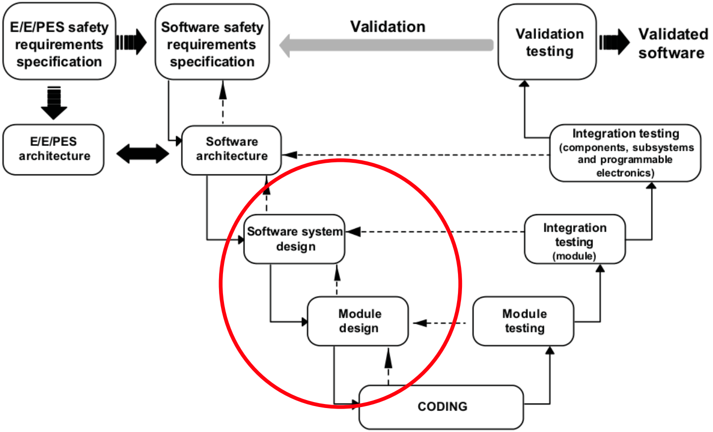
Where are we?

- Lecture 1: Concepts of Quality
- Lecture 2: Concepts of Safety and Security, Norms and Standards
- Lecture 3: Quality of the Software Development Process
- Lecture 4: Requirements Analysis
- Lecture 5: High-Level Design & Formal Modelling
- **Lecture 6: Detailed Specification, Refinement & Implementation**
- Lecture 7: Testing
- Lecture 8: Static Program Analysis
- Lecture 9: Verification with Floyd-Hoare Logic
- Lecture 10: Verification Condition Generation
- Lecture 11: Model-Checking with LTL and CTL
- Lecture 12: NuSMV and Spin
- Lecture 13: Concluding Remarks

Your Daily Menu

- **Refinement**: from abstract to concrete specification
- **Implementation**: from concrete specification to code
- Running examples: the safe autonomous robot, the birthday book

Design Specification



- At this point, we want to be **relate** implementation to the more abstract specifications in the higher lever, and have a **systematic** way to go from higher to lower levels (**refinement**).

Refinment in the Development Process

- Recall that we have **horizontal** and **vertical** structuring.
- **Refinement** is a **vertical** structure in the development process.
- The simplest form of refinement is **implicational**, where an implementation I implies the abstract requirement A

$$I \Rightarrow A$$

- Recall that refinement typically preserves **safety** requirements, but not **security** — thus, there is a systematic way to construct safe systems, but not so for secure ones.

The Autonomous Robot: Basic Types

- We first declare a datatype for the time:

  $[Time]$

- We then declare the robot parameters, and the state of the world — these are the things which do not change.

  RobotParam
  $cont : POLY$

- Obstacles are just a set of points (instead of polygons)

  World
  RobotParam
  $obs : \mathbb{P}\, VEC$

The Autonomous Robot: Safety Requirements

- The robot's state depends on the time, so we do not have pre/post conditions. It has a position vector, o, which determines the current contour polygon c.

$$
\begin{array}{|l}
\textit{Robot} \\
\hline
\textit{RobotParam} \\
c : \textit{Time} \rightarrow \textit{POLY} \\
o : \textit{Time} \rightarrow \textit{VEC} \\
\hline
c(t) = \textit{move}(\textit{cont}, o(t)) \\
\hline
\end{array}
$$

- Here is the **main safety requirement**: the robot is safe if its current contour never contains any obstacles.

$$
\begin{array}{|l}
\textit{RobotSafe} \\
\hline
\textit{Robot} \\
\hline
\forall\, t . c(t) \cap \textit{obs} = \emptyset \\
\hline
\end{array}
$$

The Autonomous Robot: Implementation

- When implementing the autonomous robot, we assume a **control loop** architecture, where a **control function** is called each T ms. It can read the**current** system state, and sets **control variables** which determine the system's behaviour over the next clock cycle.

- The cycle time ("tick") T is part of the robot parameters. We also add the braking accelaration a_{brk}.

$$
\begin{array}{l}
\textit{RobotParam} \\
\hline
cont : POLY \\
a_{brk} : \mathbb{Z} \\
T : \mathbb{Z} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{World} \\
\hline
RobotParam \\
obs : \mathbb{P}\, VEC \\
\hline
\end{array}
$$

The Autonomous Robot: Implementation

- This specifies the **control behaviour** of the robot.
- Velocity is given by the **linear velocity** *vel*, and steering angle ω. This describes the velocity vector *v* in polar form.
- This does not yet describe how the velocity is controlled.

$$
\begin{array}{|l}
\textit{Robot} \\
\hline
\textit{RobotParam} \\
\textit{vel}, \omega : \mathbb{Z} \\
v, o : \textit{VEC} \\
c : \textit{POLY} \\
\hline
c = \textit{move}\,(\textit{cont}, o) \\
v = \textit{cart}\,(\textit{vel}, \omega) \\
\hline
\end{array}
$$

- The function *cart* converts a vector in polar form to the cartesian form. A simple specification in Z might be this:

$$
\begin{array}{|l}
\textit{cart} : \mathbb{Z} \times R \rightarrow \textit{VEC} \\
\hline
\forall\, r : \mathbb{Z};\ \omega : R;\ p : \textit{VEC} \bullet \textit{cart}(r, \omega) = p \Rightarrow r * r = p.x * p.x + p.y *
\end{array}
$$

- Unfortunately, the Mathematical Toolkit does not support trigonmetric functions (or real numbers).

The Autonomous Robot: Control

- The velocity is controlled by two **input variables** $a?$ and $d\omega?$, which set the acceleration and change of steering angle for the next cycle. This determines *vel* and ω, and hence v.

RobotMoves

$\Delta Robot$
$\Xi World$
$a? : \mathbb{Z}$
$d\omega? : \mathbb{Z}$

$vel' = vel + a? * T$
$\omega' = \omega + d\omega? * T$
$o' = add\,(o, v')$

- This now describes the control loop behaviour of the robot.
- But when is it **safe**?

Moving and Driving Safely

- It is easy to say what it means for the robot to **move safely**: it will not run into any obstacles.

RobotMovesSafely

RobotMoves

$$cov\,(c, v') \cap obs = \emptyset$$

- Is that **enough**?

Moving and Driving Safely

- It is easy to say what it means for the robot to **move safely**: it will not run into any obstacles.

$$\begin{array}{|l}\hline \textit{RobotMovesSafely} \\ \textit{RobotMoves} \\ \hline cov\,(c, v') \cap obs = \emptyset \\ \hline \end{array}$$

- Is that **enough**?
- No, this will give us a **false sense** of safety — it only fails when it is **far too late** to **initiate** braking.
- To ensure safety here we would need:

$$RobotMovesSafely \Rightarrow RobotMovesSafely'$$

Braking and Safe Braking

- Our safety strategy: we must **always** be able to **brake safely**
- We first need to specify **braking** and **safe braking**. Braking is safe if the braking area is clear of obstacles.

RobotBrakes

$\Delta Robot$
$\Xi World$

$vel' = vel - a_{brk} * T$
$\omega' = \omega$
$o' = add(o, v')$

RobotBrakesSafely

$RobotBrakes$

$cov(c, brk(v, \omega, a_{brk})) \cap obs = \emptyset$

- **Implementing** the overall strategy: if we can move safely, we do, otherwise we brake.
- **Invariant**: we can always brake safely.

The Safe Robot: Implementation

- We drive **safe** if we **will** be able to brake safely.

RobotDrivesSafely

$\Delta Robot$

$\Xi World$

$$(cov\,(c, v') \cup cov\,(move\,(c, v'), brk\,(v', \omega', a_{brk}))) \cap obs = \emptyset$$

$$vel' = vel + a? * T$$

$$\omega' = \omega + d\omega? * T$$

$$o' = add\,(o, v')$$

- The safe robot implements the safety strategy:

$$RobotSafeImpl = RobotDrivesSafely \vee RobotBrakes$$

Showing Safety

- We need to **show:**

$$RobotSafeImpl \Rightarrow RobotMovesSafely$$
$$RobotSafeImpl \Rightarrow RobotMovesSafely'$$

- The first holds directly.
- The second holds because of the following:

$$RobotSafeImpl \Rightarrow RobotBrakesSafely'$$
$$RobotBrakesSafely \Rightarrow RobotMovesSafely$$
$$RobotBrakesSafely' \Rightarrow RobotMovesSafely'$$

Missing Pieces

- We start off at the origin (or anywhere else), and with velocity 0.
- We need to specify that initially we are **clear of obstacles**.

InitRobot

Robot

$$o = (0, 0)$$
$$vel = 0$$
$$\omega = 0$$
$$cont \cap obs = \emptyset$$

Summing Up

- The first, abstract, safety specification was *RobotSafe*.
- We implemented this via a second, more concrete specification *RobotSafeImpl*.
- Showing refinement required several lemmas.
- The general safety argument:
  - Safety holds for the initial position: $\mathit{InitRobot} \Rightarrow \mathit{RobotMovesSafely}$
  - Safety is preserved:
    $\mathit{RobotSafeImpl} \Rightarrow \mathit{RobotMovesSafely} \wedge \mathit{RobotMovesSafely}'$
  - Thus, safety holds always (proof by **induction**).

From Specification to Implementation

- How would we **implement** the birthday book?
- We need a **data structure** to keep track of names and dates.
- And we need to **link** this data structure with the **specification**.
- There are two ways out of this:
  - Either, the specification language also models datatypes (**wide-spectrum language**).
  - Or there is fixed mapping from the specification language to a programming language.

Implementing Arrays

- In Z, arrays can be represented as functions from $\mathbb{N}_1$. Thus, if we want to keep names and dates in arrays (linked by the index), we take

$$names : \mathbb{N}_1 \to NAME$$
$$dates : \mathbb{N}_1 \to DATE$$

- To look up $names[i]$, we just apply the function: $names(i)$.

- To assignment $names[i] := v$, we change the function with the **pointwise update operator** $\oplus$:

$$names' = names \oplus \{i \mapsto v\}.$$

Implementing the Birthday Book

- We need a variable *hwm* which indicates how many date/name pairs are known.

- The axiom makes sure that each name is associated to exactly one birthday.

BirthdayBookImpl

$$names : \mathbb{N}_1 \rightarrow NAME$$
$$dates : \mathbb{N}_1 \rightarrow DATE$$
$$hwm : \mathbb{N}$$

$$\forall\, i, j : 1 \,..\, hwm \bullet$$
$$\quad i \neq j \Rightarrow names(i) \neq names(j)$$

Linking Specification and Implementation

- We need to **link** specification and implementation.
- This is done in an abstraction or linking schema:

$$
\begin{array}{l}
\textit{Abs} \\
\hline
\textit{BirthdayBook} \\
\textit{BirthdayBookImpl} \\
\hline
\textit{known} = \{\, i : 1 \mathrel{..} \textit{hwm} \bullet \textit{names}(i) \,\} \\
\forall i : 1 \mathrel{..} \textit{hwm} \bullet \\
\qquad \textit{birthday}(\textit{names}(i)) = \textit{dates}(i) \\
\hline
\end{array}
$$

- This specificies how *known* and *birthday* are reflected by the implementing arrays.

Operation: Adding a birthday

- Adding a birthday changes the **concrete state**:

$$
\begin{array}{l}
\textit{AddBirthdayImpl} \\
\hline
\Delta \textit{BirthdayBookImpl} \\
name? : NAME \\
date? : DATE \\
\hline
\forall i : 1 \mathrel{..} hwm \bullet name? \neq names(i) \\
\\
hwm' = hwm + 1 \\
names' = names \oplus \{hwm' \mapsto name?\} \\
dates' = dates \oplus \{hwm' \mapsto date?\} \\
\hline
\end{array}
$$

- We need to show that the pre- and post-states of *AddBirthday* and *AddBirthdayImpl* are related via *Abs*.

Showing Correctness of the Implementation

- Assume a state where the precondition of the specification holds, find the corresponding state of the implementation via *Abs*, and show that this state satisfies the precondition.

- Similarly, assume a pair of states where the invariant of *AddBirthdayBook* holds, find the corresponding states of the implementation via *Abs*, and show that they satisfy the invariant.

Operation: Finding a birthday

- We specify that the found day corresponds to the name via an index i.

$$
\begin{array}{|l}
\hline
\textit{FindBirthdayImp} \\
\Xi \textit{BirthdayBookImpl} \\
\textit{name}? : \textit{NAME} \\
\textit{date}! : \textit{DATE} \\
\hline
\exists\, i : 1 \mathrel{..} \textit{hwm} \bullet \\
\qquad \textit{name}? = \textit{names}(i) \land \textit{date}! = \textit{dates}(i) \\
\hline
\end{array}
$$

- Note that we are still some way off a concrete implementation — we do not say how we **find** the index i.
- To formally show that an iterative loop from 1 to *hdw* always returns the right i, we need the **Hoare calculus** (later in these lectures); presently, we argue **informally**.

Summary

- We have seen how we **refine** abstract specifications to more **concrete** ones.
- To **implement** specifications, we need to relate the specification language to a programming language
  - In Z, there are some types which correspond to well-known datatypes, such as finite maps $\mathbb{N}_1 \to T$ and arrays of T.
- We have now reached the **bottom** of the V-model. Next week, we will climb our way up on the right-hand side, starting with **testing**.