Lecture 14: UML State Machines

2016-06-30

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal
Albert-Ludwigs-Universität Freiburg, Germany

Top Area Architecture & Design: Content

- Introduction and Vocabulary
- Principles of Design
  - (i) modularity
  - (ii) separation of concerns
  - (iii) information hiding and data encapsulation
  - (iv) abstract data types, object orientation
- Software Modelling
  - (i) views and viewpoints, the 4+1 view
  - (ii) model-driven/-based software engineering
  - (iii) Unified Modelling Language (UML)
  - (iv) modelling structure
    - (a) (simplified) class diagrams
    - (b) (simplified) object diagrams
    - (c) (simplified) object constraint logic (OCL)
  - (v) modelling behaviour
    - (a) communicating finite automata
    - (b) Uppaal query language
    - (c) implementing CFA
    - (d) an outlook on UML State Machines
- Design Patterns
- Testing: Introduction

CFA at Work

- Question: Is it (at all) possible to have no water in the vending machine model? (Otherwise, the design is definitely broken.)
- Approach: Check whether a configuration satisfying $w = 0$ is reachable, i.e. check $N_{VM} \models \exists \Diamond w = 0$. for the vending machine model $N_{VM}$.

Design Check: Scenarios

- Question: Is the following existential LSC satisfied by the model? (Otherwise, the design is definitely broken.)
- Approach: Use the following newly created CFA ‘Scenario ’ end_of_scenario instead of User and check whether location end_of_scenario is reachable, i.e. check $N'_{VM} \models \exists \Diamond \text{Scenario}.end_of_scenario$. for the modified vending machine model $N'_{VM}$.

Design Verification: Invariants

- Question: Is it the case that the “tea” button is only enabled if there is e1.50 in the machine? (Otherwise, the design is broken.)
- Approach: Check whether the implication tea_enabled $\Rightarrow$ Coin Validator. have_c150 holds in all reachable configurations, i.e. check $N_{VM} \models \forall \Box \text{tea_enabled} \Rightarrow \text{Coin Validator. have_c150}$ for the vending machine model $N_{VM}$.
Design Verification: Sanity Check

• Question: Is the "tea" button ever enabled? (Otherwise, the considered invariant $\text{tea}_{\text{enabled}} \Rightarrow \text{Coin Validator}$. $\text{have_c150}$ holds vacuously.)

• Approach: Check whether a configuration satisfying $\text{water}_{\text{enabled}} = 1$ is reachable. Exactly like we did with $\text{w} = 0$ earlier.

Design Verification: Another Invariant

• Question: Is it the case that, if there is money in the machine and water in stock, that the "water" button is enabled?

• Approach: Check $\text{NVM} = \forall \Box (\text{Coin Validator}$. $\text{have_c50}$ or $\text{Coin Validator}$. $\text{have_c100}$ or $\text{Coin Validator}$. $\text{have_c150}$) imply $\text{water}_{\text{enabled}}$. drink_ready $\text{have_c150}$ $\text{have_e1}$ $\text{have_c100}$ $\text{have_c50}$ $\text{idle}$ $\text{OK?}$ $\text{OK?}$ $\text{OK?}$ $\text{OK?}$ $\text{E1?}$ $\text{tea_enabled} := (t > 0)$ $\text{C50?}$ $\text{water_enabled} := (w > 0)$ $\text{tea_enabled} := (t > 0)$ $\text{E1?}$ $\text{soft_enabled} := (s > 0)$ $\text{C50?}$ $\text{soft_enabled} := (s > 0)$ $\text{C50?}$ $\text{water_enabled} := (w > 0)$

Recall: Universal LSC Example

LSC: buy water
AC: true
AM: invariant
I: strict
User
CoinValidator
ChoicePanel
Dispenser
C
50
WATER
¬ ($\text{C50}! \lor \text{E1}! \lor \text{pSOFT}! \lor \text{pTEA}! \lor \text{pFILLUP}!$)

$\text{water}_{\text{in}} \_\text{stock}$

$\text{OK}$

$\text{¬ ($\text{dSoft}! \lor \text{dTEA}$!)}$

Uppaal Architecture

server
verifyta
yes/no/don't know
.xml
.trc
.C++
.Java
Case Study: Wireless Fire Alarm System

(R1) The loss of the ability of the system to transmit a signal from a component to the central unit is detected in less than 300 seconds.

$\bigwedge i \in \text{C} \Box (\lceil \text{FAIL} = i \land \neg \text{DET}_i \rceil = \ell \leq 300 \text{ s})$

(R2) A single alarm event is displayed at the central unit within 10 seconds.

$\bigwedge i \in \text{C} \lceil \text{ALARM}_i \rceil = \bigwedge \Box (\lceil \neg \text{DISP}_i \rceil = \ell \leq 10 \text{ s})$, where

\begin{align*}
\text{FAIL} &\quad \text{Component failure} \\
\text{DET} &\quad \text{Alarm detection} \\
\text{DISP} &\quad \text{Alarm display}
\end{align*}
A CF A Model Is Software

Definition.
Software is a finite description $S$ of a (possibly infinite) set $\llbracket S \rrbracket$ of (finite or infinite) computation paths of the form

$$\sigma_0 \alpha_1 \rightarrow \sigma_1 \alpha_2 \rightarrow \sigma_2 \cdots$$

where
- $\sigma_i \in \Sigma$, $i \in \mathbb{N}_0$, is called state (or configuration), and
- $\alpha_i \in A$, $i \in \mathbb{N}_0$, is called action (or event).

The (possibly partial) function $\llbracket \cdot \rrbracket : S \mapsto \llbracket S \rrbracket$ is called interpretation of $S$.

Let $C(A_1, \ldots, A_n)$ be a network of CFA.
- $\Sigma = \text{Conf}$
- $A = \text{Act}$
- $\llbracket C \rrbracket = \{ \pi = \langle \vec{\ell}_0, \nu_0 \rangle \lambda_1 \rightarrow \langle \vec{\ell}_1, \nu_1 \rangle \lambda_2 \rightarrow \langle \vec{\ell}_2, \nu_2 \rangle \lambda_3 \rightarrow \cdots | \pi \text{ is a computation path of } C \}$.

Note: the structural model just consists of the set of variables and the locations of $C$.

Model-Driven Software Engineering

- (Jacobson et al., 1992): "System development is model building."
- Model driven software engineering (MDSE): everything is a model.
- Model based software engineering (MBSE): some models are used.

Idea

<table>
<thead>
<tr>
<th>Structure</th>
<th>Declarative</th>
<th>Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>′</td>
<td>Declarative</td>
<td>′</td>
</tr>
<tr>
<td>Constructive</td>
<td>Behaviour</td>
<td>′</td>
</tr>
</tbody>
</table>

Implementation

Now that we have a CFA model $C(A_1, \ldots, A_n)$ (thoroughly checked using Uppaal), we would like to have software — an implementation of the model.

This task can be split into two sub-tasks:
(i) implement each CFA $A_i$ in the model by module $S_{A_i}$, (ii) implement the communication in the network by module $S_C$.

(This has, by now, been provided implicitly by the Uppaal simulator and verifier.)

Example

w := 3

\begin{align*}
FILLUP? & \quad w := 3 \\
DOK! & \quad w > 0 \\
DWATER? & \quad w := w - 1
\end{align*}

---

CF A Model Is Software
Example
Jacobson et al., 1992: "System development is model building.

Model-driven software engineering (MDSE): everything is a model.

Model-based software engineering (MBSE): some models are used.

Idea

Structure

Declarative

Behaviour

Constructive

Behaviour

Implementation

elicit

refine

refine

refine

refine

requirements

model

requirements/constraints

design

system model

\mid \mid = \mid \mid = \mid \mid = \mid \mid = ?

generate/program

CFA at Work

continued

design checks and verification

Uppaal architecture

case study

CFA vs. Software

a CFA model is software

implementing CFA

Recall MDSE

UML State Machines

Core State Machines

steps and run-to-completion steps

Hierarchical State Machines

Rhapsody

UML Modes

Event Pool and Run-to-Completion

\begin{equation}
\begin{aligned}
&x \in \mathbb{C},
&y \in \mathbb{D},
&z \in \mathbb{E}.
\end{aligned}
\end{equation}

\begin{equation}
\begin{aligned}
&s_1, s_2, s_3 \in \mathbb{S}.
\end{aligned}
\end{equation}

\begin{equation}
\begin{aligned}
&\forall \phi \in \mathbb{E}, (\phi \in \mathbb{Expr})
&\forall \psi \in \mathbb{S}, (\psi \in \mathbb{Act})
\end{aligned}
\end{equation}

\begin{equation}
\begin{aligned}
&\forall \alpha \in \mathbb{Act}, (\alpha \in \mathbb{Act})
&\forall \beta \in \mathbb{Expr}, (\beta \in \mathbb{Expr})
\end{aligned}
\end{equation}
Recall: definition "model" (Glinz, 2008, 425):

(iii) the pragmatic attribute, i.e. the model is built in a specific context for a specific purpose.

Examples for context/purpose:
- Floorplan as sketch
- Floorplan as blueprint
- Floorplan as program
- Wiring plan
- Windows
- ...

With UML it's the Same

The last slide is inspired by Martin Fowler, who puts it like this:

"[...] people differ about what should be in the UML because there are differing fundamental views about what the UML should be. I came up with three primary classifications for thinking about the UML: 

UmlAsSketch, UmlAsBlueprint, and UmlAsProgrammingLanguage.

([...] S. Mellor independently came up with the same classifications.)

So when someone else's view of the UML seems rather different to yours, it may be because they use a different UmlMode to you.

Claim:

- This not only applies to UML as a language (what should be in it etc.),
- but at least as well to each individual UML model.

Sketch

In this UmlMode developers use the UML to help communicate some aspects of a system. [...]

Sketches are also useful in forward engineering. It can be easier to "see" the implementation rather than a complete specification. Hence my sound-bite "comprehensiveness is the enemy of comprehensibility."

Blueprint

[...] In forward engineering the idea is that blueprints are developed by a designer whose job is to build a detailed model. [...]

[...] Forward engineering tools support diagram drawing and back it up with a repository to hold the information. [...]

ProgrammingLanguage

If you can detail the UML enough, and provide semantics for everything you need in software, you can make the UML be your ProgrammingLanguage. If the promise is true, I don’t believe that graphical programming will succeed just because it's graphical. [...]

UML-Mode of the Lecture: As Blueprint

- The "mode" fitting the lecture best is AsBlueprint.

Goal:

- be precise to avoid misunderstandings.
- allow formal analysis of consistency/implication on the design level — find errors early.

Yet we tried to be consistent with the (informal semantics) from the standard documents OMG (2007a, b) as far as possible.

Plus:

- Being precise also helps to work in mode AsSketch: Knowing "the real thing" should make it easier to
  (i) "see" which blueprint(s) the sketch is supposed to denote, and
  (ii) to ask meaningful questions to resolve ambiguities.
We can use tools like Uppaal to check and verify CFA design models against requirements. CFA (and state charts) can easily be implemented using the translation scheme. Wanted: verification results carry over to the implementation. if code is not generated automatically, verify code against model.

UML State Machines are principally the same thing as CFA, yet provide more convenient syntax. Semantics uses asynchronous communication, run-to-completion steps in contrast to CFA. (We could define the same for CFA, but then the Uppaal simulator would not be useful any more.)

References


