Softwaretechnik / Software-Engineering

Lecture 13: UML State-Machines, UML, MBSE/MDSE, Design Principles

2019-07-04

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Topic Area Architecture & Design: Content

- Introduction and Vocabulary
- Software Modelling
  - model: views / viewpoints: 4+1 view
- Modelling structure
  - (simplified) Class & Object diagrams
  - (simplified) Object Constraint Logic (OCL)
- Modelling behaviour
  - Communicating Finite Automata (CFA)
  - Uppaal query language
- CFA vs. Software
- Unified Modelling Language (UML)
  - basic state-machines
  - an outlook on hierarchical state-machines
- Model-driven/-based Software Engineering
- Principles of Design
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
- Design Patterns
Content

- CFA vs. Software
- UML State Machines
  - Hierarchical State Machines
  - Core State Machines
  - steps and run-to-completion steps
  - Rhapsody
- Unified Modelling Language
  - Brief History
  - Sub-Languages
  - UML Modes
- Model-based/-driven Software Engineering
- Principles of (Good) Design
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
  - …by example

Uppaal Architecture
CFA at Work Cont’d
• Assume that query \( Q \) corresponds to a requirement on the system under development (e.g., an invariant), and \( \mathcal{N} \) is our design-idea model.

• Assume that the verification tool states \( \mathcal{N} \models Q \) (negative: no violation (or: error) found). What can we conclude from that?

\[
\begin{array}{c|c|c|c}
& \text{false negative} & \text{true negative} & \text{false positive} \\
\hline
\text{true positive} & & & \\
\hline
\text{false negative} & & & \\
\hline
\text{true negative} & & & \\
\end{array}
\]

→ if \( \mathcal{N} \) is a valid model of our idea, if the tool works correct, if if if ..., and if the system implements this design idea, and if environment assumptions hold, then the system will not fail due to an analysable design flaw.
Composite (or Hierarchical) States

- OR-states, AND-states \cite{Harel1987}.
- Composite states are about abbreviation, structuring, and avoiding redundancy.
Example

And That Would be Too Easy...

If an $F$ is available, which edges are enabled? What are the possible successor configurations?

(The full story: “Software Design, Modelling, and Analysis with UML” (in some winter semesters).)
UML Core State Machines

Event Pool and Run-To-Completion

annot ::= \[ \langle \text{event} \rangle \ast \langle \text{event} \rangle \ast \] \[] \langle \langle \text{guard} \rangle \rangle \] \[] \langle \langle \text{action} \rangle \rangle \]

with
- event ∈ \( \mathcal{E} \),
- guard ∈ \( \mathcal{Expr} \),
- action ∈ \( \mathcal{Act} \)

(optional)
(default: true, assumed to be in \( \mathcal{Expr} \))
(default: \text{skip}, assumed to be in \( \mathcal{Act} \))

\begin{center}
\begin{tabular}{c|c|c|c|c|c}
\hline
step & state & \( u_1 \) & stable & \( x \) & \( u_2 \) & stable & event pool \\
\hline
0 & \( s_1 \) & 1 & \( 27 \) & \( s_1 \) & 1 & \( E \) ready for \( u_1 \)
\hline
\end{tabular}
\end{center}
Event Pool and Run-To-Completion

<table>
<thead>
<tr>
<th>step</th>
<th>state</th>
<th>stable</th>
<th>x</th>
<th>state</th>
<th>stable</th>
<th>event pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s1</td>
<td>1</td>
<td>27</td>
<td>s1</td>
<td>1</td>
<td>E ready for u1</td>
</tr>
<tr>
<td>1</td>
<td>s2</td>
<td>1</td>
<td>27</td>
<td>s1</td>
<td>1</td>
<td>F ready for u2</td>
</tr>
</tbody>
</table>

Event Pool and Run-To-Completion

<table>
<thead>
<tr>
<th>step</th>
<th>state</th>
<th>stable</th>
<th>x</th>
<th>state</th>
<th>stable</th>
<th>event pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s1</td>
<td>1</td>
<td>27</td>
<td>s1</td>
<td>1</td>
<td>E ready for u1</td>
</tr>
<tr>
<td>1</td>
<td>s2</td>
<td>1</td>
<td>27</td>
<td>s1</td>
<td>1</td>
<td>F ready for u2</td>
</tr>
</tbody>
</table>
### Event Pool and Run-To-Completion

#### Table

<table>
<thead>
<tr>
<th>Step</th>
<th>State</th>
<th>Stable</th>
<th>$x$</th>
<th>State</th>
<th>Stable</th>
<th>Event Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$s_1$</td>
<td>1</td>
<td>27</td>
<td>$s_1$</td>
<td>1</td>
<td>$E$ ready for $u_1$</td>
</tr>
<tr>
<td>1</td>
<td>$s_2$</td>
<td>1</td>
<td>27</td>
<td>$s_1$</td>
<td>1</td>
<td>$F$ ready for $u_2$</td>
</tr>
<tr>
<td>2</td>
<td>$s_2$</td>
<td>1</td>
<td>27</td>
<td>$s_2$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$s_2$</td>
<td>1</td>
<td>27</td>
<td>$s_3$</td>
<td>0</td>
<td>$G$ ready for $u_1$</td>
</tr>
<tr>
<td>4a</td>
<td>$s_2$</td>
<td>1</td>
<td>0</td>
<td>$s_1$</td>
<td>1</td>
<td>$G$ ready for $u_1$</td>
</tr>
</tbody>
</table>
Event Pool and Run-To-Completion

Rhapsody Architecture

<table>
<thead>
<tr>
<th>step</th>
<th>state $u_1$</th>
<th>stable</th>
<th>$x$</th>
<th>state $u_2$</th>
<th>stable</th>
<th>event pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$s_1$</td>
<td>1</td>
<td>$27$</td>
<td>$s_1$</td>
<td>1</td>
<td>$E$ ready for $u_1$</td>
</tr>
<tr>
<td>1</td>
<td>$s_2$</td>
<td>1</td>
<td>$27$</td>
<td>$s_1$</td>
<td>1</td>
<td>$F$ ready for $u_2$</td>
</tr>
<tr>
<td>2</td>
<td>$s_2$</td>
<td>1</td>
<td>$27$</td>
<td>$s_2$</td>
<td>0</td>
<td>$G$ ready for $u_3$</td>
</tr>
<tr>
<td>3</td>
<td>$s_2$</td>
<td>1</td>
<td>$27$</td>
<td>$s_3$</td>
<td>0</td>
<td>$G$ ready for $u_1$</td>
</tr>
<tr>
<td>4.a</td>
<td>$s_2$</td>
<td>1</td>
<td>0</td>
<td>$s_1$</td>
<td>1</td>
<td>$G$ ready for $u_1$</td>
</tr>
<tr>
<td>5.a</td>
<td>$s_1$</td>
<td>1</td>
<td>0</td>
<td>$s_1$</td>
<td>1</td>
<td>$G$ ready for $u_1$</td>
</tr>
<tr>
<td>4.b</td>
<td>$s_1$</td>
<td>1</td>
<td>$27$</td>
<td>$s_3$</td>
<td>0</td>
<td>$G$ ready for $u_1$</td>
</tr>
<tr>
<td>5.b</td>
<td>$s_1$</td>
<td>1</td>
<td>0</td>
<td>$s_1$</td>
<td>1</td>
<td>$G$ ready for $u_1$</td>
</tr>
</tbody>
</table>
Unified Modelling Language (UML)

UML Overview (OMG, 2007, 684)

Figure A.5 - The taxonomy of structure and behavior diagram

Dobing and Parsons (2006)
**A Brief History of UML**

- **1970**: Software crisis term 'software engineering' as a discipline.
- **1980**: Visualise software with boxes, circles, arrows, automata, etc.
- **1990**: Object-Oriented Analysis/Design/Programming, Object-Modeling Technique (OMT) (Rumbaugh et al., 1990)
- **2000**: Statecharts (Harel, 1987), StateMate (Harel et al., 1990).
- **2000**: UML 0.x and 1.x ("the three amigos" joint effort); much criticised for lack of formality.
- **2000**: UML 2.x (split into infra- and superstructure documents and join again); syntax pretty defined; semantics natural language, informal.

The UML standard is published by the Object Management Group (OMG): "international, open membership, not-for-profit computer industry consortium.

"**UML Mode**" [http://martinfowler.com/bliki]

"[...] 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."

- Applies to **UML as such** (as a language),
- and to each individual **UML model**.
In this UML mode developers use the UML to help communicate some aspects of a system. In forward engineering the idea is that blueprints are developed by a designer whose job is to build a detailed design for a programmer to code up. That design should be sufficiently complete that all design decisions are laid out and the programmer should follow as a pretty straightforward activity that requires little thought. Blueprints require much more sophisticated tools than sketches in order to handle the details required for the task.

Programming Language

If you can detail the UML enough, and provide semantics for everything you need in software, you can make the UML be your programming language. Tools can take the UML diagrams you draw and compile them into executable code. The promise of this is that UML is a higher level language and thus more productive than current programming languages. The question, of course, is whether this promise is true. I don’t believe that graphical programming will succeed just because it’s graphical.
Model-based/-driven Software Engineering
• (Jacobson et al., 1992): “System development is model building.”
• Model-based software engineering (MBSE): some formal models are used.
• Model-driven software engineering (MDSE): all artefacts are (formal) models.
Tell Them What You’ve Told Them...

- We can use tools like Uppaal to
  - check and verify CFA design models against requirements.
- CFA (and state machines)
  - can easily be implemented using a translation scheme.
- UML State Machines are
  - principally the same thing as CFA, yet provide more convenient syntax.
- Semantics:
  - asynchronous communication,
  - run-to-completion steps
  (CFA: synchronous (or: rendezvous)).
- Mind UML Modes.
- Wanted: verification results carry over to the implementation.
  - if code is not generated automatically, verify code against model. → VL is
- Vocabulary: Model-based/-driven Software Engineering
Software Architecture — The software architecture of a program or computing system is the structure or structures of the system which comprise software elements, the externally visible properties of those elements, and the relationships among them. (Booch et al., 2003)
Goals and Relevance of Design

- The structure of something is the set of relations between its parts.
- Something not built from (recognisable) parts is called unstructured.

Design...
(i) structures a system into manageable units (yields software architecture),
(ii) determines the approach for realising the required software,
(iii) provides hierarchical structuring into a manageable number of units at each hierarchy level.

Oversimplified process model “Design”:

Content

- CFA vs. Software
- UML State Machines
  - Hierarchical State Machines
  - Core State Machines
  - steps and run-to-completion steps
  - Rhapsody
- Unified Modelling Language
  - Brief History
  - Sub-Languages
  - UML Modes
- Model-based/-driven Software Engineering
- Principles of (Good) Design
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
  - ... by example
Overview

1.) **Modularisation**
   - split software into units / components of manageable size
   - provide well-defined interface

2.) **Separation of Concerns**
   - each component should be responsible for a particular area of tasks
   - group data and operation on that data; functional aspects;
     functional vs. technical; functionality and interaction

3.) **Information Hiding**
   - the “need to know principle” / information hiding
   - users (e.g., other developers) need not necessarily know the algorithm
     and helper data which realise the component’s interface

4.) **Data Encapsulation**
   - offer operations to access component data,
     instead of accessing data (variables, files, etc.) directly

   → many programming languages and systems offer means
to enforce (some of) these principles technically: use these means.
1.) Modularisation

**modular decomposition** — The process of breaking a system into components to facilitate design and development; an element of modular programming.

IEEE 610.12 (1990)

**modularity** — The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.

IEEE 610.12 (1990)

- So, modularity is a property of an architecture.

- Goals of modular decomposition:
  - The structure of each module should be simple and easily comprehensible.
  - The implementation of a module should be exchangeable; information on the implementation of other modules should not be necessary. The other modules should not be affected by implementation exchanges.
  - Modules should be designed such that expected changes do not require modifications of the module interface.
  - Bigger changes should be the result of a set of minor changes. As long as the interface does not change, it should be possible to test old and new versions of a module together.

2.) Separation of Concerns

**Separation of concerns** is a fundamental principle in software engineering:

- each component should be responsible for a particular area of tasks,
- components which try to cover different task areas tend to be unnecessarily complex, thus hard to understand and maintain.

- Criteria for separation/grouping:
  - in object oriented design, data and operations on that data are grouped into classes,
  - sometimes, functional aspects (features) like printing are realised as separate components,
  - separate functional and technical components,
  - assign flexible or variable functionality to own components.
    Example: different networking technology (wireless, etc.)
  - assign functionality which is expected to need extensions or changes later to own components.
  - separate system functionality and interaction
    Example: most prominently graphical user interfaces (GUI), also file input/output
3.) Information Hiding

• By now, we only discussed the grouping of data and operations.
  One should also consider accessibility.

• The “need to know principle” is called information hiding in SW engineering. (Parnas, 1972)

  information hiding— A software development technique in which each module’s interfaces reveal as little as possible about the module’s inner workings, and other modules are prevented from using information about the module that is not in the module’s interface specification.

  IEEE 610.12 (1990)

• Note: what is hidden is information which other components need not know (e.g., how data is stored and accessed, how operations are implemented).

  In other words: information hiding is about making explicit for one component which data or operations other components may use of this component.

• Advantages / goals:
  • Hidden solutions may be changed without other components noticing, as long as the visible behaviour stays the same (e.g. the employed sorting algorithm).
  IOW: other components cannot (unintentionally) depend on details they are not supposed to.
  • Components can be verified / validated in isolation.

4.) Data Encapsulation

• Similar direction: data encapsulation (examples later).
  • Do not access data (variables, files, etc.) directly where needed, but encapsulate the data in a component which offers operations to access (read, write, etc.) the data.

  Real-World Example: Users do not write to bank accounts directly, only bank clerks do.
(i) information hiding and data encapsulation not enforced,
(ii) → negative effects when requirements change,
(iii) enforcing information hiding and data encapsulation by modules,
(iv) abstract data types,
(v) object oriented without information hiding and data encapsulation,
(vi) object oriented with information hiding and data encapsulation.

References


