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

Top 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

Vocabulary

Techniques

informal
semi-formal
formal

Uppaal Architecture

server verifyta yes / no / don't know
.xml .trc .q

C++
Java

CFA at Work Cont'd

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

Lecture 13: UML State-Machines, UML, MBSE, Design Principles
What Can We Conclude From Verification Results?

• Assume that query $Q$ corresponds to a requirement on the system under development (e.g., an invariant), and $N$ is our design-idea model.

• Assume that the verification tool states $N \models Q$ (negative: no violation (or: error) found).

What can we conclude from that?

$N \models Q$ → if the design idea does not satisfy $Q$, then if the tool works correctly, 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.

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

MainDefaultComponent.cpp

(build / make)

D.cpp
C.cpp
DfltCmp.exe

generate

Diagram

Timing
Communication

Example Model Execution (UML)

Unified Modelling Language (UML)

Event Pool and Run-To-Completion (UML)
A Brief History of UML

The UML standard is published by the Object Management Group (OMG): "international, open membership, not-for-profit computer industry consortium." The OMG was created to develop and promote standards for software development, including the UML.

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.

**UML-Mode of the Lecture: As Blueprint**

In forward engineering the idea is that blueprints are developed by a designer whose job is to build a system. The goal is to provide a detailed diagram of the system, which can be used to implement the system. Forward engineering tools support diagram drawing and back it up with a repository to hold the information.

Sketch

In this **UMLMode** developers use the UML to help communicate some aspects of a system. Sketches are also useful in reverse engineering. They are often not complete and can be extended. A detailed sketch is easier to understand than a complete specification. Hence my sound-bite "comprehensive-ness is the enemy of comprehensibility".

ProgrammingLanguage

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. The promise is true. I don’t believe that graphical programming will succeed just because it’s graphical.

Model-based/-driven Software Engineering

Model-based/-driven Software Engineering is the practice of using models to represent and manipulate software systems. Models are used to capture the functionality and structure of a system, and to reason about its behavior. Models can be used to guide the development of a system, and to verify its correctness.
Idea
Structure
Declarative Behaviour

Constructive Behaviour

Implementation
elicit refine
requirements
model
requirements/
constraints
design
system model

refine

• (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.

Approach: Transform vs. Write-Down-and-Check

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.

• Vocabulary: Model-based/-driven Software Engineering

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

• System Software System Component Software Component Module Interface Component Interface consists of 1 or more "is a" may be a has is an Software Architecture Architecture Architectural Description Design 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. (Bass et al., 2003)
As long as the interface does not change, it should be possible to test old and new versions of a module together.

• Technically (some of) these principles enforce minor changes should be the result of a set of bigger changes.

Modules should be designed such that
• information on the implementation of other modules should not be necessary.
• The other modules should not be affected by implementation exchanges.

• 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.
• So, facilitates design and development; an element of modular programming.

Goals of modular decomposition:
• Information Hiding and data encapsulation
• Modularity, separation of concerns
• Principles of (Good) Design

Separation of Concerns: different networking technology (wireless, etc.) vs. functional communication protocol (wireless, etc.)

Assignment flexible or variable functionality to own components.

• Data Encapsulation: most prominently graphical user interface (GUI), also file input/output exchange of (physical) messages using a certain technology (wireless, etc.)

3.) Separation of Concerns

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

Declarative Process Model Design

Overview

Oversimplified process model “Design”:

(i) the approach for realising the required software, determines structures of something is the set of relations between its parts

(ii) provides something not built from (recognisable) parts

(iii) provides at each hierarchy level. structures of something is the set of relations between its parts

The content of something is the set of relations between its parts

Principles of (Architectural) Design

Principles of (Good) Design

• Principles of (Architectural) Design

Model-based/-driven Software Engineering

• Principles of (Good) Design

UML Modes

• Principles of (Good) Design

Sub-Languages

• Principles of (Good) Design

Unified Modelling Language

• Principles of (Good) Design

Hierarchical State Machines

• Principles of (Good) Design

Core

• Principles of (Good) Design

State Machines

• Principles of (Good) Design

Hierarchical State Machines

• Principles of (Good) Design

UML State Machines

• Principles of (Good) Design

CFA vs. Software
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.

"Tell Them What You've Told Them"

(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


