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

VL 10
- **Introduction and Vocabulary**
- **Software Modelling**
  - model; views / viewpoints; 4+1 view
- **Modelling structure**

VL 11
  - (simplified) Class & Object diagrams
  - (simplified) Object Constraint Logic (OCL)

VL 12
- **Modelling behaviour**
  - Communicating Finite Automata (CFA)
  - Uppaal query language

VL 13
  - 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

VL 14
  - information hiding and data encapsulation
  - abstract data types, object orientation
- **Design Patterns**

Vocabulary

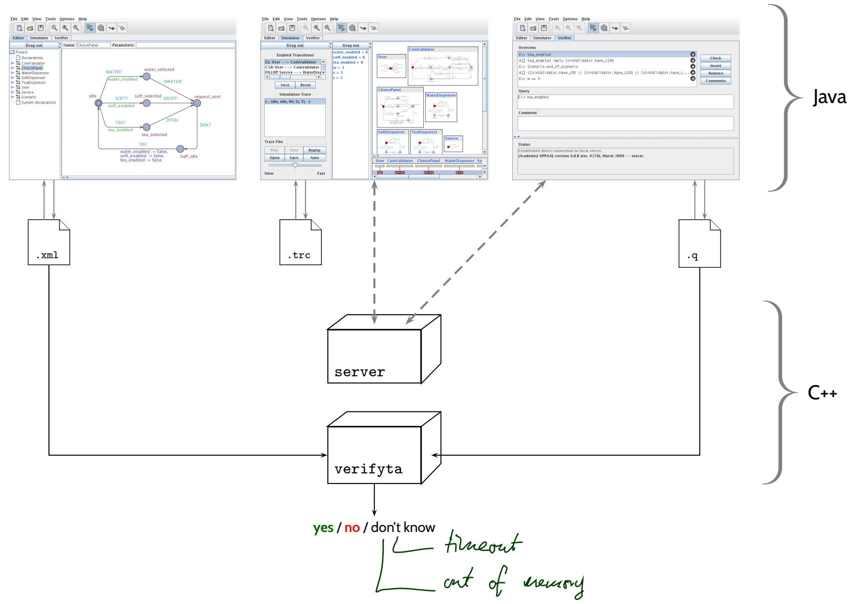
Techniques

informal ↕ semi-formal ↕ formal

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

What Can We Conclude From Verification Results?

- Assume that query Q correponds 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?

Tippfehler beim Modellieren
Fehler im Tool
does-not-sat not visible in $\mathcal{N}$

the design idea	tool result: $\mathcal{N} \models Q$	tool result: $\mathcal{N} \not\models Q$
does not sat. Q	false negative	true positive ✓
sat. Q	true negative ✓	false positive ✓

$\rightarrow$ **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,

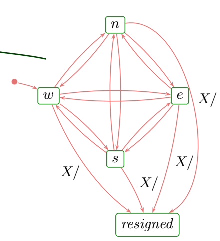
then the system **will not** fail **due to** an **analysable** design flaw.

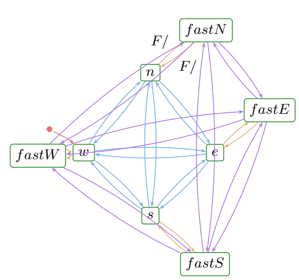
Content

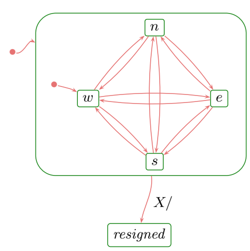
UML State Machines

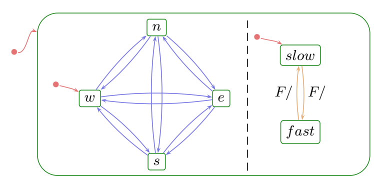
Composite (or Hierarchical) States

- OR-states, AND-states Harel (1987).
- Composite states are about **abbreviation**, **structuring**, and **avoiding redundancy**.

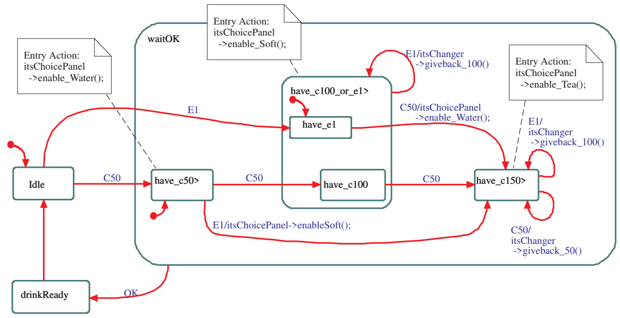


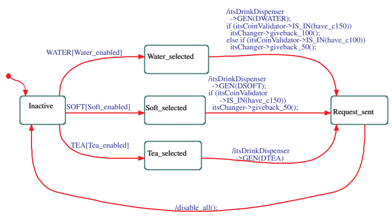


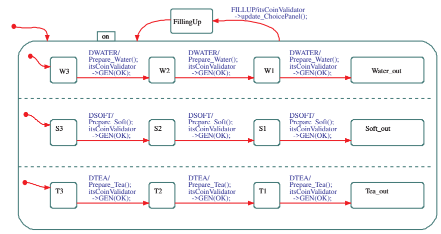




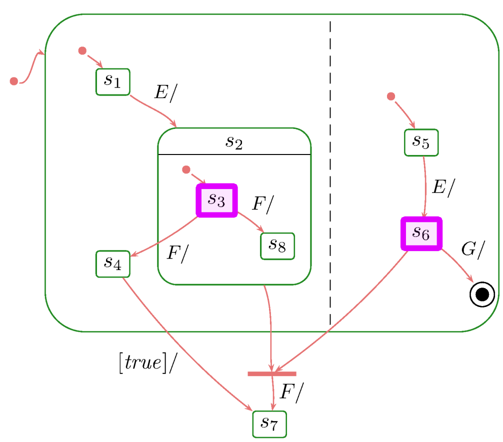
Example







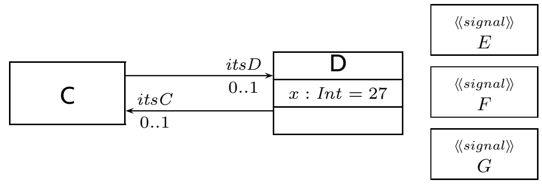
And That Would be Too Easy. . .

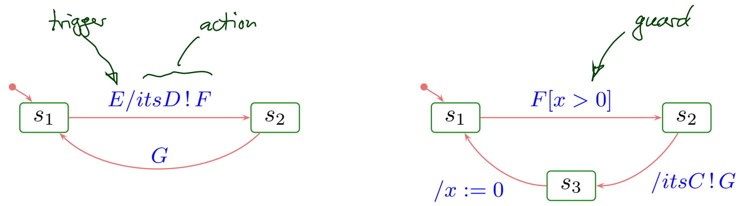


If an F is available, which edges are **enabled**? What are the possible **successor configurations**?

(The full story: "**Software Design, Modelling, and Analysis with UML**" (in some winter semesters).)

UML Core State Machines



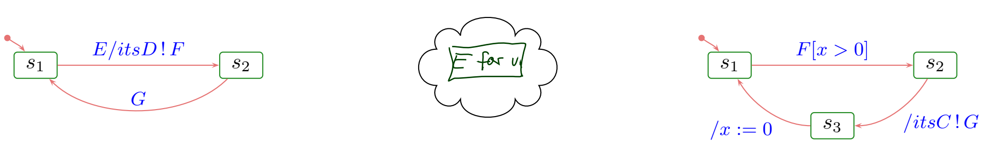


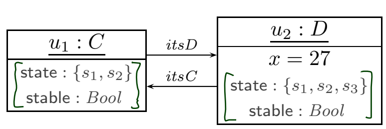
$$annot ::= \big[\ \underbrace{\langle event\rangle[\,\textbf{.}\,\langle event\rangle]^*}_{trigger}\ \big]\ \ [\,\textbf{[}\,\langle guard\rangle\,\textbf{]}\,]\ \ [\,\textbf{/}\,\langle action\rangle]\ \big]$$

with

- $event \in \mathcal{E}$, (optional)
- $guard \in Expr_{\mathscr{S}}$ (default: *true*, assumed to be in $Expr_{\mathscr{S}}$)
- $action \in Act_{\mathscr{S}}$ (default: `skip`, assumed to be in $Act_{\mathscr{S}}$)

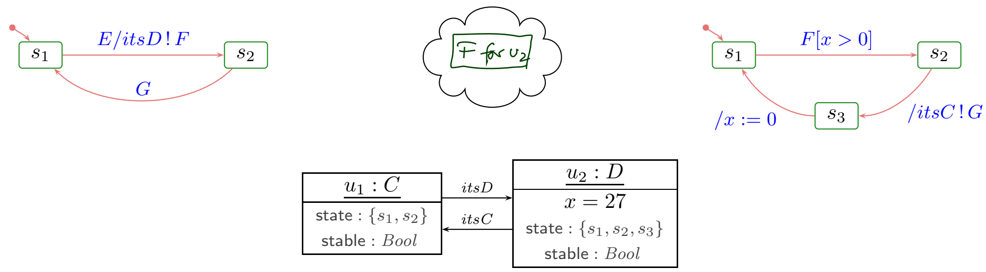
Event Pool and Run-To-Completion





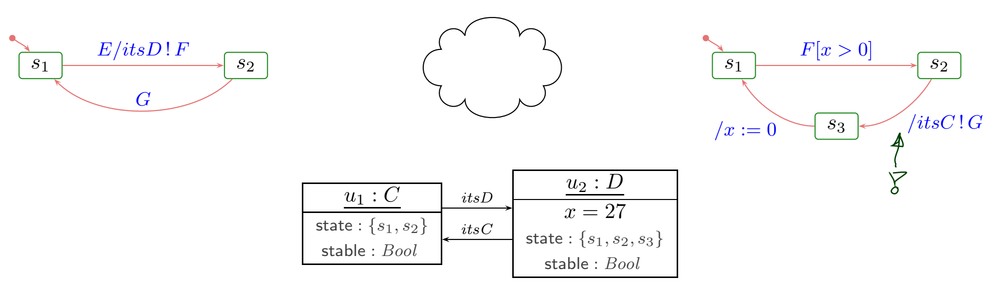
	u_1		u_2			
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1

Event Pool and Run-To-Completion



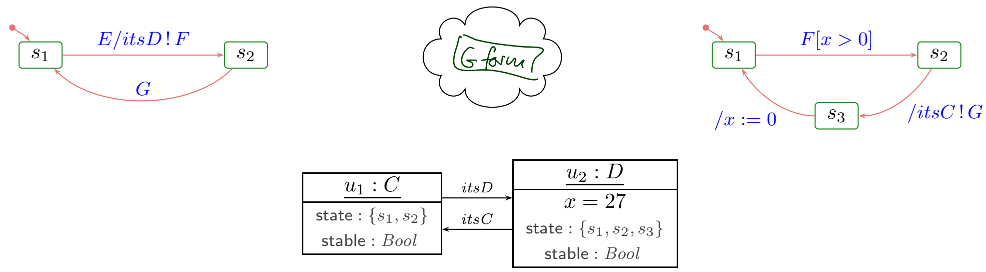
	u_1		u_2			
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2

Event Pool and Run-To-Completion



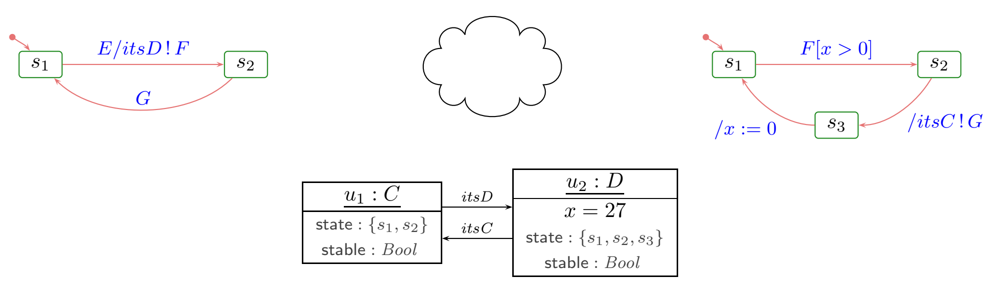
	u_1		u_2			
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	

Event Pool and Run-To-Completion



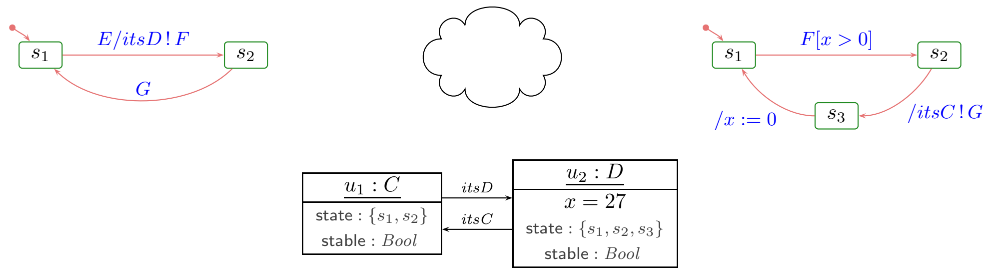
step	u_1 state	u_1 stable	x	u_2 state	u_2 stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1

Event Pool and Run-To-Completion



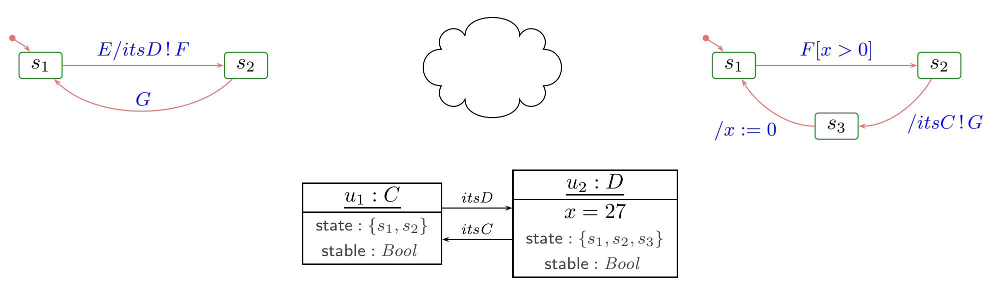
step	u_1 state	u_1 stable	x	u_2 state	u_2 stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1
4.a	s_2	1	0	s_1	1	G ready for u_1

Event Pool and Run-To-Completion



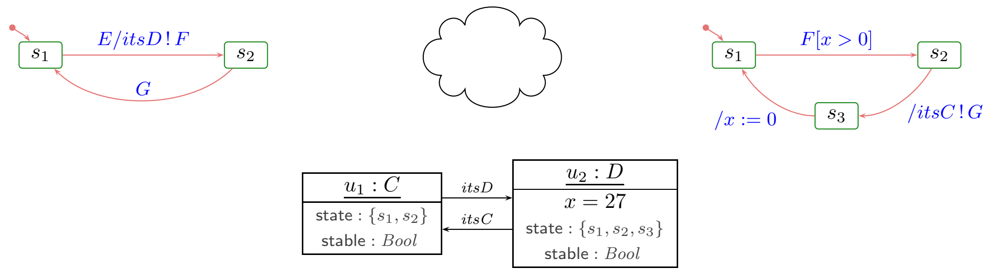
step	u_1 state	u_1 stable	x	u_2 state	u_2 stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1
4.a	s_2	1	0	s_1	1	G ready for u_1
5.a	s_1	1	0	s_1	1	

Event Pool and Run-To-Completion



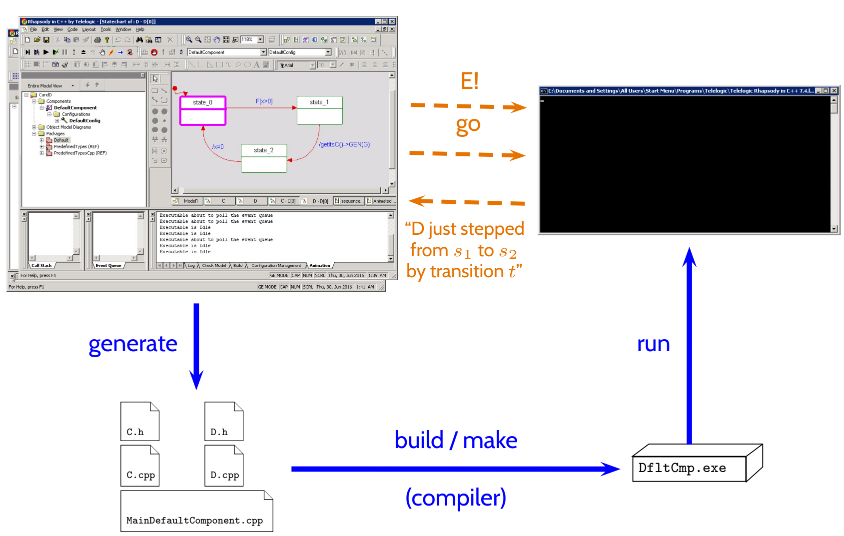
step	u_1 state	u_1 stable	x	u_2 state	u_2 stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1
4.a	s_2	1	0	s_1	1	G ready for u_1
5.a	s_1	1	0	s_1	1	
4.b	s_1	1	27	s_3	0	

Event Pool and Run-To-Completion



	u_1			u_2		
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1
4.a	s_2	1	0	s_1	1	G ready for u_1
5.a	s_1	1	0	s_1	1	
4.b	s_1	1	27	s_3	0	
5.b	s_1	1	0	s_1	1	

Rhapsody Architecture



Unified Modelling Language (UML)

UML Overview (OMG, 2007, 684)

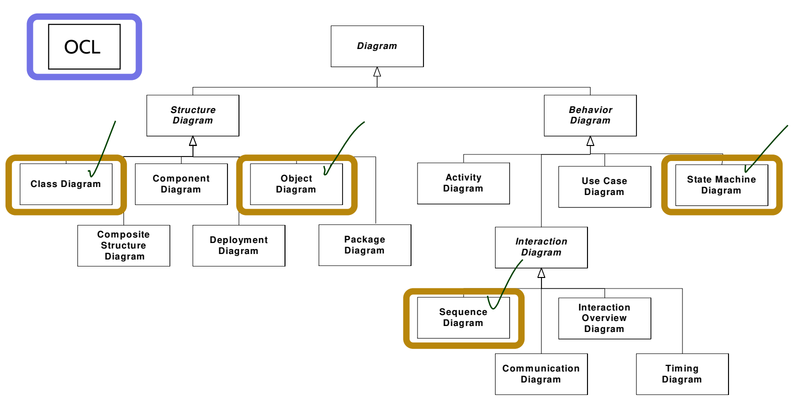


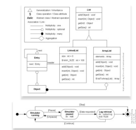
Figure A.5 - The taxonomy of structure and behavior diagram

Dobing and Parsons (2006)

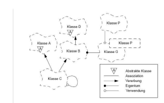
A Brief History of UML

- stone age — visualise software with boxes, circles, arrows, automata, etc.
- 'software crisis', term 'software engineering'
- 1970
- modelling languages: **Flowcharts**, **Nassi-Shneiderman**, **Entity-Relation Diagrams**, etc.
- 1980
- **Statecharts** (Harel, 1987), **StateMate** (Harel et al., 1990)
- 1990
- Object-Oriented Analysis/Design/ Programming,

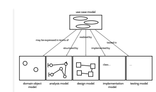
  Object-Modeling Technique (OMT) (Rumbaugh et al., 1990)



  Booch Method and Notation (Booch, 1993)



  Object-Oriented Software Engineering (OOSE) (Jacobson et al., 1992)



- **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;
- 2010

The UML standard is published by the **Object Management Group** (OMG):

*"international, open membership, not-for-profit **computer industry** consortium".*

– 13 – 2019-07-04 – Sumloutlook –

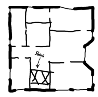
"UML Mode" [http://martinfowler.com/bliki]

> *"[...] 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."*

- Aplies to **UML as such** (as a language),
- and to each individual **UML model**.

– 13 – 2019-07-04 – Sumlmode –

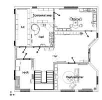
UML-Mode of the Lecture: As Blueprint



Sketch

In this UmlMode developers use the UML to help communicate some aspects of a system. [...]
Sketches are also useful in documents, in which case the focus is communication ra- ther than completeness. [...]
The tools used for sketching are lightweight drawing tools and often people aren't too particular about keeping to every strict rule of the UML.
Most UML diagrams shown in books, such as mine, are sketches.
Their emphasis is on selective communication rather than complete specification.

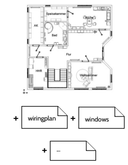
Hence my sound-bite "comprehensive-ness is the enemy of comprehensibility"



Blueprint

[...] In forward engineering the idea is that blueprints are developed by a designer whose job is to build a detailed design for a programmer to code up.
That design should be sufficiently complete that all design decisions are laid out and the programming 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. [...]

Forward engineering tools support dia-gram drawing and back it up with a repos-itory to hold the information. [...]



ProgrammingLanguage

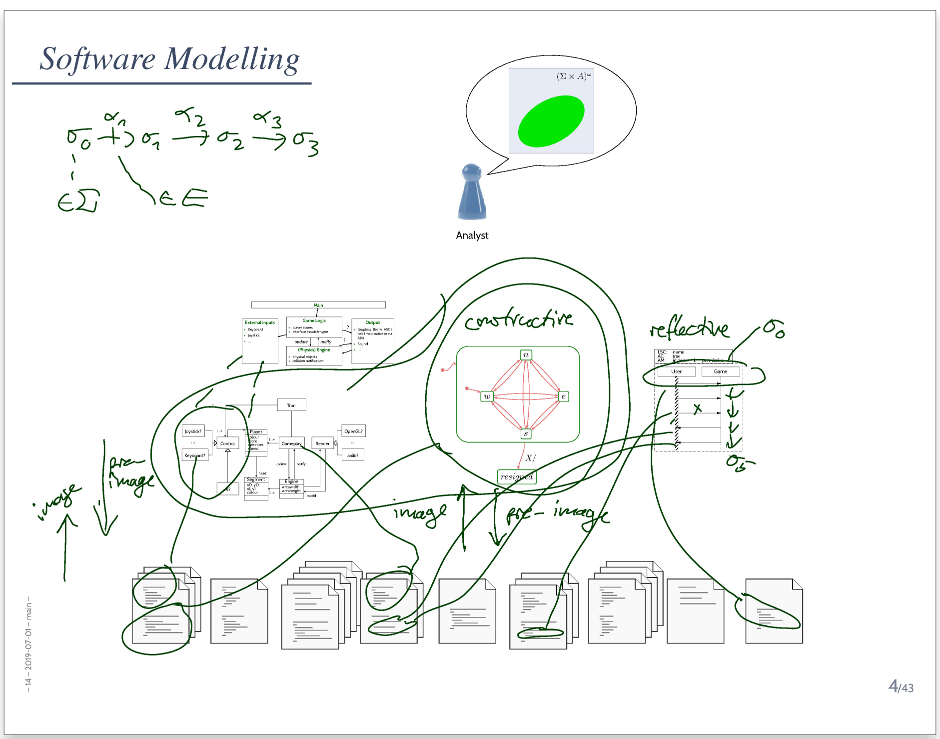
If you can detail the UML enough, and provide semantics for everything you need in software, you can make the UML be your programming language.
Tools can take the UML diagrams you draw and compile them into executable code.
The promise of this is that UML is a higher level language and thus more productive than current programming languages.
The question, of course, is whether this promise is true.

I don't believe that graphical program-ming will succeed just because it's graphi-cal. [...]

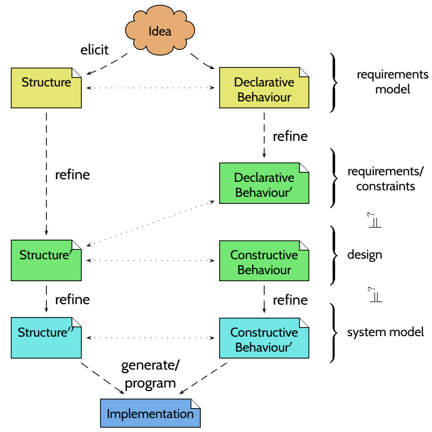
Content

- **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

Model-based/-driven Software Engineering



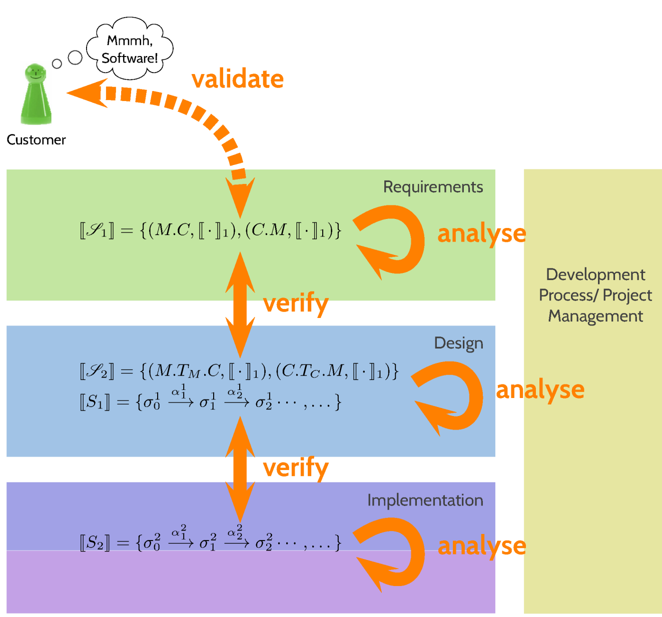
Model-Driven Software Engineering



- (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**.

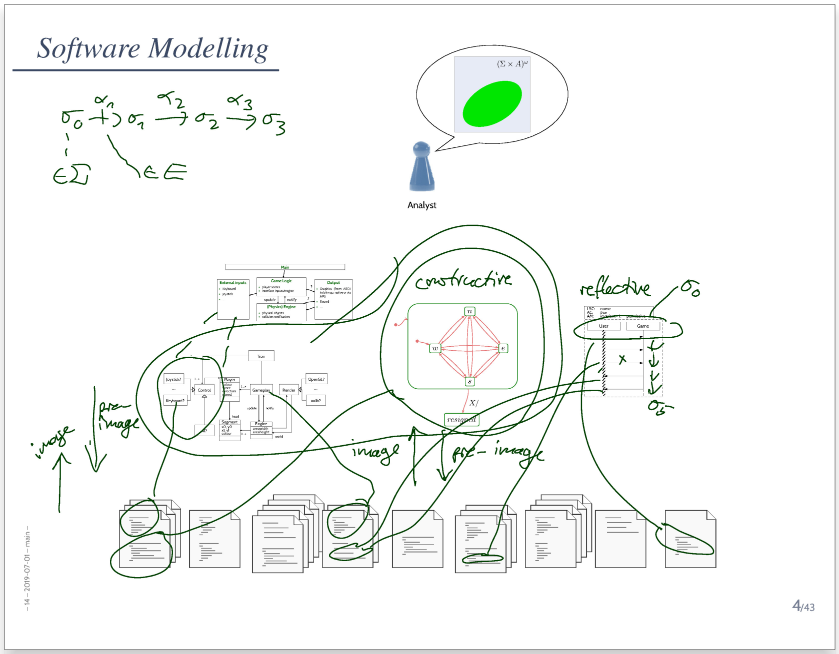
– 13 – 2019-07-04 – Smdse –

Formal Methods in the Software Development Process



– 13 – 2019-07-04 – Smdse –

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**. → VL 15
- Vocabulary: **Model-based/-driven Software Engineering**

Topic Area Architecture & Design: Content

VL 10
- **Introduction and Vocabulary**
- **Software Modelling**
  - model; views / viewpoints; 4+1 view

VL 11
- **Modelling structure**
  - (simplified) Class & Object diagrams
  - (simplified) Object Constraint Logic (OCL)

VL 12
- **Modelling behaviour**
  - Communicating Finite Automata (CFA)
  - Uppaal query language

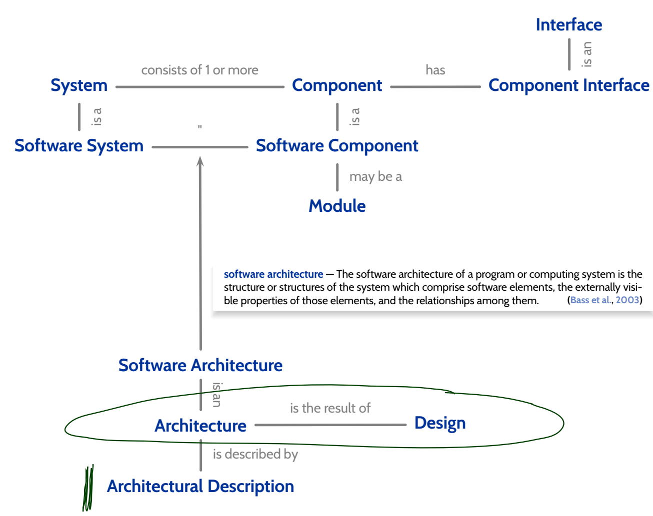
VL 13
  - CFA vs. Software
  - Unified Modelling Language (UML)
    - basic state-machines
    - an outlook on hierarchical state-machines
- **Model-driven/-based Software Engineering**
- **Principles of Design**
  - modularity, separation of concerns
  - information hiding and data encapsulation

VL 14
  - abstract data types, object orientation
- **Design Patterns**

Vocabulary

Techniques: informal ↕ semi-formal ↕ formal

Once Again, Please



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**.

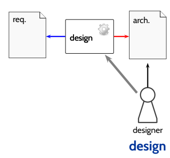
Design…

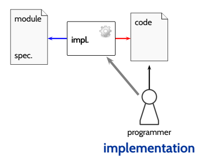
(i) **structures** a system into **manageable** units (yields software architecture),

(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

Principles of (Architectural) Design

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,

    Example: logical flow of (logical) messages in a communication protocol (**functional**) vs. exchange of (physical) messages using a certain technology (**technical**).
  - 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) $\rightarrow$ 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

References

Booch, G. (1993). *Object-oriented Analysis and Design with Applications*. Prentice-Hall.

Dobing, B. and Parsons, J. (2006). How UML is used. *Communications of the ACM*, 49(5):109–114.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

Harel, D., Lachover, H., et al. (1990). Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414.

IEEE (1990). *IEEE Standard Glossary of Software Engineering Terminology*. Std 610.12-1990.

Jacobson, I., Christerson, M., and Jonsson, P. (1992). *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley.

Ludewig, J. and Lichter, H. (2013). *Software Engineering*. dpunkt.verlag, 3. edition.

Nagl, M. (1990). *Softwaretechnik: Methodisches Programmieren im Großen*. Springer-Verlag.

OMG (2007). Unified modeling language: Superstructure, version 2.1.2. Technical Report formal/07-11-02.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1990). *Object-Oriented Modeling and Design*. Prentice Hall.