Softwaretechnik / Software-Engineering
Lecture 15: UML State Machines
& Software Quality Assurance

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)
- Principles of Design
  - modularity, separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
- Design Patterns
- Modelling behaviour
  - communicating finite automata (CFA)
  - Uppaal query language
- CFA vs. Software
- Model-driven/-based Software Engineering
  - Unified Modelling Language (UML)
  - basic and hierarchical state-machines

CFA vs. Software

A CFA 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 $\sigma_0 \alpha_1 \rightarrow \sigma_1 \alpha_2 \rightarrow \sigma_2 \cdot \cdot \cdot$ 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 $C(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ be a network of CFA.
- $\Sigma = \text{Conf}$
- $A = \text{Act}$
- $\llbracket C \rrbracket = \{ \pi = \langle \vec{\ell}_0, \nu_0 \rangle \lambda_1 \rightarrow \langle \vec{\ell}_1, \nu_1 \rangle \lambda_2 \rightarrow \langle \vec{\ell}_2, \nu_2 \rangle \lambda_3 \rightarrow \cdot \cdot \cdot \mid \pi \text{ is a computation path of } C \}$.

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

Example: Software Specification

Alphabet:
- $M$ – dispense cash only,
- $C$ – return card only,
- $M C$ – dispense cash and return card.

Customer: “I don’t care about the order of $M$ and $C$”

$S_1 = (M.C \mid \mid \mid C.M \mid \mid \mid M C)$

Refined Specification: “be consistent: either always $M.C$ or always $C.M$”

$S_2 = (M.C)^\omega$ or $(C.M)^\omega$

Design Idea: “consider human errors: always do $C.M$”

$S_1 = (C.M)^\omega$

Implementation (goal): software $S_2$ behaves according to the design idea.
Example: Vending Machine — Model Architecture

- CoinValidator
- User
- ChoicePanel
- WaterDispenser
- SoftDispenser
- TeaDispenser
- Service

C50, E1
WATER, SOFT, TEA
OK

• Shared variables:
  • bool water_enabled, soft_enabled, tea_enabled;
  • int w = 3, s = 3, t = 3;

Note: Our model does not use scopes ("information hiding") for channels. That is, 'Service' could send 'WATER' if the modeler wanted to.

Design Sanity Check: Drive to Configuration

• 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 \( N_{VM} | = \exists \diamond w = 0 \).

Design Check: Scenarios

• Question: Is the following existential LSC satisfied by the model? (Otherwise, the design is definitely broken.)

LSC: buy tea
AC: true
AM: initial
I: permissive

User
Coin Validator
Choice Panel
C50
C50
C50
TEA
¬E1
!

• Approach: Use the following newly created CFA 'Scenario 'end_of_scenario

TEA!
C50!
C50!
C50!

instead of User and check whether location end_of_scenario is reachable, i.e. check \( N'_{VM} | = \exists \diamond \text{Scenario } \end{of_scenario}.\)

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 \( N_{VM} | = \forall \square \text{tea } _{\text{enabled}} \text{ imply Coin Validator. have_c150} \) for the vending machine model \( N_{VM} \).
Design Verification: Sanity Check

• Question: Is the "tea" button ever enabled? (Otherwise, the considered invariant tea\_enabled = \Rightarrow Coin Validator have_c150 holds vacuously.)

• Approach: Check whether a configuration satisfying water\_enabled = 1 is reachable. Exactly like we did with w\_enabled = 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 \exists VM | \forall □ (Coin Validator have_c50 or Coin Validator have_c100 or Coin Validator have_c150) imply water\_enabled.

Recall: Universal LSC Example

LSC: buy water
AC: true
AM: invariant
I: strict
User CoinValidator ChoicePanel Dispenser
C 50 pWATER
¬ (C50! ∨ E1! ∨ pSOFT! ∨ pTEA! ∨ pFILLUP!)
water\_in\_stock

¬ (dSoft! ∨ dTEA!)

CFA vs. Software
• a CFA model is software
CFA at Work
• drive to configuration, scenarios, invariants
• tool demo (verifier).

Model-based/-driven Software Engineering
• Unified Modelling Language
• Brief History
• Sub-Languages
• UML Modes
• UML State Machines
• Hierarchical State Machines
• Core State Machines
• steps and run-to-completion steps
• Rhapsody

Content I (Architecture & Design)
• (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.

Development Approaches

- top-down risk: needed functionality hard to realise on target platform.
- bottom-up risk: lower-level units do not “fit together”.
- inside-out risk: user interface needed by customer hard to realise with existing system,
- outside-in risk: elegant system design not reflected nicely in (already fixed) UI.

Transform vs. Write-Down-and-Check

Content I (Architecture & Design)

CFA vs. Software

CFA at Work

drive to configuration, scenarios, invariants

tool demo (verifier).

Model-based/-driven Software Engineering

Unified Modelling Language

Brief History

Sub-Languages

UML Modes

UML State Machines

Hierarchical State Machines

Core State Machines

steps and run-to-completion steps

Rhapsody

Code Generation from CFA: A Simple Example

\[
\text{W0}\quad \text{dispense}\quad \text{Wi} \quad FILLUP? \\
\quad w := 3 \quad FILLUP? \quad w := 3 \quad w == 0 \quad \text{DOK!} \\
\quad w > 0 \quad \text{DOK!} \quad \text{DWATER?} \quad w := w - 1 \quad \text{int } w := 3; \\
\quad \text{typdef} \quad \text{st}_T \quad \text{st}_T := \text{Wi}; \\
\quad \text{Set } \langle \text{Act} \rangle \quad \text{take}\_\text{action}(\text{Act }\alpha) \quad \{ \\
\quad \text{Set } \langle \text{Act} \rangle \quad R := \emptyset; \quad \text{if } \□ \text{st}=\text{Wi}: \quad \text{if } \□ \alpha = \text{DWATER}?: \quad w := w - 1; \quad \text{st} := \text{dispense}; \quad \text{if } (w = 0) \quad R := R \cup \{\text{DOK!}\}; \quad \text{if } (w > 0) \quad R := R \cup \{\text{DOK!}\}; \quad \text{□ }\alpha = \text{FILLUP}?: \quad w := 3; \quad \text{st} := \text{Wi}; \quad R := R \cup \{\text{DWATER}?, \text{FILLUP}?\}; \quad \text{□ }\text{else} : \quad R := R \cup \{\text{DWATER}?, \text{FILLUP}?\}; \quad \text{□ }\text{fi}; \quad \text{□ }\text{st}=\text{dispense}: \quad \text{if } \□ \alpha = \text{DOK! }\land w = 0: \quad \text{st} := \text{W0}; \quad R := R \cup \{\text{FILLUP}?\}; \quad \text{□ }\alpha = \text{DOK! }\land w > 0: \quad \text{st} := \text{Wi}; \quad R := R \cup \{\text{FILLUP}?\}; \quad \text{□ }\text{else} : \quad R := R \cup \{\text{DOK}?\}; \quad \text{□ }\text{fi}; \quad \text{□ }\text{st}=\text{W0}: \quad \text{if } \□ \alpha = \text{FILLUP}?: \quad w := 3; \quad \text{st} := \text{Wi}; \quad R := R \cup \{\text{DWATER}?, \text{FILLUP}?\}; \quad \text{□ }\text{else} : \quad R := R \cup \{\text{FILLUP}?\}; \quad \text{□ }\text{fi}; \quad \text{□ }\text{fi} \}; \\
\quad \text{return } R; \quad \}
\]
as far as possible.

In this UmlMode developers use the UML to help communicate some aspects of a system. [\ldots] Sketches are also useful in rather than complete specification. Hence my sound-bite "comprehensiveness is the enemy of comprehensibility." 

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

In forward engineering the idea is that blueprints are developed by a designer whose job is to build a ... engineering tools support diagram drawing and back it up with a repository to hold the information. [\ldots]
Event Pool and Run-To-Completion

- \( s_1 \)
- \( s_2 \)

\[ \text{itsD} \]

\( \text{itsC} \)

\[ x > 0 \]

\( \text{u1} : \text{C} \)

state:

\( \{ s_1, s_2 \} \)

stable:

\( \text{Bool} \)

\( \text{u2} : \text{D} \)

state:

\( \{ s_1, s_2, s_3 \} \)

stable:

\( \text{Bool} \)

\( \text{u1} \)

\( \text{u2} \)

step state stable

\( x \)

state stable

event pool

0

\( \text{E} \)

\( \text{ready for} \)

\( \text{u1} \)

1

\( \text{F} \)

\( \text{ready for} \)

\( \text{u2} \)

2

\( \text{G} \)

\( \text{ready for} \)

\( \text{u1} \)

3

\( \text{G} \)

\( \text{ready for} \)

\( \text{u1} \)

4.a

\( \text{G} \)

\( \text{ready for} \)

\( \text{u1} \)

5.a

\( \text{G} \)

\( \text{ready for} \)

\( \text{u1} \)

5.b

\( \text{Rhapsody Architecture} \)

C.h

D.h

C.cpp

D.cpp

MainDefaultComponent.cpp

DfltCmp.exe

generate

build / make

(compiler)

run

\( \text{E!} \)

\( \text{go} \)

\( \text{D just stepped from} \)

\( s_1 \)

\( \text{to} \)

\( s_2 \)

by transition \( t \)

Content I (Architecture & Design)

- CFA vs. Software
- a CFA model is software
- CFA at Work
- drive to configuration, scenarios, invariants
- tool demo (verifier).
- Model-based/-driven Software Engineering
- Unified Modelling Language
- Brief History
- Sub-Languages
- UML Modes
- UML State Machines
- Hierarchical State Machines
- Core State Machines
- steps and run-to-completion steps
- Rhapsody

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 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 anymore.)

Mind UML Modes.

Code Quality Assurance

- Introduction and Vocabulary
- Test case, test suite, test execution.
- Positive and negative outcomes.
- Limits of Software Testing
- Glass-Box Testing
- Statement-, branch-, term- coverage.
- Other Approaches
- Model-based testing
- Runtime verification.
- Program Verification
- partial and total correctness
- Proof System PD

Review
in part, i.e., the states of the form

\[ \sigma \rightarrow \cdots \rightarrow \alpha \]

After 10 steps the latest, "expect WATER and any time later."

Shorthand notation

\[ \text{strlen} = \text{Soll}(\Sigma) \subseteq S \]

In a description, what it is supposed to do. "Testing is the execution of a program with the goal to discover errors."

Definition.
experiment means systematic test; if not systematic, call it test:

In the following for non-systematic tests: Experiment, 'Rumprobieren'.

(Our) Synonyms

Same case: power outage (if continuous power supply is considered in input sequence).

• is neither positive nor negative (only defined for test executions).

π2013, Ludewig and Lichter

a test execution.

is not π, T

Thus

If a tester does not adhere to an allowed input sequence of

• is neither positive nor negative (only defined for test executions).

π

Thus

Note

— a test such that

Systematic Test

results are documented and assessed according to criteriathat have been fixed before.

• (environment) conditions are defined or precisely documented,

• — (one or multiple) execution(s) of a program on a computer with the goal to find

T est

— (one or multiple) execution(s) of a program on a computer with the goal to find

T est

• (other goal)

metrics

of the program with a debugger

• (other goal)

analysis by software-tools for, e.g., values of

• (other goal)

• (other goal)

of the program

Sie (other goal)

• (other goal)

of the program

— (one or multiple) execution(s) of a program on a computer with the goal to find

T est

• (other goal)

metrics

of the program with a debugger

• (other goal)

analysis by software-tools for, e.g., values of

• (other goal)

• (other goal)

of the program

— (one or multiple) execution(s) of a program on a computer with the goal to find

T est

• (other goal)

metrics

of the program with a debugger

• (other goal)

analysis by software-tools for, e.g., values of

• (other goal)

• (other goal)

of the program

— (one or multiple) execution(s) of a program on a computer with the goal to find

T est

• (other goal)

metrics

of the program with a debugger

• (other goal)

analysis by software-tools for, e.g., values of

• (other goal)

• (other goal)

of the program

— (one or multiple) execution(s) of a program on a computer with the goal to find

T est

• (other goal)

metrics

of the program with a debugger

• (other goal)

analysis by software-tools for, e.g., values of

• (other goal)

• (other goal)

of the program

— (one or multiple) execution(s) of a program on a computer with the goal to find

T est

• (other goal)

metrics

of the program with a debugger

• (other goal)

analysis by software-tools for, e.g., values of

• (other goal)

• (other goal)

of the program

— (one or multiple) execution(s) of a program on a computer with the goal to find

T est

• (other goal)

metrics

of the program with a debugger

• (other goal)

analysis by software-tools for, e.g., values of

• (other goal)

• (other goal)

of the program

— (one or multiple) execution(s) of a program on a computer with the goal to find

T est

• (other goal)

metrics

of the program with a debugger

• (other goal)

analysis by software-tools for, e.g., values of

• (other goal)

• (other goal)

of the program

— (one or multiple) execution(s) of a program on a computer with the goal to find

T est

• (other goal)

metrics

of the program with a debugger

• (other goal)

analysis by software-tools for, e.g., values of

• (other goal)

• (other goal)

of the program

— (one or multiple) execution(s) of a program on a computer with the goal to find

T est

• (other goal)

metrics

of the program with a debugger

• (other goal)

analysis by software-tools for, e.g., values of

• (other goal)

• (other goal)

of the program

— (one or multiple) execution(s) of a program on a computer with the goal to find

T est

• (other goal)

metrics

of the program with a debugger

• (other goal)

analysis by software-tools for, e.g., values of

• (other goal)

• (other goal)

of the program
Strictly speaking, a test case is a triple \((In, Soll, Env)\) comprising a description \(Env\) of (environmental) conditions. \(Env\) describes any aspects which could have an effect on the outcome of a test execution and cannot be specified as part of \(In\), such as:

- Which program (version) is tested?
- Built with which compiler, linker, etc.?
- Test host (OS, architecture, memory size, connected devices (configuration?), etc.)?
- Which other software (in which version, configuration) is involved?
- Who is supposed to test when?
- etc. etc.

→ Test executions should be (as) reproducible and objective (as possible).

Full reproducibility is hardly possible in practice—obviously (err, why...?).

Steps towards reproducibility and objectivity:
- have a fixed build environment,
- use a fixed test host which does not do any other jobs,
- execute test cases automatically (test scripts).

Recall:

Quotes on Testing

"Software testing can be used to show the presence of bugs, but never to show their absence!"

(E. W. Dijkstra, 1970)

Why Can't We Show The Absence of Errors (in General)?

Consider a simple pocket calculator for adding 8-digit decimals:

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
+ & 7 & 8 & 9 & 0 & 4 & 5 & 6 \\
\hline
1 & 2 & 3 & \text{E-} \\
\end{array}
\]

- Requirement: If the display shows \(x\), \(+\), and \(y\), then after pressing \(=\),
  - the sum of \(x\) and \(y\) is displayed if \(x + y\) has at most 8 digits,
  - otherwise "-E-" is displayed.

- With 8 digits, both \(x\) and \(y\) range over \([0, 10^8 - 1]\).

- Thus there are \(10^{16} = 10,000,000,000,000\) possible input pairs \((x, y)\) to be considered for exhaustive testing, i.e. testing every possible case!

- And if we restart the pocket calculator for each test, we do not know anything about problems with sequences of inputs...
  (Local variables may not be re-initialised properly, for example.)
Observation: Software Usually Has Many Inputs

Example: Simple Pocket Calculator.

With ten thousand (10,000) different test cases (that's a lot!), 9,999,999,999,990,000 of the 10^16 possible inputs remain uncovered. In other words: Only 0.0000000001% of the possible inputs are covered, 99.99999999% not touched.

In diagrams (red: uncovered, blue: covered):

Point vs. Range Errors

Software is (in general) not continuous.

Consider a continuous function, e.g. the one to the right:

For sufficiently small \( \varepsilon \)-environments of an input, the outputs differ only by a small amount \( \delta \).

Physical systems are (to a certain extent) continuous:

• For example, if a bridge endures a single car of 1000 kg, we strongly expect the bridge to endure cars of 990 kg or 1010 kg.
• And anything of weight smaller than 1000 kg can be expected to be endured.
• For software, adjacent inputs may yield arbitrarily distant output values.

Vocabulary:

• Point error: an isolated input value triggers the error.
• Range error: multiple "neighbouring" inputs trigger the error.

For software, (in general, without extra information) we cannot conclude from some values to others.

Content (Part II)

Introduction

• quotes on testing,
• systematic testing vs. 'rumprobieren'.

Test Case

• definition,
• execution,
• positive and negative.

Test Suite

• Limits of Software Testing

• Software examination paths

• Is exhaustive testing feasible?
• Range vs. point errors

• More Vocabulary

• Tell Them What You've Told Them.

• Testing is about finding errors, or demonstrating scenarios.

• A test case consists of input sequences and expected outcome(s).

• A test case execution is positive if an error is found, negative if no error is found.

• A test suite is a set of test cases.

Distinguish (among others),

• glass-box test: structure (or source code) of test item available,
• black-box test: structure not available.

References


