The Satisfaction of Uppaal Queries by Configurations

...
Composite (or Hierarchical) States

OR-states, AND-states

Event Pool and Run-to-Completion

Event Pool and Run-to-Completion

Avoiding redundancy, and structuring, abbreviation...
Would be Too Easy.

• s1
• s2
• s3
s8
• s4
s5
s6
E/
F/
F/
E/
G/

→ "Software Design, Modelling, and Analysis with UML" in the winter semester.

Rhapsody Architecture

UML Modes

And the Pragmatic Attribute

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:

+wiringplan
+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.
With UML it's the Same

[http://martinfowler.com/bliki]

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 ... communication rather than 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 ... engineering tool supports diagram drawing and back it up with a repository to hold the information. ([..]

ProgrammingLanguage

If you can detail the UML enough, and provides semantics for everything you need in software, you can make the UML be ... is whether this promise is true. ([..]

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.

Architecture Patterns

Introduction

• Over decades of software engineering, many clever, proved and tested designs of solutions for particular problems emerged.

• Question: can we generalise, document and re-use these designs?

• Goal: “don’t re-invent the wheel” / benefit from “clever”, “proven and tested”, “solution”.

architectural pattern — An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Buschmann et al. (1996)

• Using an architectural pattern implies certain characteristics or properties of the software (construction, extendibility, communication, dependencies, etc.),

• determines structures on a high level of the architecture, thus is typically a central and fundamental design decision.

• The information that (where, how, ...) a well-known architecture / design pattern is used in a given software can make comprehension and maintenance significantly easier.

Example: Layered Architectures

• (Züllighoven, 2005):

• A layer whose components only interact with components of their direct neighbour layers is called protocol-based layer. A protocol-based layer hides all layers beneath it and defines a protocol which is (only) used by the layers directly above.

• Example: The ISO/OSI reference model.

1. Physical
2. Data link
3. Network
4. Transport
5. Session
6. Presentation
7. Application

• object-oriented layer: interacts with layers directly and possibly further above and below.

• Rules: the components of a layer may use:
  • only components of the protocol-based layer directly beneath,
  • all components of layers further beneath.

• Examples: GNOME etc. Applications

• GTK+, GDK, ATK
• Cairo, GLib, GIO
• Pango

Example: Layered Architectures Cont’d

• (Züllighoven, 2005):

• A layer whose components only interact with components of their direct neighbour layers is called protocol-based layer. A protocol-based layer hides all layers beneath it and defines a protocol which is (only) used by the layers directly above.

• Example: The ISO/OSI reference model.

1. Physical
2. Data link
3. Network
4. Transport
5. Session
6. Presentation
7. Application

• object-oriented layer: interacts with layers directly and possibly further above and below.

• Rules: the components of a layer may use:
  • only components of the protocol-based layer directly beneath,
  • all components of layers further beneath.

• Examples: GNOME etc. Applications

• GTK+, GDK, ATK
• Cairo, GLib, GIO
• Pango
Example: Layered Architectures Cont'd
• object-oriented layer: interacts with layers directly and possibly further above and below.
• Rules: the components of a layer may use
  • only components of the protocol-based layer directly beneath,
  • all components of layers further beneath.

Example: Three-Tier Architecture
Desktop Host
presentation tier
Application Server
(business) logic tier
data tier
Database Server
DBMS

• presentation layer: user interface; presents information obtained from the logic layer to the user, controls interaction with the user, i.e. requests actions at the logic layer according to user inputs,
• logic layer: core system functionality; layer is designed without information about the presentation layer, may only read/write data according to data layer interface,
• data layer: persistent data storage; hides information about how data is organised, read, and written, offers particular chunks of information in a form useful for the logic layer.

Example: Model-View-Controller
controller
• sees change of
• uses

view
• sees
• uses

model
• sees
• uses

Advantages:
• one model can serve multiple view/controller pairs;
• view/controller pairs can be added and removed at runtime;
• model visualisation always up-to-date in all views;
• distributed implementation (more or less) easily.

Disadvantages:
• if the view needs a lot of data, updating the view can be inefficient.
In a sense the same as architectural patterns, but on a lower scale. Often traced back to (Alexander et al., 1977; Alexander, 1979).

Design patterns... are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. (Gamma et al., 1995)

Example: Strategy

**Problem**
The only difference between similar classes is that they solve the same problem by different algorithms.

**Solution**
- Have one class `StrategyContext` with all common operations.
- Another class `Strategy` provides signatures for all operations to be implemented differently.
- From `Strategy` derive one sub-class `ConcreteStrategy` for each implementation alternative.
- `StrategyContext` uses concrete Strategy-objects to execute the different implementations via delegation.

**Structure**
```
StrategyContext + contextInterface()
Strategy + algorithm()
ConcreteStrategy1 + algorithm()
ConcreteStrategy2 + algorithm()
```

Example: Pattern Usage and Documentation

```
Painter
SimpleUpdateStrategy
DrawingView
Tool
DrawingEditor
CreationTool
SelectionTool
Drawing
```

Example: Singleton and Memento

**Singleton**
Problem
Of one class, exactly one instance should exist in the system.
Example
Print spooler.

**Memento**
Problem
The state of an object needs to be archived in a way that allows re-constructing this state without violating the principle of data encapsulation.
Example
Undo mechanism.

Example: Mediator, Observer, and State

**Mediator**
Problem
Objects interacting in a complex way should only be loosely coupled and be easily exchangeable.
Example
Appearance and state of different means of interaction (menus, buttons, input fields) in a graphical user interface (GUI) should be consistent in each interaction state.

**Observer**
Problem
Multiple objects need to adjust their state if one particular other object is changed.
Example
All GUI object displaying a file system need to change if files are added or removed.

**State**
Problem
The behaviour of an object depends on its (internal) state.
Example
The effect of pressing the room ventilation button depends (among others?) on whether the ventilation is on or off.
For some application domains, there are
•
•
•
•
•
•
•
•
•

For example:

Libraries and Frameworks

Advantages
•

Disadvantages
•

In order to provide a consistent
treatment of interfaces, a rich
collection of patterns is needed,
which makes it difficult to
know which patterns to apply in
different contexts.

The development of design patterns
is considered to be one of the most
important innovations of software
engineering in recent years.

“Surprise, surprise” (software may
be easy — using design patterns
appropriately in new designs
requires experience).

Here: Understanding abstract
descriptions of design patterns
or their use in existing projects:

•

•

•

•

•

Classical
•

Meta Design Pattern: Inversion of Control

Examples

•

•

•

•

•

•

•

•

•

•

•

•


•

•


Functionality High
Interface with outside world High
Sequences of behaviour Medium
Passive object's attribute Medium
Long-lived information structures Low
Object from application [domain] Low

Note
- •
- •
- •
- •

Proposal (i) When a program is used (ii) When a program is modified

- •

Risk: lower-level units do not "fit together".

Risk: needed functionality hard to realise on target platform.

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: elegant system design not reflected nicely in (already fixed) UI.

Risk: risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,

Risk: user interface needed by customer hard to realise with existing system,