Designing Adaptive Software Systems

John Mylopoulos
University of Trento

Faculty of Informatics,
Univ. of Vienna, Vienna,
December 12, 2012
Abstract

Adaptive systems of any sort (software, hardware, biological or social) consist of a base system that carries out activities to fulfill some requirements $R$, and a feedback loop that monitors the performance of the system relative to $R$ and takes corrective action if necessary. We adopt this view of adaptivity for software-intensive systems and sketch a framework for designing adaptive systems which starts with requirements models, extends them to introduce control-theoretic concepts, and uses them at run-time to control the behaviour of the base system.

The presentation is based on joint research with Vitor Souza, Alexei Lapouchnian, Kostas Angelopoulos, all with the University of Trento.
Software Engineering (SE)

- Aims to develop concepts, tools and techniques for building software systems.
- Born in 1968 during a NATO workshop.
- Founded on the premise that software should be designed and built using engineering principles for design, testing, etc.
- Still in its formative years, compared to other engineering disciplines.
Software Engineering (SE)

- Aims to develop concepts, tools and techniques for building software systems.
- Born in 1968 during a NATO workshop.
- Founded on the premise that software should be designed and built using engineering principles for design, testing, etc.
- Still in its formative years, compared to other engineering disciplines.
Requirements Engineering (RE)

- Concerned with the elicitation, modelling and analysis of software requirements from stakeholders.
- researched and practiced since the mid-70s.
- Often credited with the introduction of models in SE.
The Requirements Problem (J&Z)

In its original formulation [Jackson95], a requirements problem consists of deriving a specification $S$ for a given set of requirements $R$ and indicative environment properties $E$ such that

$$E, S \models R$$

meaning: “... satisfaction of the requirements can be deduced from satisfaction of the specification, together with the environment properties...” [Jackson95]

We prefer a formulation where environment properties are replaced by domain assumptions (D) and inference is replaced by entailment

$$D, S \models R$$
Requirements as Goals (GORE)

Requirements are now goals and (requirements) problem solving amounts to incremental goal refinement.

Good quality schedule

Collection of timetables

Find free room

Choose schedule

Get free room

Collect

Rooms available

Schedule

Schedule meeting

Softgoal

Choice points

©2012 John Mylopoulos

Domain assumption

Task

UniVie'12 -- 7
Requirements as Goals

Here, specifications consist of tasks, domain assumptions and quality constraints that together satisfy requirements, e.g., for G:ScheduleMtg, one specification is \{T:Collect, T;Schedule, D:RoomsAv, QC: ‘>70% participation’\}

Unlike J&Z, goal refinement generates a space of possible specifications and the requirements problem amounts to finding those that satisfy R.

The GORE version of the requirements problem can be reduced to SAT solving [Sebastiani04].

These are design-time models: they tell us what are the requirements and possible functionalities through which the system-to-be can fulfilling these requirements.
Monitoring requirements

- FLEA+KAOS [Feather98], ReqMon [Robinson06], ...

Schedule meetings

Dynamic analysis

Vanilla RE

Specification

Feedback

Specification
What requirements lead to adaptivity functions?
Our task

Schedule meetings

Schedule meetings + ??

Feedback

Specification

Specification
Goal operationalizations

We use three types of operationalizations, two proactive, one *opportunistic*. 
Requirements + control parameters

- Collect timetables
  - By person
  - By system

- Get available rooms
  - OR

- Find rooms
  - OR

- Choose schedule
  - OR

- Schedule meeting
  - AND

- Good quality schedule

**FhM** - From how many?
**RfM** - # of Rooms for Meetings

©2012 John Mylopoulos
**Runtime requirements models**

- Runtime requirements models are supposed to keep track of how each instance of a design-time requirement (e.g., ‘Schedule meeting’) is doing.

- Accordingly, they need to capture additional information that design-time models don’t:
  - State of requirement instances, e.g. for goals: initiated, suspended, pursued, fulfilled, denied, …)
  - History: What percentage of instances of requirement R were denied between dates X and Y?
  - Behaviour: What are allowable instantiations of a requirements model?
Modeling state

We can use FSMs, such as the following one (for goals). Every goal instance can be in one of these states.
Specifying behaviour

This is a design-time model. What are its runtime instances?

... How about (CT CS)+ D? Possible instances include CT CS D, CT CS CT CS D, ...

We can use regular expressions to describe behaviour, but also Finite State Machines, Labelled Transition Systems, ...
Design-time vs runtime models

- Design-time models are intended to help us capture \textit{required functionality} for the system-to-be.
- Runtime models are intended to help us \textit{monitor behaviour} of the system and take corrective action, if necessary.
- We know how to (formally) reason with design-time models.
- How do we reason with runtime ones? For example, if we know that for an instance of SM, we have 2 instances of CT, both satisfied, and two instances of CS, one satisfied, the other pursued, what can we infer about the instance of SM?
- See [Morandini] for some results on this.
Specification space

From a requirements model \( M \) with \( n \) OR-refinements, we can derive \( O(2^n) \) specifications. Each alternative consists of making a choice for each OR-refinement. Control parameters add to this, possibly an infinite space.

One of these specifications is implemented by the running version of a system. We’ll refer to this specification as \textit{CurrentSpec}. 
Uncertainty

Operationalizations here (and in all GORE frameworks) basically mean: If tasks T can be carried out and quality constraints QC and domain assumptions DA hold, then requirements R can be fulfilled.

$$T, DA, QC |= R$$

That’s an optimistic view of the world …

✓ Rooms may not be available for meetings,
✓ Someone may not turn in his timetable, …

Executions of an operationalization may fail

→ uncertainty!
Adaptivity is all about coping/controlling uncertainty
Awareness

Awareness: Consciousness, sentience, ability to sense and respond to the environment.

Many types of awareness play a role in the design of software systems (security/process/context/location ... )

Important topic, not only in Computer Science.

In Philosophy, awareness plays a key role in several theories of consciousness. Our notion of awareness requirements draws on the distinction between higher-order awareness (awareness of one’s own mental states) and first-order awareness (awareness of things external) [Rosenthal05].
Awareness requirements

- Refer to other requirements (Goals/Tasks/Quality Constraints/Domain Assumptions) and their success/failure at runtime.
- Consider

  \[ r = \text{\'schedule meeting\'}, \quad da = \text{\'always rooms available\'} \]
  \[ r_1 = \text{\'r will be completed within 2hrs\'} \quad \text{(delta)} \]
  \[ r_2 = \text{\'r won\’t fail >3 times per year\'} \quad \text{(aggregate)} \]
  \[ r_3 = \text{\'avg r time won\’t increase between months\'} \quad \text{(trend)} \]
  \[ r_4 = \text{\'da won\’t fail >3 times per year\'} \]
Strata of awareness

Of course, one could have awareness requirements about other awareness requirements.

For example,

\[ r = \text{‘schedule meeting’}, \text{da = ‘always rooms available’} \]

\[ r1 = \text{‘r will be completed within 2hrs’} \]

\[ r1’ = \text{‘r1 won’t fail >3 times per year’} \]

To avoid paradoxes, we need to stratify our requirements space: base requirements are at stratum 0, awareness requirements about base requirements are at stratum 1, ...

In general, an awareness requirement at stratum n+1 talks about a requirement at stratum n, and can only talk about requirements at strata \( \leq n \).
Evolution requirements

Some requirements don’t just talk about other requirements, they also call for changing them.

Consider:

\[ r = \text{‘schedule meeting’}, \ r_1 = \text{‘r will be completed within 2hrs’} \]
\[ r_2 = \text{‘r will be completed within 3hrs’} \]
\[ er = \text{‘if r1 fails more than 3 times in a row, replace it with r2’} \]

Here the evolution requirement \( er \) calls for “relaxing” \( r_1 \) to \( r_2 \) if \( r_1 \) proves problematic.

\( er \) is clearly a requirement, i.e., something stakeholders will tell you during requirements elicitation.
Expressing evolution requirements

- We need some actions that can operate on runtime requirements models.
- Evolution requirements are then expressed as “if <awReq> then <action>”
- For example,
  
  \[\begin{align*}
  \text{dur}(r1.r.\text{collectTT}) & > 2\text{hrs} \Rightarrow \text{fail}(r1) \land \text{initiate}(r2) \quad \text{(relax)} \\
  2.2\text{hrs} & > \text{dur}(r1.r) > 2\text{hrs} \Rightarrow \text{fulfill}(r1) \quad \text{(good-enough)} \\
  \text{dur}(r1.r.\text{collectTT}) & > 3\text{hrs} \Rightarrow \text{fail}(r1) \land \text{fail}(r) \quad \text{(abort)} \\
  \text{failed}(r.\text{da}) & \Rightarrow \text{initiate}(\text{gmr}) \quad \text{(compensation)} \\
  \text{dur}(r1.r.\text{collectTT}) & > 2\text{hrs} \Rightarrow \text{changePar}(\text{FhM}:\text{FhM-20\%}) \quad \text{(compensation)}
  \end{align*}\]
Evolution reqs: Changing CurrentSpec

So far, examples of adaptivity requirements affect the status of requirement *instances*. But sometimes, we want adaptation to amount to a “from now on” change.

Suppose *CurrentSpec* assumes there are rooms available. If this DA fails more than 3 times within a month, we want to change choice point 2 for *CurrentSpec* and increase the # of rooms available for meetings (RfM):

\[
\text{size}(\text{select } x \text{ RA.inst st failed?}(x) \land \text{within}(x,1yr,\text{now}) > 3) \\
\Rightarrow \text{changespec}(\text{cp2} \neq 1, \text{RfM}: = \text{RfM}+10)
\]

It would be useful to have variations of *changespec* that (i) maximize familiarity in the new spec relative to the old one, or (ii) minimize new implementation needed for the adaptation, etc. (*many variations*) [Ernst]
Reactive Operationalizations

Sometimes, a goal is fulfilled by simply waiting for its fulfillment. More generally, for a goal $G$ consisting of $G_1 \land G_2$, we may choose to do nothing until $G_1$ or $G_2$ are true (… opportunistic operationalization).

Let’s postulate that each awareness requirement (on another requirement $r$) is to be operationalized reactively with a feedback loop consisting of monitor, diagnose, reconcile, compensate tasks.
Monitor-Diagnose-Reconcile-Compensate

Monitor – the environment; usually applies to tasks, quality constraints and domain assumptions, *not* to goals.

Diagnose – interpret monitored data; figure out the status of goals given information on the execution of tasks; see [Wang09], [Souza09] for principled ways of interpreting log data about tasks to offer a diagnosis about the failure of goals.

Reconcile – what’s the adaptation, at the instance/class level.

Compensate – change management, may include new implementation and/or adoption of results of aborted execution.
Operationalizations revisited

- We now have four types of operationalizations.
- The first two are proactive, one is opportunistic, the last reactive.

Awareness requirement

Monitor-Diagnose-Compensate-Execute
Where do adaptivity reqs come from?

The need for adaptivity comes from “upper level” NFRs, such as criticality and flexibility. These, in turn, can have their origins in safety, dependability, reliability, etc.

Consider: Meeting scheduling (MS) is a critical requirement for our organization; hence we allocate more resources to MS, we do more V&V for our MS system, AND we impose some awareness requirements for it as well ...

©2012 John Mylopoulos
So what?

Schedule meetings

Schedule meetings

Awareness requirements

Evolution requirements

Feedback

Specification

Specification
Conclusions

We have sketched a theory of software adaptivity founded on the concept of awareness.

The theory is based on a careful account of what awareness means in the context of requirements, how to express it, and how to operationalize it in terms of reactive plans.

The three major extensions required for GORE frameworks to account for that theory are: (i) Runtime requirements models, (ii) Introduction of control variables, (iii) A new reactive operationalization primitive.
References


References (cont’d)


References (cont’ d)

