Challenges of self-adaptive software

the fading boundary between
development time and run time

Internetware 2012 & ISHCS 2012, Qingdao (China)

Carlo Ghezzi
Politecnico di Milano
Deep-SE Group @ DEI-PoliMI
The vision

- **World fully populated by computationally rich devices offering services (disappearing computer)**
  - appliances, sensors/actuators, ... “things”
- **Cyber-physical systems**
- **Mobility**
- **Situation-aware computing**
  - new “services” built dynamically in a situation-dependent manner
- **Continuously running systems**
  - need to evolve while they offer service
The challenge

• Change and flexibility adversary of dependability

• Can they be reconciled through sound design methods?
The *machine* and the *world*

**World (the environment)**

- **Goals**
- **Requirements**

**Machine**

- **Domain properties (assumptions)**
- **Shared phenomena**
- **Specification**

Dependability arguments

• Assume that a rigorous (formal) representation is given for
  – R = requirements
  – S = specification
  – D = domain assumptions

if S and D are all satisfied and consistent, it is necessary to prove
  – S, D |= R
Change comes into play

- Changes in **goals/requirements**
- Changes in **domain assumptions**
  - **Usage context**
    - request profiles
  - **Physical context**
    - space, time, …
  - **Computational context**
    - external components/services (*multiple ownership*)
    - systems increasingly built out of parts that are developed, maintained, and even operated by independent parties
    - no single stakeholder oversees all parts, which may change independently
    - yet by assembling the whole one commits to achieving certain goals
Changes may affect dependability

- Changes may concern
  - $R$ evolution
  - $D$ adaptation, here I focus on $D$
- We can decompose $D$ into $D_f$ and $D_c$
  - $D_f$ is the fixed/stable part
  - $D_c$ is the changeable part

We need to detect changes to $D_c$ change detection and make changes to $S$ to keep satisfying $R$ (self) adaptation.
Change revisited

• Change recognized as a crucial problem since the 1970’s (see work by M. Lehman)
• Traditionally managed off-line: **software maintenance**
• What is new here
  – the unprecedented degree of change
  – the request that software responds to changes while the system is running (continuously running systems), possibly in a **self-managed** manner
A paradigm change

• Conventional separation between development time and run time is blurring
• Models + requirements need to be kept + updated at run time
• Continuous verification must be performed to detect the need for adaptation

Zoom-in

A framework for (self) adaptation

- C. Ghezzi, G. Tamburrelli, "Reasoning on Non Functional Requirements for Integrated Services”, RE 2009
- I. Epifani, C. Ghezzi, G. Tamburrelli, "Change-Point Detection for Black-Box Services”, FSE 2010
Specific focus

• **Non-functional** requirements
  – reliability, performance, energy consumption, cost, …
• Quantitatively stated in **probabilistic** terms
• $D_c$ decomposed into $D_u, D_s$
  – $D_u$ = usage profile
  – $D_s = S_1 \land \ldots \land S_n$ $S_i$ assumption on i-th service

Hard to estimate at design time + very likely to change at run time
Our approach in a nutshell

Offline evolution

“Real” parameters

Bayesian approach

Changes
User profiles
External services

the world

Verifier Adapter

Models

Learner

Monitor

Model-driven development

Probes

Code

Components Services

Goals
Requirements
Assumptions
Models

- Different models provide different viewpoints from which a system can be analyzed
- Focus on non-functional properties and quantitative ways to deal with uncertainty
- Use of Markov models
  - DTMCs for reliability
  - CTMCs for performance
  - Reward DTMCs for energy/cost
Properties and verification (the case of reliability)

- PCTL (probabilistic extension of CTL) provides the necessary expressive power
  - most reliability specifications can be stated as reachability properties
    - $P_{>0.8} [\Diamond (\text{system state} = \text{success})]$  
- excellent tools exist to perform verification via model checking
  - PRISM (Kwiatkowska et al.)
    - http://www.prismmodelchecker.org/
  - MRMC (Katoen at al.)
    - http://www.mrmc-tool.org/trac/
The approach in in action: e-commerce service composition

3 probabilistic requirements:
R1: “Probability of success is > 0.8”
R2: “Probability of a ExpShipping failure for a user recognized as BigSpender < 0.035”
R3: “Probability of an authentication failure is less then < 0.06”

Users classified as BigSpender or SmallSpender based on their usage profile.
## Assumptions

### User profile domain knowledge

<table>
<thead>
<tr>
<th>$D_{u,n}$</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_{u,1}$</td>
<td>$P(\text{User is a BS})$</td>
<td>0.35</td>
</tr>
<tr>
<td>$D_{u,2}$</td>
<td>$P(\text{BS chooses express shipping})$</td>
<td>0.5</td>
</tr>
<tr>
<td>$D_{u,3}$</td>
<td>$P(\text{SS chooses express shipping})$</td>
<td>0.25</td>
</tr>
<tr>
<td>$D_{u,4}$</td>
<td>$P(\text{BS searches again after a buy operation})$</td>
<td>0.2</td>
</tr>
<tr>
<td>$D_{u,5}$</td>
<td>$P(\text{SS searches again after a buy operation})$</td>
<td>0.15</td>
</tr>
</tbody>
</table>

### External service assumptions (reliability)

<table>
<thead>
<tr>
<th>$D_{s,n}$</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_{s,1}$</td>
<td>$P(\text{Login})$</td>
<td>0.03</td>
</tr>
<tr>
<td>$D_{s,2}$</td>
<td>$P(\text{Logout})$</td>
<td>0.03</td>
</tr>
<tr>
<td>$D_{s,3}$</td>
<td>$P(\text{NrmShipping})$</td>
<td>0.05</td>
</tr>
<tr>
<td>$D_{s,4}$</td>
<td>$P(\text{ExpShipping})$</td>
<td>0.05</td>
</tr>
<tr>
<td>$D_{s,5}$</td>
<td>$P(\text{CheckOut})$</td>
<td>0.1</td>
</tr>
</tbody>
</table>
DTMC model

Property check via model checking
R1: “Probability of success is > 0.8” 0.84
R2: “Probability of a ExpShipping failure for a user recognized as BigSpender < 0.035” 0.031
R3: “Probability of an authentication failure is less then < 0.06” 0.056
What happens at run time?

• We monitor the actual behavior
• A statistical (Bayesian) approach estimates the updated DTMC matrix (posterior) given run time traces and prior transitions
• Boils down to the following updating rule

\[ m_{i,j}^{(N_i)} = \frac{c_i^{(0)}}{c_i^{(0)} + N_i} \times m_{i,j}^{(0)} + \frac{N_i}{c_i^{(0)} + N_i} \times \sum_{h=1}^{d} \frac{N_{i,j}^{(h)}}{N_i} \]

A-priori Knowledge  A-posteriori Knowledge
Faults and failures

- **Fault**
  - Machine or environment do not behave as expected
- **Failure**
  - Experienced violation of requirement
- Assume that an *environment* fault is detected
  Three cases are possible
  - All Reqs still valid
    - *OK, but my signal contract violation*
  - Some Req violated + violation experienced in real world
    - *Failure detection*
  - Some Req violated, but violation not experience yet
    - *Failure prediction*
Predicted vs. detected failure

R2: Probability of a ExpShipping failure for a user recognized as BigSpender < 0.035

Suppose that execution traces that lead to updating the failure probability of ExpShipping are those involving small spenders.


Rethinking run-time environments

• Traditionally software engineering has been mostly concerned with development time
• The result is code that simply needs to be run

(Self-)adaptive software requires much more
- must be able to reason at run time about itself and the environment
  ✓ models
  ✓ goals and requirements
  ✓ strategies
  must be available at runtime
Run-time agility, incrementality

- Agility taken to extremes
  - time boundaries shrink
    ✓ constrained by real-time requirements
- Verification approaches must be re-visited
  - they must be incremental

Given $S$ system (model), $P$ property to verify for $S$
Change = new pair $S'$, $P'$

Incremental verification reuses part of the proof of $S$ against $P$ to verify $S'$ against $P'$
How to make verification incremental

Incrementality by encapsulation

• Grounded on seminal work of D. Parnas (1972)
  - Design for change
    ✓ changes must be anticipated and encapsulated within *modules*
    ✓ interface vs implementation
    ✓ interfaces formalized via contracts (B. Meyer)
• Known as *assume-guarantee* when contextualized to verification (C. Jones)

D.L. Parnas, On the criteria being used to decompose modules into systems, Comm ACM, 1972
B. Meyer, Applying “design by contract”, Computer, 1992
C. Jones. Tentative steps toward a development method for interfering programs. ACM TOPLAS, 1983
Assume-guarantee

• Show that module M1 guarantees property P1 assuming that module M2 delivers property P2, and vice versa
• Then claim that the system composed of M1 || M2 guarantees P1 and P2 unconditionally
  - these arguments support *compositional reasoning*
• Approach works if changes do not percolate through the module’s interface, affecting contract
  - effect of change *encapsulated* within the boundaries predicted at design time
How to make verification incremental

**Incrementality by parameterization**

- Requires anticipation of changes, which become parameters
- Does not require modular reasoning
- Still requires identification of elementary sources of change
- Inspired by the concept of partial evaluation (Ershov 1977)

Let $P$ be a program $P : I \rightarrow O$

Suppose $I$ can be partitioned into $I_s$ and $I_d$, where $I_s$ set of input data known statically, before runtime

Partial evaluation transforms $P$ into an equivalent residual program $P' : I_d \rightarrow O$ from by precomputing static input before runtime
An example

Requirement: $F > 0.8 \ [s = 16]$
An example (continues)

Satisfaction of requirement $F_{>0.8} \ [ s = 16]$ can be checked at design time, but at run-time, e.g. user profiles may change. We can treat them as variables and compute at design time a parametric verification formula which is then evaluated at run time.
How to make verification incremental

**Syntax-driven incrementality**

- Assumes artifact to analyze with a syntactic structure expressible as a formal grammar
- Verification is expressed via attributes (à la Knuth)
- Changes can be of any kind
Intuition

Syntax-driven incrementality

• Incremental parsing strategy finds boundary for artifact re-analysis

• Knuth proved that attributes can be only synthesized (computed bottom-up) and thus only need to be recomputed for the changed portion + propagated to the root node
Incremental parsing: intuition

- Assume $w$ is the modified portion
- Ideally, re-analyze only a sub-tree "covering" $w$, rooted in $\langle N \rangle$, and "plug-it-in" the unmodified portion of tree
- The technique works if the sub-tree is small, and complexity of re-analysis is the same as complexity of "main" algorithm
Incremental parsing: past and new results

• Past work on “mainstream” LR grammars
    ✓ Saves the maximum possible portion of the syntax tree, but the re-analyzed portion can still be large in certain cases

• Recent work resurrected Floyd’s operator precedence grammars
  - Floyd’s grammars cannot generate all deterministic CF languages
  - but in practice any programming language can be described by a Floyd grammar
  - parsing can be started from any arbitrary point of the artifact to be analyzed
Initial validation of the approach

- Case 1: reliability (QoS) analysis of composite workflows
  - a (BPEL) workflow integrates external Web services having given reliability and we wish to assess reliability of composition
  - if reliability of an external service changes, does our property about reliability of composition change?
  ✓ our previous work framed this into probabilistic model checking
  ✓ here we can deal with unrestricted changes, also in the workflow in a very efficient way
Initial validation of the approach

• Case 2: reachability analysis as supported by program model checking
  - given a program and a safety property, is there an execution of the program that leads to a violation of the property?
  - if the program changes, how does our property change?
✓ similar problem faced by Henzinger et al.
Zoom-in

Control-theory based self adaptation

[ASE 2011] A. Filieri, C. Ghezzi, A. Leva, M. Maggio,
Self-Adaptive Software Meets Control Theory: A Preliminary Approach Supporting Reliability
Goal

- Tune software through its model via feedback control loop
- Formally prove controller’s capabilities (error reduction, convergence, ...)
Conclusions and future work

• (Self-)adaptation is needed
• It requires a paradigm shift
• Run-time environments must become semantically rich
• Run-time **reasoning** must be supported, not just execution
• Continuous change and the quest for incrementality
• Benefits can be achieved by applying methods from control theory
Thanks to the group
Acknowledgements

This work is supported by and Advanced Grant of the European Research Council Programme IDEAS-ERC, Project 227977---SMScom