“Self-Healing”
Softening Precision to avoid Brittleness

Mary Shaw
Carnegie Mellon University
http://www.cs.cmu.edu/~shaw/

Institute for Software Research, International
What do I mean by “Self Healing”?

“Self-healing” means to me …

… adaptive system change that improves quality of delivered service and provides resilience to perturbation

But …

✓ Health is in the eye of the beholder
   □ Exact requirement varies among users and different times
   □ Likewise the tolerance for degraded service
   □ Most users are inarticulate about exact needs

✓ Must a system be broken in order to invoke the mechanisms provide improvement?
   □ Must you be sick before you can benefit from healing?
What sort of healing?

Local

- Fine-grained architectural reactions to local conditions
  - Replacement, tuning, rebalancing
  - Design strategies for robustness / continual improvement

Incremental

- Independent healing for different properties
- Properties that can change smoothly

Threshold-free (don’t obsess about the sick/well distinction)

- But with reference criterion or identified properties
- Emphasis on improvement, not good/bad transitions
- Scalable dependability – appropriate to need
Related ideas

\( \text{cf D. Wile} \)
- Perturbation tolerance
- Autonomic (instinctive) response
  - but here, not wired in as component-specific technique

\( \text{cf S. Gustavsson} \)
- Self-stabilization, eventual consistency
- Global properties arise from localized action
Reactive Fault Tolerance

Normal

Fault tolerant

Broken?

Command change

Institute for Software Research, International
Problems

Cost of specification
- External specifications are too complex and costly to document completely
- Diverting resources to internal distinctions seems counterproductive

User-centered requirements
- Different users have different thresholds for health
  - And different thresholds and different times
  - But often they cannot describe their thresholds precisely!

Independence of mechanism from state
- Systems often invoke the same healing actions in different states
Idea: *Homeostasis*

- Instead of distinguishing “good” and “bad” states, react to *all* change in a way that sustains a property
- Analogy: biological and ecological systems
- Examples
  - Background maintenance (garbage collection)
  - Dynamic resource selection (Internet packet routing)
  - Slack, excess capacity (service capacity)
- Advantages
  - Techniques operate over a wide range of performance
  - Performance improves independent of health status
  - Does not require precise distinctions between internal states, freeing that effort for other aspects of design
Reaction vs *Homeostasis*

**Normal**

\[ P_{NB} \]

**Fault tolerant**

\[ P'_{DN}, P'_{BD}, P'_{NB} \]

**States -> Gradients**

**Fault tolerant**

\[ \text{Normal, Degraded, Broken} \]

Institute for Software Research, International
Special case: homeostasis via slack

- Simon’s examples include excess capacity
  - e.g., inventory level sufficient to hide supply chain delay

- Queueing theory (Poisson arrivals)
  - As arrival rate approaches service time, wait \( \infty \)

- Fragility of tight coupling
  - Tightly-coupled systems have domino failures
  - Tight coupling often arises from optimization

- Safety factors
  - Civil engineers routinely overdesign by factor of 2-3

- Slack can provide mechanism-free homeostasis
  - Especially for transient perturbations
Cost-effectiveness of System Slack

- Excess capacity incurs system cost
- But so does reactive self-healing
  - Analysis
  - Monitoring
  - Healing/recovery/adaptation
  - Capacity to support reaction mechanisms
  - Reliability issues of mechanism
- Both (can) provide benefits, sometimes comparable
- Challenge is
  - compare costs for comparable benefit
  - establish comparative value when benefits differ
Realizing Homeostasis: Design rather than Mechanism

Getting beyond the specific examples

- Incremental update / consistency maintenance
  - Background garbage collection
  - Consistency propagation
- Dynamic resource selection (vs pre-bound decision)
  - Internet packet routing
  - Load balancing
- Safety factors, excess capacity, slack
  - Processing capacity instead of queue lengths
  - Mirror sites on Internet
  - Faster-than-necessary control cycle allows dropouts
Expand concept of “self-healing” to include health maintenance

  Don’t require precise distinctions between healthy and unhealthy operating states

Maintain health by making normal operation sustain performance (homeostasis)

  Generalize from examples of homeostasis to find broad design principles
BEHOLD, WE HAVE SIGNAL
Ways to deal with failure

- Traditional: prevent through careful development, analysis
- User centered: set criteria for proper operation to reflect user needs
- Fault tolerant: repair failures as they occur
- Compensatory: provide financial compensation
Ways to deal with failure

- Traditional: prevent through careful development, analysis
- User centered: set criteria for proper operation to reflect user needs
- Fault tolerant: repair failures as they occur
- Compensatory: provide financial compensation
Security technology *portfolio selection*

- Different sites have different security issues
- Elicit concerns about threats and relative priorities with multi-attribute decision techniques
  - Converts subjective comparisons to quantitative values
- Associate threat analysis with cost of successful attack and countermeasures available in the market
  - Consider cost-effectiveness and defense in depth
- Iterate, using sensitivity analysis and multiattribute techniques to refine recommendations
  - Get better understanding as well as recommendation
- Shawn Butler (finishing PhD this year)
  - Papers in ICSE 2002, CERIAS 2002
Utility-based Adaptive Configuration

- Mobile systems are resource-limited
  - Processor power, bandwidth, battery life, storage capacity, media fidelity, user distraction, …
- Users require different capabilities at different times
  - Editing, email, viewing movies, mapping, …
  - Dynamic preferences for quantity and quality of service
- Abstract capabilities can be provided by different combinations of services
  - Specific editors, browsers, mailers, players, …
- Use utility theory and linear/integer programming to find best set and configuration of services

Vahe Poladian (2nd year PhD student)
Idea: *Multidimensional cost analysis*

- **Types of cost**
  - Dollars, computer resources, user distraction, staff time, reputation, schedule, lives lost

- **Naïve view**
  - Convert all costs to a single scale, e.g., dollars

- **Problem**
  - Cost dimensions have different properties

- **Resolution**
  - Carry cost vector as far into analysis as possible
  - Convert to single scale at the latest point possible

- **Butler and Poladian, independently**
Idea: *Calculus of preference*

- Needed: a way to reconcile conflicting information
  - Multiple stakeholders
  - Multiple sources of credential information
  - Nonmonotonic information

- Possible contributing technologies
  - Utility theory: combining utility functions
  - Multi-attribute decision theory
  - Auctions
  - Priority scheduling
  - Engineering design judgments for reconciling conflicting constraints
Ways to deal with failure

- Traditional: prevent through careful development, analysis
- User centered: set criteria for proper operation to reflect user needs
- Fault tolerant: repair failures as they occur
- Compensatory: provide financial compensation
Anomaly Detection

- If you have specifications, you can detect violations
- Most everyday software does not have good specs
- Problem: how to discover “normal” behavior and capture this as predicates
  - Infer predicates from resource’s history
  - Set-up elicits user expectations while tuning predicates
  - Operation applies inferred predicates
- Inferred predicates serve as proxies for specs
- Orna Raz (PhD thesis research in progress)
  - Paper in ICSE 2002
Ways to deal with failure

- Traditional: prevent through careful development, analysis
- User centered: set criteria for proper operation to reflect user needs
- Fault tolerant: repair failures as they occur
- Compensatory: provide financial compensation
Compensation, not Prevention

🌿 For everyday software, compensation may be a reasonable alternative to repair
  ✓ Especially for time-dependent results
  ✓ Especially if consequences of failure are large enough to matter but not large enough to be catastrophic

🌿 Compensation techniques need
  ✓ Actuarial model
    🔴 Failure rate prediction based on component history
    🔴 Definitions of share-risk pools
  ✓ Ways to identify failure (e.g., anomaly detection)
  ✓ Means of assessing damages

🌿 Software component insurance

🌿 Paul Li (2nd year PhD student)
Everyday Software

The computing game has changed
  - Distributed interdependent communities of user-managed resource coalitions

Criteria for evaluating systems must change
  - User-centered requirements
  - Sufficient correctness
  - Value-based software engineering

The dependability game should also change
  - Portfolio selection
  - Anomaly detection
  - Homeostasis
  - Software component insurance
For everyday software, set criterion for dependability as “fitness for the task at hand”

Consider a wide range of approaches to achieving dependability

Achieve value through technical approaches adapted from economics and social science
Contacts

Mary Shaw
  v mary.shaw@cs.cmu.edu
  v http://cs.cmu.edu/~shaw/

Students
  v Shawn Butler (security technology selection)
  v Orna Raz (semantic anomaly detection)
  v Vahe Poladian (mobile dynamic configuration)
  v Paul Li (software component insurance)

Assistant
  v Janet New Hilf
Idea: **Aggregate Reasoning**

- Recognize that software systems are too complex for exact analysis
  - We don’t understand gasses by solving the N-body problem for extremely large N. Instead, we use the aggregate gas laws $PV = nRT$
- Seek aggregate models with system-level abstractions
  - Anomaly detection, software component insurance
  - Probabilistic certification of software components provides alternative to verification (Wallnau)
  - Exact “webs of trust” would be fragile; based on preponderance of evidence they might be robust
  - Modeling Internet as “scale-free system” yields new results, e.g. about virus spread characteristics (Barabasi)
Everyday Software

> The computing game has changed
  ✓ Internet supports mobility and a vast sea of resources
  ✓ User expectations imply context-sensitive requirements

> Criteria for evaluating systems must change
  ✓ Costs matter, not just capabilities
  ✓ Specifications will inevitably be incomplete
  ✓ “Good enough” is good enough

> The dependability game should also change
  ✓ Reconcile conflicting objectives
  ✓ Augment incomplete specs with user expectations
  ✓ Use homeostasis as alternative to feedback
  ✓ Provide compensation as alternative to repair
The Mobile Computing Challenge

- Limited hardware
  - Computer power, disk & memory capacity, battery

- Uncertain, dynamically varying services
  - Bandwidth, latency
  - Locally available information services

- Costly human attention
  - Individual, time-varying utility functions
  - Usage vs administration
  - Multi-user utility conflicts
Internet Resources as Components

Unlike conventional software components

自治 (Autonomous)
- Independently created and managed
- May change structure or format without notice
- Availability, format, semantics may change

异质 (Heterogeneous)
- Different packagings
- Different business objectives, conditions of use

开放式表现力 (Open affordances)
- Independent systems, not dependent components
- Output usually for viewing, not computation
- Incidental effects may be useful
Objective: compose autonomous distributed resources

- “Coalitions” because the resources will not have a shared objective
- “Open” in contrast to control assumed for closed-shop development

This changes everything!
## What’s changed?

<table>
<thead>
<tr>
<th>Classical</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>Localized</td>
<td>Distributed</td>
</tr>
<tr>
<td>Independent</td>
<td>Interdependent</td>
</tr>
<tr>
<td>Installations</td>
<td>Communities</td>
</tr>
<tr>
<td>Centrally-administered Software</td>
<td>User-managed Resource</td>
</tr>
<tr>
<td>Systems</td>
<td>Coalitions</td>
</tr>
</tbody>
</table>

From his keynote talk yesterday …

➤ We don’t know how software will be used:
  v Dynamic
  v Heterogeneous
  v Distributed

➤ Software evolves continually

➤ Development is still manual

➤ Increased requirements for
  v Security
  v Continuous operation
  v Maintenance
  v Change
Everyday Software

- The computing game has changed
  - Distributed interdependent communities of user-managed resource coalitions
- Criteria for evaluating systems must change
  - Costs matter, not just capabilities
  - “Good enough” is good enough
- The dependability game should also change
  - Reconcile conflicting objectives
  - Augment incomplete specs with user expectations
  - Use homeostasis as alternative to feedback
  - Provide compensation as alternative to repair
Context- Sensitive Requirements

- Different users have …
  - …different tolerance for system error and failure
  - …different interests in results from a resource
  - …different tolerance and interests at different times

- Criteria for proper operation should reflect these differences
  - Requirements can’t be tied solely to resource
  - Users need ways to express differences

- Multiple co-located users must mediate preferences
- Need *user-centered requirements* as part of resource composition techniques
Sufficient Correctness

- Traditional model of program correctness
  - Gold standard is functional correctness
  - For systems, also need extrafunctional properties

- In practice
  - Most software in everyday use has bugs …
    - … yet we get work done
  - It isn’t practical to get complete specifications
    - Too many properties people can depend on
    - Variable confidence in what we do know
    - Too expensive to collect specification information
    - Specifications should reflect users’ needs
  - We don’t really need “correctness”, but rather assurance that the software is good enough for its intended use
<table>
<thead>
<tr>
<th>Programs</th>
<th>vs</th>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete knowledge</td>
<td>Approximate knowledge</td>
<td></td>
</tr>
<tr>
<td>Goal: correctness</td>
<td>Goal: adequacy, fitness</td>
<td></td>
</tr>
<tr>
<td>Failure prevention</td>
<td>Problem remediation</td>
<td></td>
</tr>
<tr>
<td>Good component specs</td>
<td>Components poorly understood</td>
<td></td>
</tr>
<tr>
<td>Monolithic design</td>
<td>Cohesion/coupling issues</td>
<td></td>
</tr>
<tr>
<td>Stable configuration</td>
<td>Shifting (dynamic) parts</td>
<td></td>
</tr>
<tr>
<td>Open loop operation</td>
<td>Closed loop operation</td>
<td></td>
</tr>
<tr>
<td>Requirements tied to components</td>
<td>Requirements sensitive to context of use</td>
<td></td>
</tr>
<tr>
<td>Greenfield</td>
<td>Brownfield</td>
<td></td>
</tr>
<tr>
<td>Cost not a major factor</td>
<td>Cost a design driver</td>
<td></td>
</tr>
<tr>
<td><strong>Creating capability</strong></td>
<td><strong>Creating value</strong></td>
<td></td>
</tr>
</tbody>
</table>
The Value Proposition

Engineering seeks timely, cost-effective solutions to practical problems, preferably based on math and science

- This entails reconciling conflicting constraints.
- This entails making decisions with limited time, knowledge, and resources.
- This entails understanding the contribution of design decisions to cost as well as to capability.

... and so ...

The objective of software engineering should be to create value, not simply to create capability.
Value-based Software Engineering

- Include cost-benefit tradeoffs in technical decisions
  - cost-benefit of getting information as well as of analysis
  - cost-benefit of ownership as well as of development

- Adapt techniques such as
  - machine learning
  - utility theory
  - real options
  - game theory
  - multi-attribute decision theory
  - linear programming
  - classical optimization
  - portfolio selection

- Harnessing the knowledge of other disciplines, especially social scientist, in service of better software
Everyday Software

- The computing game has changed
  - Distributed interdependent communities of user-managed resource coalitions

- Criteria for evaluating systems must change
  - User-centered requirements
  - Sufficient correctness
  - Value creation, not just capability creation

- The dependability game should also change
  - Reconcile conflicting objectives
  - Augment incomplete specs with user expectations
  - Use homeostasis as alternative to feedback
  - Provide compensation as alternative to repair
Types of self-healing

- Distinguish external from internal environment
  - System has control over internal environment
  - External environment operates independently

- Fault-tolerance through feedback
  - Internal: detect system state, compare to criterion, repair if necessary, good knowledge for planning
    - Load balancing, adaptive integration
  - External: attempt to infer external state, compare to objective, react if necessary,
    - Control of mechanical systems

- Homeostasis
  - Design so normal operation maintains good conditions
    - Internet packet routing, background garbage collection

Institute for Software Research, International