

Design and implementation of Self-Healing systems

Presented by Rean Griffith
PhD Candidacy Exam
November 23rd, 2004

Self-Healing Systems

- A self healing system “...automatically detects, diagnoses and repairs localized software and hardware problems” — *The Vision of Autonomic Computing 2003 IEEE Computer Society*

Roots of Self-Healing Systems

Area	Description	Foundation Techniques
Fault tolerant computing ('70's)	Ability of a system to respond gracefully to an unexpected hardware or software failure	Fault Model specification [5] Fault Avoidance [1,3,5] Fault Detection [1,3,4,5], Fault Masking [3,4,5]
Dependable computing ('80's)	The trustworthiness of a system that allows reliance to be justifiably placed on the service it delivers	Focus on reliability and other system qualities e.g. Availability[26], Integrity[23,25], Degradation [7], Maintainability
Self-Healing systems ('90's+)	Automatically detects, diagnoses and repairs localized software and hardware problems	Dependability, Modeling [6,7,8], Monitoring [9], Diagnosis [11], Planning [16], Adaptation for Repair [13, 19, 21] (System evolution)

Overview

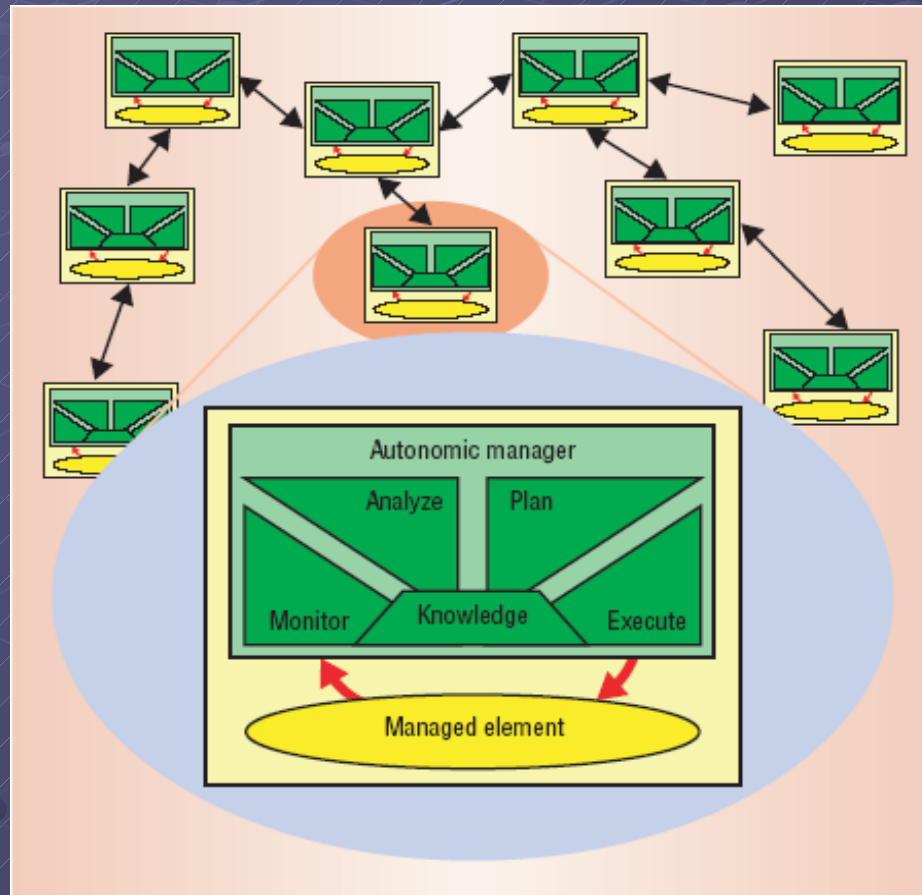
- Motivation for self-healing systems
- Building a self-healing system
- Evaluating a self-healing system*

* - omitted from the presentation for the sake of brevity but will be discussed during Q & A as necessary

Motivation for self-healing systems

- We are building increasingly complex (distributed) systems
 - Harder to build [1,12]
 - Systems must evolve over time to meet changing user needs [13,14]
 - Systems still fail embarrassingly often [26]
 - Complex and expensive to manage and maintain [12] – human in the loop a potential bottleneck in problem resolution
- Computer systems are pervading our everyday lives
 - Increased dependency on systems e.g. financial networks [2]
 - The more dependent users are on a system the less tolerant they are of interruptions [14]
 - The cost of downtime is prohibitive – not just monetary cost [18,26]
- Can we remove the human operator and let the system manage and maintain itself?
 - Possible 24/7 monitoring and maintenance
 - Cheaper
 - Faster response and resolution than human administrators

Conceptual implementation of a self-healing system



Building blocks

- Fault model specification (Koopman[5])
- Design techniques
 - Fault avoidance (Cheung[1])
 - Incorporating repair via system modeling (Koopman[7], Garlan[10])
- Implementation of system responses to faults
 - Monitoring and Detection (Avizienis[4], Debusmann[9])
 - Analysis & Diagnosis (Stojanovic[12], Chen[11])
 - Specific fault responses

Fault Model Specification

- The fault model determines response strategy based on:
 - Fault duration
 - Permanent, intermittent, transient
 - Fault manifestation
 - How will we detect?
 - What happens if we do nothing?
 - Fault source
 - Design defect, implementation defect, operational mistake, malicious attack, “wear-out”/system aging
 - Fault granularity
 - Fault containment (component, subsystem, node, task)
 - Fault profile expectations
 - Any symptoms that are definite markers of imminent failure?
 - Need to be robust for unexpected faults

Incorporating repair via modeling

Technique	When constructed	Remarks
Architectural models [8]	Design time	Architectures encapsulate knowledge and provide a basis for principled adaptation
Architectural reflection [6]	Design time	Reify architectural features as meta-objects which can be observed and manipulated at runtime
Utility modeling (Degradation) [7]	Design time	Use architecture to group components into feature sets that enable reasoning about overall system utility. Gracefully degrading systems have positive utility when any combination of components fail
Discovering architecture (Discotect) [10]	Runtime	Disconnect between design and implementation Compliment design-time models by filling in gaps

Building blocks

- Fault model specification (Koopman[5])
- Design techniques
 - Fault avoidance (Cheung[1])
 - Incorporating repair via system modeling (Koopman[7], Garlan[10])
- Implementation of system responses to faults
 - ➔ ▪ Monitoring and Detection (Avizienis[4], Debusmann[9])
 - Analysis & Diagnosis (Stojanovic[12], Chen[11])
 - Specific fault responses

Monitoring & Detection

● Detection

- First step in responding to faults (Koopman[5])
- Different granularities
 - Internal to the component (Self-checking software[1])
 - Supervisory checks (Recovery blocks [3])
 - Comparisons with replicated components (N-version [4])
 - Instrumentation (Garlan[8],Debusmann[9],Valetto[21])
- Different levels of intrusiveness of detection [1]
 - Non-intrusive checking of results
 - Execution of audit/check tasks
 - Redundant task execution
 - Online self-tests/periodic reboots for self-tests
 - Fault injection


Building blocks

- Fault model specification (Koopman[5])
- Design techniques
 - Fault avoidance (Cheung[1])
 - Incorporating repair via system modeling (Koopman[7], Garlan[10])
- Implementation of system responses to faults
 - Monitoring and Detection (Avizienis[4], Debusmann[9])
 - ➔ ▪ Analysis & Diagnosis (Stojanovic[12], Chen[11])
 - Specific fault responses

Analysis & Diagnosis

- Analysis [12] for problem determination
 - Data correlation and inference technologies to support automated continuous system analysis over monitoring data
 - We need reference points – models and/or knowledge repositories which we can use/reason over to determine whether a problem exists
- Diagnosing faults [11]
 - Locates source of a fault after detection e.g. Decision trees [11]
 - Specificity down the offending line of code often not necessary
 - Improve adaptation strategy selection

Building blocks

- Fault model specification (Koopman[5])
 - Design techniques
 - Fault avoidance (Cheung[1])
 - Incorporating repair via system modeling (Koopman[7], Garlan[10])
 - Implementation of system responses to faults
 - Monitoring and Detection (Avizienis[4], Debusmann[9])
 - Analysis & Diagnosis (Stojanovic[12], Chen[11])
 - Specific fault responses
- 

Specific fault responses

● Fault response

■ Reactive

- Degradation e.g. Killing less important tasks (Koopman[5,7])
- Repair Adaptations (Dynamic updates[18], Reconfigurations [2,14,20,23])
- Roll forward with compensation (System-level undo [25])*
- Functional alternatives [28]*
- Requesting help from the outside (Koopman[5])

■ Proactive

- Actions triggered by faults that are indicative of possible near-term failure (Koopman[5])

■ Preventative

- Periodic micro-reboot, sub-system reboot, system reboot (Software Rejuvenation[24])*

* - omitted from the presentation for the sake of brevity but will be discussed during Q & A as necessary

Degradation

● Degradation (Koopman[7])

- The degree of degraded operation of a system
 - ...is its resilience to damage that exceeds built in redundancy (Koopman[5])
- System might not be able to restore 100% functionality after a fault
- Some systems must fail if they are not 100% functional
- Others can degrade performance, shed tasks or failover to less computationally expensive degraded mode algorithms

Adaptation

- Adaptation allows a system to respond to its environment
- Self-healing systems are expected to be amenable to evolutionary change with minimal downtime (Kramer[13])
 - Characteristic types of evolution (Oreizy[15])
 - Corrective – removes software faults
 - Perfective – enhances product functionality
- Self-healing systems can be:
 - Closed adaptive (Oreizy[16]) – self contained
 - Open adaptive (Oreizy[16]) – amenable to adding new behaviors

Repair Adaptations

● Many kinds of repairs

- ADT/Module replacement – Fabry[2]
- Method replacement – Dymos [14]
- Component replacement (Hot-swaps) DAS [14], K42 [20]
- Server replacement – Conic[13], MTS [14]
- Verifiable patches – Popcorn [18] (C-like language)

● Risky changes to make in the regular update cycle of:

- Stop system
- Apply update
- Restart

● Now we may have to perform repair as the system runs

- Cost and risks will determine whether online repair is a viable prospect
- Can we make online repair less risky?

Adaptation techniques

Adaptation granularity	Language/compiler support	Indirection	Quiescence	State transfer	Interface Changes supported?	Verifiable/provably safe?
ADT/ Module replacement		Fabry[2]	Fabry[2]	Fabry[2]		
Method replacement	Dymos[14]		Dymos – program state	Dymos [14]	Dymos[14]	
Component replacement		DAS[14] K42[20]	DAS[14] K42[20]		Zdonik[23]	
Server		Conic[13]	Conic[13]			
Arbitrary patches		Popcorn[18]	Popcorn – programmer hints	Popcorn [18]		Popcorn [18] – provably safe patch

Managing the risk of online repair

● Key activities

- Change management [13]

- A principled aspect of runtime system evolution that:

- Helps identify what must be changed
 - Provides context for reasoning about, specifying and implementing change
 - Controls change to preserve system integrity as well as meeting the additional extra-functional requirements of dependably systems e.g. availability, reliability etc.

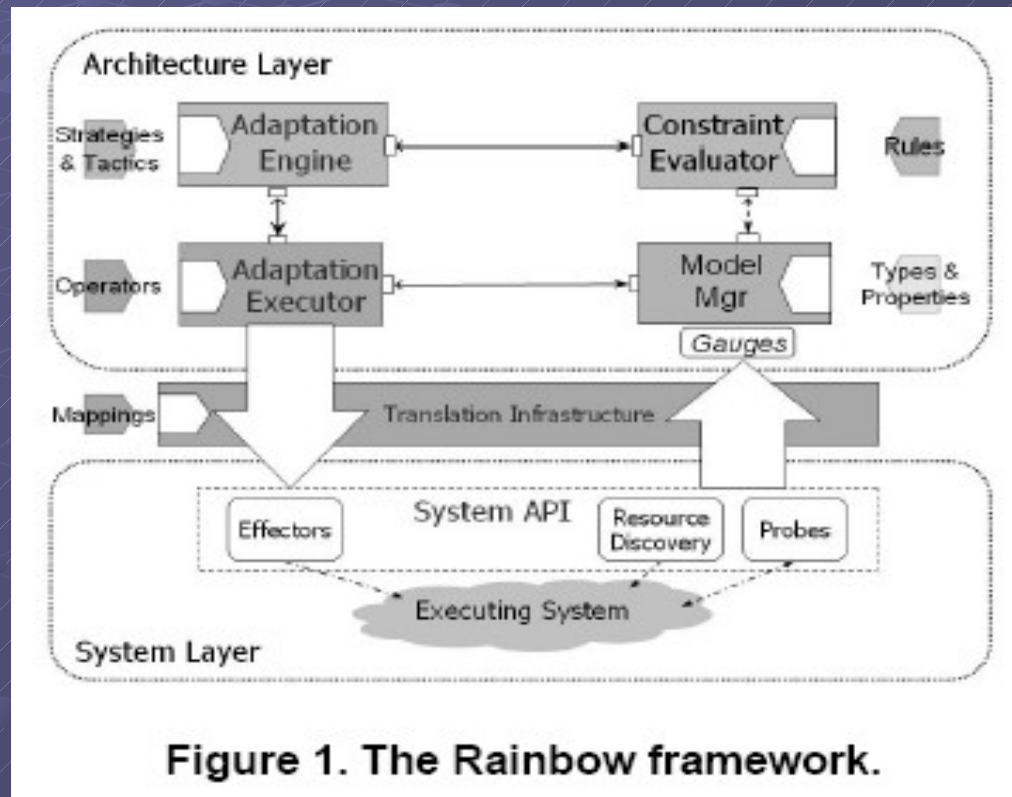
Motivation for Change Management

- Why change management is necessary
 - Need for novel approaches to maintenance
 - Principled approaches – Model-based healing [19] – supported by reusable infrastructure e.g. Rainbow [2], Workflakes [21] preferred over ad hoc adaptation mechanisms
- Objectives of change management (Kramer[13])
 - Changes specified only in terms of system structure e.g. (Wermlinger[17])
 - Change specifications are independent of application protocols, algorithms or states
 - Changes leave the system in a consistent state
 - Change should minimize disruption to system

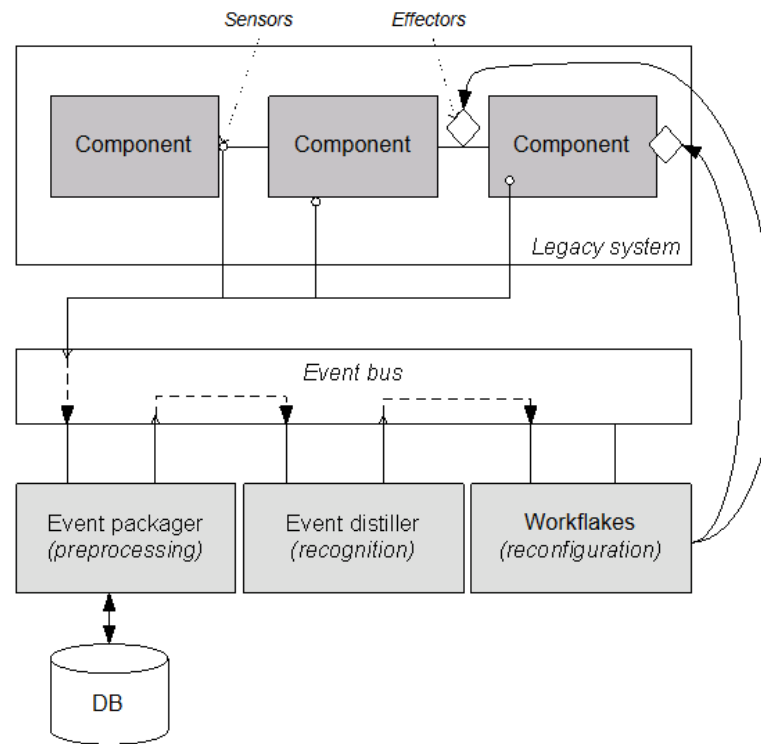
Externalized adaptation [19,21,22]

- One or more models of a system are maintained at runtime and external to the application as a basis for identifying problems and resolving them
- Describe changes in terms of operations on the model
- Changes to the model effects changes to the underlying system

Rainbow[22]



Kinesthetics eXtreme (KX) featuring Workflakes[21]



Conclusions

- Using models as the basis for guiding and validating repair and system integrity is a promising principled approach
- Ontologies allow us to capture, reuse and extend domain specific knowledge
- More flexible to construct the system such that there is a separation between functional concerns and repair adaptation logic
- Change management is integral to minimizing the risks of repairing a running system
- We need other “benchmarks” for evaluating self-healing systems other than raw performance

References

1. **Design of Self-Checking Software**
S. S. Yau, R. C. Cheung
Proceedings of the international conference on Reliable software pp 450 - 455 1975.
4. **How to Design a System in which Modules can be changed on the fly**
R. S. Fabry
International Conference on Software Engineering 1976.
- **System Structure for Software Fault Tolerance**
B. Randell
Proceedings of the international conference on Reliable software pp 437 - 449 1975.
- **The N-Version Approach to Fault-Tolerant Software**
Algirdas Avizienis, Fellow, IEEE, IEEE Transactions on Software Engineering Vol. SE-11 No. 12 December 1985.
- **Elements of the Self-Healing Problem Space**
Philip Koopman
ICSE Workshop on Architecting Dependable Systems 2003.
15. **Architectural Reflection: Realising Software Architectures via Reflective Activities**
Francesco Tisato, Andrea Savigni, Walter Cazzola, Andrea Sosio
Engineering Distributed Objects, Second International Workshop, EDO 2000, Davis, CA, USA, November 2-3, 2000, Revised Papers pp 102 - 115.
18. **Using Architectural Properties to Model and Measure System-Wide Graceful Degradation**
C. Shelton and P. Koopman
Accepted to the Workshop on Architecting Dependable Systems sponsored by the International Conference on Software Engineering (ICSE2002)
- **Exploiting Architectural Design Knowledge to Support Self-Repairing Systems**
Bradley Schmerl, David Garlan
Proceedings of the 14th international conference on Software engineering and knowledge engineering Pages: 241 - 248 2002.
- **Efficient and transparent Instrumentation of Application Components using an Aspect-oriented Approach**
M. Debusmann and Kurt Geihs
14th IFIP/IEEE Workshop on Distributed Systems: Operations and Management (DSOM 2003) pp 209--220.
- **Discotect: A System for Discovering Architectures from Running Systems**
Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, Rick Kazman
International Conference on Software Engineering archive Proceedings of the 26th International Conference on Software Engineering - Volume 00 Pages: 470 - 479 2004.
- **Failure Diagnosis using Decision Trees**
Mike Chen, Alice X. Zheng, Jim Lloyd, Michael I. Jordan, Eric A. Brewer
1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA pp36 - 43.

References

1. **The role of ontologies in autonomic computing systems**
L. Stojanovic, J. Schneider, A. Maedche, S. Libischer, R. Studer, Th. Lumpp, A. Abecker, G. Breiter, and J. Dinger
IBM Systems Journal Volume 43, Number 3, 2004 Unstructured Information Management.
4. **The Evolving Philosophers Problem: Dynamic Change Management**
Jeff Kramer, Jeff Magee
IEEE Transactions on Software Engineering November 1990 Vol. 16 - No. 11 pp 1293 - 1306.
7. **On-The-Fly Program Modification: Systems For Dynamic Updating**
Mark E. Segal, Ophir Frieder
IEEE Software Volume 10, Number 2, March 1993 pp 53 - 65.
10. **Architecture-Based Runtime Software Evolution**
Peyman Oreizy, Nenad Medovic, Richard N. Taylor
In Proceedings of the International Conference on Software Engineering 1998 (ICSE'98), pages 177--186, April 1998.
- **An Architecture-Based Approach to Self-Adaptive Software**
Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, Alexander L. Wolf
IEEE Intelligent Systems archive Volume 14, Issue 3 (May 1999) pp 54 - 62.
17. **A Graph Based Architectural (Re)configuration Language**
Michael Wermlinger, Antonia Lopes, Jose Luiz Fiadeiro
Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001 pp 21 - 32.
21. **Dynamic Software Updating**
Michael W. Hicks, Jonathan T. Moore, Scott Nettles
Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001. SIGPLAN Notices 36(5) (May 2001), ACM, 2001, ISBN 1-58113-414-2 pp 13 - 23.
26. **Model-based Adaptation for Self-Healing Systems**
David Garlan, Bradley R. Schmerl
Proceedings of the First Workshop on Self-Healing Systems, WOSS 2002, Charleston, South Carolina, USA, November 18-19, 2002 pp 27 - 32.
30. **System Support for Online Reconfiguration**
C. A. N. Soules, J. Appavoo, K. Hui, D. D. Silva, G. R. Ganger, O. Krieger, M. Stumm, R. W. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis
In Proceedings of the Usenix Technical Conference, San Antonio, TX, USA, June 2003.
34. **Using Process Technology to Control and Coordinate Software Adaptation**
Giuseppe Valetto, Gail Kaiser
International Conference on Software Engineering archive Proceedings of the 25th international conference on Software engineering Portland, Oregon Pages: 262 - 272 2003.

References

- 22. Rainbow: Architecture-based Self-adaptation with Reusable Infrastructure**
Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley R. Schmerl, Peter Steenkiste
1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA.
- 23. Maintaining Consistency in a Database with Changing Types**
Stanley B. Zdonik Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming Yorktown Heights, New York, United States Pages: 120 - 127 1986.
- 24. Software Rejuvenation: Analysis, Module and Applications**
Yennun Huang, Chandra Kintala, Nick Kolettis, N. Dudley Fulton
Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25), Pasadena, CA, pp. June 1995, pp. 381-390.
- 25. Rewind, Repair, Replay: Three R's to Dependability**
A. Brown and D. A. Patterson
In 10th ACM SIGOPS European Workshop, 2002.
- 26. Improving Availability with Recursive Micro-Reboots: A Soft-State System Case Study**
George Candea, James Cutler, Armando Fox
Dependable systems and networks-performance and dependability symposium (DSN-PDS) 2002: Selected papers pp 213 - 248.
- 27. Predicting Problems Caused by Component Upgrades**
Stephen McCamant and Michael D. Ernst
In 10th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pages 287--296, Helsinki, Finland, September 2003.
- 28. Improving System Dependability with Functional Alternatives**
Charles P. Shelton and Philip Koopman
Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04) - Volume 00 Page: 295.

Backup slides

Component definitions

● Component

- Software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard

● Component model

- Defines specific interaction and composition standards

● Component model implementation

- Dedicated set of executable software elements required to support the execution of components that conform to that model

Efficient and transparent instrumentation of application components using an aspect-oriented approach

● Overview of AOP

- Developed at Xerox PARC in the early 90's (Gregor Kiczales et. Al)
- Modularize cross-cutting concerns (non-functional requirements) and separate them from functional requirements
- Some terms
 - Joinpoint – well defined point in execution flow
 - Pointcut – identifies a joinpoint e.g. using a regex
 - Advice – defines additional code to be executed at a joinpoint (before, around, after)
 - Introductions – modify classes, add members, change class relationships
- Aspects and application code can be developed separately and weaved together
 - Compile time weaving – AspectJ
 - Runtime weaving – JBoss bytecode weaving
 - Hybrid

Rainbow – architecture-based self-adaptation with reusable infrastructure

Definitions

- Architecture
 - An abstract view of the system as a composition of computational elements and their interconnections [Shaw & Garlan]
- Architectural model
 - Represents the system architecture as a graph of interacting components [popular format – ACME, xADL, SADL]
 - Nodes in the graph are termed components – principal computation elements and data stores of the system
 - Arcs are termed connectors and represent the pathways of interaction
- Architectural style
 - Style specifies a vocabulary of component and connector types and the rules of system composition, essentially capturing the units of change from system to system
 - Characterizes a family of systems related by shared structural and semantic properties {Types, Rules, Properties, Analyses}
- Architecture-based self-adaptation
 - The use of architectural models to reason about system behavior and guide adaptation

System Validation/Assurance

- Why should users trust a self-healing system to do the “right” thing?
 - The system must be able to give guarantees about its qualitative characteristics before, during and after adaptations
 - Availability, reliability, performance etc.
 - What kind of guarantees can the system give?
 - Absolute
 - Probabilistic
 - Partial assurance of a few key system properties
 - Assurance during degraded mode operations
 - Assurance between fault occurrence time and detection time
 - Not all faults are detectable

System Validation/Assurance

● Improving dependability

- Software rejuvenation [24]
 - Addresses faults caused by software aging e.g. resource leaks
- System-level undo [25]
 - Addresses operator-induced/compounded failures
 - Rewind – revert state to an earlier time
 - Repair – operator makes any desired changes to the system
 - Replay – roll forward with the “new” system
- Functional alternatives rather than redundancy [28]
 - Exploit a system’s available features to provide some system redundancy without additional redundant components
 - Leverages components that satisfy overlapping system objectives

Rewind, repair, replay: three r's to dependability

● Mechanism

- How is this different from related work?
 - Backup/restore/checkpointing schemes offer rewind/repair but deny the ability to roll forward once changes have been made
 - Recovery systems for databases use rewind/replay to recover from crashes, deadlocks and other fatal events but do not offer the ability to inject repair into the recovery cycle
 - The standard transaction model does not allow committed transactions to be altered or removed however extended transaction models have introduced the notion of a compensating transaction

Rewind, repair, replay: three r's to dependability

● Mechanism

- Tracking recoverable state for replay
 - When an undo is carried out it is the responsibility of the replay step to restore all the state changes that are important to the user
 - Defining exactly what state this encompasses is tricky e.g. an upgrade of a mail server could change the underlying format of the mailbox file
 - Solution: Track user actions at the intentional level e.g. in an undoable email system a user's act of deleting a message is recorded as "delete message with ID x" rather than "alter byte range x – y of file z"

Rewind, repair, replay: three r's to dependability

● Mechanism

- How does intentional tracking work
 - In the network service environment being targeted, users interact with the system through standardized application protocols (usually comprised of a few action verbs e.g. HTTP PUT, GET, POST).
 - Intercept and record user interactions at the protocol level

Rewind, repair, replay: three r's to dependability

● Mechanism

- Are repairs also tracked?
 - No, because of practical reasons
 - User interactions are limited by the verbs-action commands permitted by the protocol, however repair actions are limited only by the ingenuity of the human operator

Rewind, repair, replay: three r's to dependability

● Mechanism

- Architecture

- Basic structure of an undoable system

- A verb-based application interface that uses logical state names. This allows the undo system to properly disambiguate recoverable user state changes from other system events
 - The verb based interface should cover all interactions with the system that affect user-visible state, including operator interfaces that are used to create, move, delete and modify user state repositories
 - If existing protocols do not satisfy these requirements then new protocols can be created and layered above existing interfaces

System Validation/Assurance

● Improving availability

- Recursive micro-reboots [26]
 - Consider failures as facts to be coped with
 - Unequivocally returns the recovered system to its start state – best understood/tested state
 - High confidence way to reclaim resources
 - Easy technique to understand and employ, automate, debug, implement
- Crash-only components
 - One way to shutdown – crash
 - One way to start up – recover
 - All persistent state kept in crash-only data stores
 - Strong modularity, timeout based communication, lease-based resource allocation

● Improving integrity

- Managing multiple versions via version handlers [23]
- Predicting problems caused by component upgrades [27]
- Model constraints used in change management [13,19]

Improving availability with recursive micro-reboots

● Mechanism

- Recursive recovery and micro-reboots
 - A recursive approach to recovery
 - A minimal subset of components is recovered
 - If that fails progressively larger subsets are recovered
 - A micro-reboot is a low-cost form of reboot applied at the level of fine-grained software components
 - For a system to be recursively recoverable it must contain fine-grained components that are independently recoverable such that part of the system can be recovered without touching the rest
 - Components must be loosely coupled AND prepared to be denied service from other components that may be in the process of micro-rebooting
 - Popularity of loosely-coupled componentized systems (EJB,.NET)
 - A recursively recoverable system gracefully tolerates successive restarts at multiple levels in the sense that it does not lose or corrupt data or cause other systems to crash

Repair adaptation backup slides

ADT/Module replacement

- Changing modules on-the-fly [2]
 - Simple case – local data only
 - Indirection mechanism
 - Old and new version instances available side-by-side for a short time
 - Old version services existing requests, new requests are redirected transparently to the new version
 - General case – Shared/persistent data
 - New version of a module modifies representation of a shared data structure on first encounter
 - Version numbers signal whether to modify
 - Modifications must be delayed until old versions are completely finished with shared instances
- Other requirements
 - Correctness of modifications should be analyzed/proven
 - Access control to limit who can change indirection mechanisms and initiate module swapping

Method replacement

● Dynamic Modification System (Dymos) [14]

- Supports changing a procedure's interface between versions
- Supports the implementation of static data local to a procedure
- Dymos is completely integrated – source code, object code, compilation artifacts (parse trees, ASTs and symbol tables) are available at all times
- Supports the specification of the modules to update and the conditions that must be satisfied
- Tightly tied to the StarMod language, which was modified to support dynamic program updating
- Tools must be on the host system to manipulate the compilation artifacts
- Assumes the source code is always available
- No support for distributed systems

Component replacement

- Dynamically Alterable System (DAS) operating system [14]
 - Supports re-plugging – swapping modules with the same interface
 - Supports restructuring of data within modules
 - Does not handle interface changes
- K42 operating system [20]
 - Two basic mechanisms for online reconfiguration
 - Interposition
 - Hot-swapping

Server replacement

● Conic [13]

- Handle configuration at the configuration level
 - Components and connectors (architecture)
- Specify change declaratively in terms of system structure only
- System modification via change transactions generated from change specifications
- Use change specifications to scope change
- Do not force change rather wait for nodes to quiesce. Nodes remain quiescent during change
- For change as a result of failure nodes would need to include recovery actions

● Michigan Terminal System (MTS) [14]

- Assumes interface remains constant between versions
- Coarse granularity for replacements – large system components e.g. a command interpreter
- Does not address cases where a new service calls some old service
- Data formats cannot change between versions
- Services disabled for the duration of the upgrade

Verifiable patches

● Popcorn [18]

- Code and data patches
- Build dynamic patches on top of dynamic linking mechanisms
- Program re-linking after loading patch
 - Programmer specifies when to apply patches
- Semi-automatic patch generation based on source comparison
 - Identify changes to functions and data
 - Generate stub functions and state transformers
- Native code used in dynamic patches is coupled with annotations such that the code is provably safe. A well-formed Typed Assembly Language (TAL) program is
 - Memory-safe – no pointer forging
 - Control-flow safe – no jumps to arbitrary memory locations
 - Stack-safe – no modification of non-local stack frames
- System includes a TAL verifier and a prototype compiler from a safe-C language, Popcorn, to TAL