Architecture-based Systems Management

Sacha Krakowiak
University of Grenoble & INRIA
http://sardes.inrialpes.fr/~krakowia
The challenge of complexity

- An increasing number of human activities now rely on computing systems.
  - Communication, transportation
  - Commerce, finance
  - Energy production
  - Health care
The challenge of complexity

- An increasing number of human activities now rely on computing systems.
  - Communication, transportation
  - Commerce, finance
  - Energy production
  - Health care

- However, today’s computing systems have become so complex that one hardly understands how they work...
The challenge of complexity

- An increasing number of human activities now rely on computing systems.
  - Communication, transportation
  - Commerce, finance
  - Energy production
  - Health care

- However, today’s computing systems have become so complex that one hardly understands how they work...

- ... and one hardly understands why they fail.

  - Some investigations
  - Murphy (1993)
  - Oppenheimer, Ganapathi, Patterson (2003)
The origin of failures in Internet-based systems

Failure cause by % of service failures

Reminder:

A failure is a deviation from the specified behavior

A fault is any (potential) cause of a failure

The origin of failures in Internet-based systems

Failure cause by % of service failures

**Online**
- Operator: 33%
- Network: 20%
- Software: 25%
- Hardware: 10%
- Unknown: 12%

**Content**
- Operator: 36%
- Network: 15%
- Software: 25%
- Hardware: 2%
- Unknown: 22%

**Read Mostly**
- Operator: 19%
- Network: 62%
- Software: 5%
- Unknown: 14%

In addition, most operator faults are **configuration** faults

Reminder:
A **failure** is a deviation from the specified behavior
A **fault** is any (potential) cause of a failure

D. Oppenheimer, A. Ganapathi, D. A. Patterson. Why do Internet services fail and what can be done about it? *Proc 4th Usenix Symp. On Internet Technologies and Systems (USITS’03), 2003*
The origin of failures in Internet-based systems

**Failure cause by % of service failures**

**Online**
- Operator: 33%
- Network: 20%
- Software: 25%
- Hardware: 10%
- Unknown: 12%

**Content**
- Operator: 36%
- Network: 15%
- Software: 25%
- Hardware: 2%
- Unknown: 22%

**ReadMostly**
- Operator: 19%
- Network: 62%
- Software: 5%
- Unknown: 14%

In addition, most operator faults are configuration faults.

**Reminder:**

A **failure** is a deviation from the specified behavior.

A **fault** is any (potential) cause of a failure.

**Earlier studies:**

- **Tandem Systems (Gray)**
  1985: Operator 42%, S/W 25%, H/W 18%
  1989: Operator 15%, S/W 55%, H/W 14%

- **Vax (Murphy)**
  1993: Operator 50%, S/W 20%, H/W 10%

**D. Oppenheimer, A. Ganapathi, D. A. Patterson. Why do Internet services fail and what can be done about it? Proc 4th Usenix Symp. On Internet Technologies and Systems (USITS’03), 2003**
The challenge of system administration

- System administration is getting too complex for humans
  One remedy: computer-assisted administration

- What is system administration?
  Ensuring that the system provides a given level of quality of service
  Maintaining this QoS level in the face of adverse conditions.

- Quality of service has many facets
  Availability
    Including partial availability
  Performance
    Mean throughput, latency, etc.
    Differentiated levels
  Security
    Well-known and new threats
System administration tasks

- Defining policies
  - Defining QoS evaluation criteria
  - Defining goals
  - Setting priorities
System administration tasks

- Defining policies
  - Defining QoS evaluation criteria
  - Defining goals
  - Setting priorities

- Configuring and deploying a system
  - Selecting components
  - Choosing location for placement
  - Setting parameter values
System administration tasks

- **Defining policies**
  - Defining QoS evaluation criteria
  - Defining goals
  - Setting priorities

- **Configuring and deploying a system**
  - Selecting components
  - Choosing location for placement
  - Setting parameter values

- **Reacting to external events**
  - Unexpected / undesirable events
    - Hardware, software or network failure
    - Load peak
    - Security attack
  - Reaction often involves system reconfiguration
System administration tasks

- Defining policies
  - Defining QoS evaluation criteria
  - Defining goals
  - Setting priorities

- Configuring and deploying a system
  - Selecting components
  - Choosing location for placement
  - Setting parameter values

- Reacting to external events
  - Unexpected / undesirable events
    - Hardware, software or network failure
    - Load peak
    - Security attack
  - Reaction often involves system reconfiguration

*Can be (partially) automated*
Architecture-based management
Architecture-based management

- **System architecture**
  
  A framework for describing a system as an assembly of parts (components)
Architecture-based management

- **System architecture**
  A framework for describing a system as an assembly of parts (components)

- **What is architecture-based management?**
  Using the architectural description of the managed system as a guide for defining and implementing management functions
System architecture
A framework for describing a system as an assembly of parts (components)

What is architecture-based management?
Using the architectural description of the managed system as a guide for defining and implementing management functions

Why architecture-based management?
Higher abstraction level
Convenient mapping between management and architecture notions
Reduced architectural erosion (discrepancy between conceptual and actual architecture)
Automated support for management functions
Main concepts of software architecture (1)

- Describing a system as an assembly of parts
- Compositional entities
  
  Component.
  
  A unit of composition and independent deployment
  Fulfils a specific function
  May be assembled with other components
  Has contractually specified interfaces (provided and required)

  Connector
  
  A device that allows assembling components, using provided and required interfaces
  Two roles: binding and communication

  Configuration
  
  An assembly of components (may or may not be itself a component).
Main concepts of software architecture (2)

Architecture Description Language (ADL)

Provides a common (formal or semi-formal) global description of a system, for designers and implementers

Can be used by various tools (visualisation, verification, code generation, deployment and reconfiguration, etc.)

Not all component systems use an ADL

Some use dependency descriptions (examples later)

No commonly accepted standard

Current issues for ADLs

Extension mechanisms

Common core + extensions

XML as main notation

Dynamic ADLs

Executed at run time

Causes the structure to evolve
Plan of this talk

✶ Managing component-based systems
  Configuration and deployment
  Case study
    The SmartFrog framework
  Package-based software distributions
  Case studies
    EDOS
    Nix

✶ Self-repair
  Case study: the Jade framework

✶ Perspectives
Configuration and deployment tasks

- Selecting the components, setting parameters
- Verifying the consistency of the system (e.g., dependencies)
- Determining the sites on which the system is to be installed and placing each component on the appropriate site
- Setting up the connections between the components
- Starting the components in an appropriate order
Configuration and deployment (2)

- Requirements
  Allow variability (ability to modify a system according to needs); this implies flexibility, i.e., ability to:
  - Apply changes at any point of the product’s lifecycle
  - Delay changes up to the latest possible moment
  - Use any policy for change management
  - Allow several versions of a component to coexist
Configuration and deployment (2)

✦ Requirements

Allow *variability* (ability to modify a system according to needs); this implies *flexibility*, i.e., ability to:

- Apply changes at any point of the product’s lifecycle
- Delay changes up to the latest possible moment
- Use any policy for change management
- Allow several versions of a component to coexist

✦ Why is this difficult?

Large scale, complex systems
Keeping track of multiple configurations
Maintaining consistency in the face of change
Configuration and deployment (2)

❖ Requirements

Allow *variability* (ability to modify a system according to needs); this implies *flexibility*, i.e., ability to:

- Apply changes at any point of the product’s lifecycle
- Delay changes up to the latest possible moment
- Use any policy for change management
- Allow several versions of a component to coexist

❖ Why is this difficult?

- Large scale, complex systems
- Keeping track of multiple configurations
- Maintaining consistency in the face of change

❖ Bad practice

- Configuration data scattered in many places (sometimes repeated)
- Incompatible lifecycles between components
- Ad hoc configuration and deployment procedures
Problems of configuration and deployment

- Preventing unresolved dependencies
  Dependencies are not always explicit
  Dependencies may occur at build time or at run time
  Dependencies may even be unknown to the administrator

- Allowing multiple versions to coexist
  Different applications may require different versions of a library
  Multiple versions may be mutually incompatible

- Preventing component interference
  An upgrade of a component may invalidate another component
  (file overwriting, etc.)
  Using “standard” paths (e.g., in Unix) is a potential cause of interference
Architecture-based deployment

- The description of a system’s configuration and deployment is separate from the code and expressed in terms of the system’s architecture.
- This description is used as a base for automating the process of configuration and deployment.
Configuration and deployment: case study

- SmartFrog
  “Smart Framework for Object Groups”
  A configuration and deployment framework for (potentially large) distributed systems
  Examples
  - a network monitoring system
  - a 3-tier web application
  Developed by HP Labs
  Available in open source
  Used in production environments
Introducing SmartFrog

- SmartFrog provides capabilities for
  - Configuration: describing and composing a distributed application out of Java components
  - Deployment: installing a configuration on a set of computing resources
  - Lifecycle management: orchestrating the progress of components through their lifecycles (deploy, start, terminate, …)
  - Discovery and communication: locating components both statically and at run time; communicating between components

- SmartFrog consists of
  - A component model
  - A declarative language for configuration and deployment description
  - A run time system (distributed workflow engine)
SmartFrog component structure

Standardised APIs:
- access to configuration data
- lifecycle API

Application-specific API
- interface of managed entity (component)

The lifecycle manager is used as a wrapper for legacy software.

Components persist at run time (the component structure does not disappear after deployment)

Configuration information may be statically provided or discovered at run time (see later)
The lifecycle API for a component consists of the methods

- deploy
- start
- terminate

The lifecycle for a configuration (a compound component, extending the predefined Compound class) is implemented by lifecycle managers (described later on), which use the components’ API
SmartFrog configuration description (1)

- **Requirements**
  - Composable description
  - Late binding
  - Ability to extend framework
  - Parameterised description (templates)

- **Overview**
  - A declarative data description language (not a programming language)
    - Attribute = name-value pair
  - Prototype-based (instance-inheritance)
  - Templates
    - May be extended, overridden, combined
  - May include assertions, to check validity of data
  - Interpreted by the run time system
    - No semantics built in the language
#include "wstemplate.sf"
#include "dbtemplate.sf"

sfConfig extends {
    commonPort 8080;
    ws1 extends webServerTemplate {
        sfProcessHost "webserver1.hpl.hp.com;"
        port ATTRIB commonPort;
    }
    ws2 extends webServerTemplate {
        sfProcessHost "webserver2.hpl.hp.com;"
        port ATTRIB commonPort;
        useDB LAZY ATTRIB db;
    }
    db extends dbTemplate {
        userTable: rows 6;
    }
}
**SmartFrog configuration description (2)**

**wstemplate.sf**

```plaintext
webServerTemplate extends {
   sfProcessHost "localhost";
   port 80;
   useDB;
}
```

**dbtemplate.sf**

```plaintext
dbTemplate extends {
   userTable extends {
      columns 4;
      rows 3;
   }
   dataTable extends {
      columns 2;
      rows 5;
   }
}
```

**webservice.sf**

```plaintext
#include "wstemplate.sf"
#include "dbtemplate.sf"

sfConfig extends {
   commonPort 8080;
   ws1 extends webServerTemplate {
      sfProcessHost "webserver1.hpl.hp.com;"
      port ATTRIB commonPort;
   }
   ws2 extends webServerTemplate {
      sfProcessHost "webserver2.hpl.hp.com;"
      port ATTRIB commonPort;
      useDB LAZY ATTRIB db;
   }
   db extends dbTemplate {
      userTable: rows 6;
   }
}
```
SmartFrog configuration description (2)

wstemplate.sf

webServerTemplate extends {
    sfProcessHost "localhost";
    port 80;
    useDB;
}

dbtemplate.sf

dbTemplate extends {
    userTable extends {
        columns 4;
        rows 3;
    }
    dataTable extends {
        columns 2;
        rows 5;
    }
}

webservice.sf

#include "wstemplate.sf"
#include "dbtemplate.sf"

sfConfig extends {
    commonPort 8080;
    ws1 extends webServerTemplate {
        sfProcessHost "webserver1.hpl.hp.com;"
        port ATTRIB commonPort;
    }
    ws2 extends webServerTemplate {
        sfProcessHost "webserver2.hpl.hp.com;"
        port ATTRIB commonPort;
        useDB LAZY ATTRIB db;
    }
    db extends dbTemplate {
        userTable: rows 6;
    }
}
SmartFrog configuration description (2)

wstemplate.sf

```c
webServerTemplate extends {
    sfProcessHost "localhost";
    port 80;
    useDB;
}
```

dbtemplate.sf

```c
dbTemplate extends {
    userTable extends {
        columns 4;
        rows 3;
    }
    dataTable extends {
        columns 2;
        rows 5;
    }
}
```

webservice.sf

```c
#include "wstemplate.sf"
#include "dbtemplate.sf"

sfConfig extends {
    commonPort 8080;
    ws1 extends webServerTemplate {
        sfProcessHost "webserver1.hpl.hp.com;",
        port ATTRIB commonPort;
    }
    ws2 extends webServerTemplate {
        sfProcessHost "webserver2.hpl.hp.com;",
        port ATTRIB commonPort;
        useDB LAZY ATTRIB db;
    }
    db extends dbTemplate {
        userTable: rows 6;
    }
}
```
SmartFrog configuration description (2)

**wstemplate.sf**

```
webServerTemplate extends {
    sfProcessHost "localhost";
    port 80;
    useDB;
}
```

**dbtemplate.sf**

```
dbTemplate extends {
    userTable extends {
        columns 4;
        rows 3;
    }
    dataTable extends {
        columns 2;
        rows 5;
    }
}
```

**webservice.sf**

```
#include "wstemplate.sf"
#include "dbtemplate.sf"

sfConfig extends {
    commonPort 8080;
    ws1 extends webServerTemplate {
        sfProcessHost "webserver1.hpl.hp.com;"
        port ATTRIB commonPort;
    }
    ws2 extends webServerTemplate {
        sfProcessHost "webserver2.hpl.hp.com;"
        port ATTRIB commonPort;
        useDB LAZY ATTRIB db;
    }
    db extends dbTemplate {
        userTable: rows 6;
    }
}
```
A lifecycle manager may be attached to any piece of configuration data (e.g., a compound configuration). This extends the notion of a lifecycle manager for a single component.

A lifecycle manager is an instance of a Java class (defined by the *sfClass* attribute).

A lifecycle manager for a compound configuration is responsible for the coordination and phasing of actions for its components (e.g., sequential, parallel, etc.). This extends to nested groups.

```java
webServer extends {
    port 80;
    // other generic
    // web server data
}
jetty extends {
    sfClass "org.smartfrog.jetty.Jetty";
    // other jetty specific data
}
apache extends {
    sfClass "org.smartfrog.jetty.Jetty";
    // other apache specific data
}
myJettyServer extends webServer, jetty;
myApacheServer extends webServer, apache;
```

The *sfClass* attribute specifies the class of a lifecycle manager.
A lifecycle manager may be attached to any piece of configuration data (e.g., a compound configuration). This extends the notion of a lifecycle manager for a single component.

A lifecycle manager is an instance of a Java class (defined by the `sfClass` attribute).

A lifecycle manager for a compound configuration is responsible for the coordination and phasing of actions for its components (e.g., sequential, parallel, etc.). This extends to nested groups.

```java
webServer extends {
    port 80;
    // other generic
    // web server data }

jetty extends {
    sfClass "org.smartfrog.jetty.Jetty";
    // other jetty specific data

apache extends {
    sfClass "org.smartfrog.jetty.Jetty";
    // other apache specific data
}

myJettyServer extends webServer, jetty;
myApacheServer extends webServer, apache;

system1 extends Compound {
    // shared fate
    server1 extends webServer;
    server2 extends webServer;
}
```

The `sfClass` attribute specifies the class of a lifecycle manager.
A lifecycle manager may be attached to any piece of configuration data (e.g., a compound configuration). This extends the notion of a lifecycle manager for a single component.

A lifecycle manager is an instance of a Java class (defined by the `sfClass` attribute).

A lifecycle manager for a compound configuration is responsible for the coordination and phasing of actions for its components (e.g., sequential, parallel, etc.). This extends to nested groups.

**SmartFrog configuration lifecycle**

webServer extends {
    port 80;
    // other generic
    // web server data }

jetty extends {
    sfClass "org.smartfrog.jetty.Jetty";
    // other jetty specific data

apache extends {
    sfClass "org.smartfrog.jetty.Jetty";
    // other apache specific data
}

myJettyServer extends webServer, jetty;
myApacheServer extends webServer, apache;

system2 extends Parallel { // independent fate
    server1 extends webServer;
    server2 extends webServer; }
The SmartFrog runtime system
SmartFrog Summary

✦ Strengths

A highly flexible framework
  Can be easily modified/extended (component-based)
  Accommodates legacy components through wrapping techniques

Scales well
  Loosely coupled workflow engine

Secure deployment
  Based on PKI

✦ Limitations

No organised repository

No formal or conceptual base for language and component model

Language lacks higher-order constructions (parameterized deployment)
Managing package-based software distributions

✶ EDOS

Environment for the development & Distribution of Open Source Software
A collaborative research project funded under the European Sixth Framework
A formal statement and thorough analysis of installation and upgrade problems
A set of tools for safe and efficient management of free and open source software

✶ Nix

A research project, University of Utrecht, NL
A framework for organising component repositories, allowing various deployment policies
Safe, purely functional deployment
Managing the distribution of Free and Open Source Software (FOSS)

To put some order in the “FOSS bazaar”,

a new actor: the distribution editor

A basic deployment unit: the *package*

A tool for managing the package lifecycle:

the package manager

The role of the distribution editor

Tracking source evolution

Integrating and testing

Distributing

Upstream software providers

Distribution editor

packages

End users
Package-based distribution: the EDOS view (2)

- Set of files
  - Configuration files
- Set of valued *meta-information*
  - Inter-package relationships
- Executable *configuration scripts*

A package
What is in a package?

A set of files (binaries, data, documentation)
- Configuration files have a special role
  (to be locally customised)

A set of meta-information
- Identification, version, description
- Inter-package relationships
  (dependencies, conflicts)

Executable configuration scripts
- To be executed at installation or upgrade
- May involve local files on the installation machine (not part of the package)
Package-based distribution: the EDOS view (3)
Managing relationships between packages

Depends
- Specifies packages (including version numbers) that *must be present* to make the current package functional

Conflicts
- Specifies packages that *cannot coexist* with the current package

Pre-Depends
- Specifies packages that *must already be present* to successfully deploy the current package
Managing relationships between packages

Depends
- Specifies packages (including version numbers) that *must be present* to make the current package functional

Conflicts
- Specifies packages that *cannot coexist* with the current package

Pre-Depends
- Specifies packages that *must already be present* to successfully deploy the current package

Why is this difficult?

Typical size: 20,000 packages, 200,000 relationships

Package installability may be formulated as a boolean satisfiability problem (SAT)

Finding a combination of values that makes a Boolean formula evaluate to TRUE

Therefore, it is NP-complete in the general case!

However, it turns out to be practically tractable in most current situations
Formalizing package installability in EDOS

Deciding package installability is equivalent to boolean satisfiability (SAT)

- each package \( p \) (in version \( v \)) is denoted as a boolean variable \( p_v \)
- each version constraint (e.g., \( v > 4.0 \)) is expanded into the disjunction of the packages that satisfy that constraint, e.g., \( p_{v1} \lor p_{v2} \lor \ldots \)
- each dependency is interpreted as an implication, e.g.,
  \[
  aterm \rightarrow \text{libc6} \land (\text{libce6} \lor \text{xlibs}) \land \ldots
  \]
- each conflict between packages \( a \) and \( b \) is interpreted as the formula \( \neg (a \land b) \)

Then a package \( p_v \) is installable iff there exists a boolean assignment that makes \( p_v \) TRUE and satisfies the conjunction of all the logical implications introduced by the dependencies and conflicts.
EDOS summary

- A formalisation of the package dependency problem
- A set of tools for the distribution editors
  
  Not visible to the user
  
  About 110 K lines of code in OCaml
  
  Checker for package installability
  
  Environment for repository inspection
  
  Parser/converter between package list formats
  
  Used by distribution editors: Debian, Mandriva, ...

- A follow-on project: Mancoosi
  
  Utilities for the user
Introducing Nix

Nix is a safe and flexible package management system
Safe: guarantees that all dependencies are satisfied
Flexible: unconstrained choice of deployment policies

Nix consists of
A store: repository for components (packages)
  Each component has a closure (the set of components on which it depends)
A (functional) language for describing build actions (derivations)
  Derivation expressions are interpreted

Origin
Academic project (Eelco Dolstra’s PhD thesis)
Univ. Utrecht, now Univ. Delft (NL)

The Nix store

- Unique names are built by hashing all inputs involved in building component.
- Dependencies are in terms of store paths (unique names) rather than of individual files.
- Arrows show dependencies.

(a) Organization of the store

(b) Closure value for subversion
How Nix works

Example: derivation value for subversion.

This information is used to determine the closure value shown on the previous slide.

It includes both a deployment description and the program (shell scripts) of some of the deployment tasks.

It is not intended to be written by hand, but to be generated from a higher level description.

It is used as input for performing the actual build.

Nix Summary

✩ Strengths

A purely functional system

A language for expressing derivations (build actions)
No side effects
A configuration does not change once it has been built

Allows multiple versions of a package

Upgrading/uninstalling an application cannot break another one

Atomic upgrade/rollback

Allows both source code and binary components

✩ Limitations

No experience yet with distributed systems
Not compliant with Unix Standards Base
Configuration and deployment summary

✦ Achievements
  The importance of configuration and deployment is recognized
  Systematic architecture-based approaches are being developed
    (and find their way into products)
  Formal methods are emerging, with some successful results

✦ Problems
  Lack of standards

✦ Some current research directions
  Using Model Driven Architecture
  Investigating reconfigurable architectures
    (described by dynamic ADLs)
Self-repair

- **Motivation**
  Maintain the system’s availability in the face of failures

- **Goal**
  Suppress or minimize the (user perceived) effects of a failure

- **Problems**
  Many failures (specially in communication) do not follow the fail-stop mode
  Tracing the precise location of a software failure may be difficult
  Restoring state is a complex issue

- **Approaches**
  Relate failure to system structure: architecture-based approach (see case study)
  Reduce recovery time
    - Early detection
    - Fast restoration (example: Micro-reboot, after fine-grained location)
  Consider degraded mode operation (not all failures are fatal)
    - Performability studies (fault injection, etc.)
Case study
Jade, an experiment in architecture-based self-management

* The Jade project
  Developed by research team Sardes (Univ. of Grenoble and INRIA, 2003-2009)
  A framework based on reflective components
  Experiments in various aspects of autonomic computing (configuration, performance, security, fault tolerance)
  Targeted to medium to large size clusters for Internet services
  One industrial application (with Bull)
  Site: http://sardes.inrialpes.fr/jade.html

Recent publications:

The following presentation is mainly based on the last paper
Thanks to the authors
Fractal, a reflective component model

Main features
- A general component model, allows hierarchical composition and sharing
- Three sorts of interfaces: provided, required, and control (meta—level)
- Components are run time structures
- High—level architectural description through an ADL

Diagram:
- A composite component
- A primitive component
- Required interface
- Provided interface
- Meta-data
Fractals, a reflective component model

- **Main features**
  - A general component model, allowing hierarchical composition and sharing.
  - Three sorts of interfaces: provided, required, and control (meta—level).
  - Components are runtime structures.
  - High—level architectural description through an ADL.

- **The meta—level interface**
  - Attribute controller: read/modify the state variables.
  - Life cycle controller: start, stop.
  - Binding controller: manages connections.
  - Contents controller: manages included components.
  - This list is optional and extensible.

Control interfaces

A composite component

required interface

provided interface

A primitive component

Fractal, a reflective component model

- **Main features**
  - A general component model, allows hierarchical composition and sharing.
  - Three sorts of interfaces: provided, required, and control (meta—level).
  - Components are run time structures.
  - High—level architectural description through an ADL.

- **The meta—level interface**
  - Attribute controller: read/modify the state variables.
  - Life cycle controller: start, stop.
  - Binding controller: manages connections.
  - Contents controller: manages included components.
  - This list is optional and extensible.

Control interfaces

A composite component

required interface

provided interface

A primitive component

Fractal, a reflective component model

- **Main features**
  - A general component model, allows hierarchical composition and sharing.
  - Three sorts of interfaces: provided, required, and control (meta—level).
  - Components are run time structures.
  - High—level architectural description through an ADL.

- **The meta—level interface**
  - Attribute controller: read/modify the state variables.
  - Life cycle controller: start, stop.
  - Binding controller: manages connections.
  - Contents controller: manages included components.
  - This list is optional and extensible.

Control interfaces

A composite component

required interface

provided interface

A primitive component
Fractal ADL

A J2EE 3-tier application
Fractal ADL

A J2EE 3-tier application

```xml
<!-- J2EE ARCHITECTURE -->
<component name="MyJ2EE">
  definition="fr.jade.resource.j2ee.J2eeResourceType">
  <!-- APACHE -->
  <component name="apache1" definition="fr.jade.resource.j2ee.apache.ApacheResourceType">
    <attributes>
      <attribute name="resourceName" value="apache"/>
      <attribute name="dirLocal" value="/tmp/j2ee"/>
      <attribute name="user" value="admin"/>
      <attribute name="group" value="admin"/>
      <attribute name="port" value="8081"/>
      <attribute name="serverAdmin" value="jade_admin@inrialpes.fr"/>
    </attributes>
    <virtual-node name="node1"/>
    <packages>
      <package name="Apache HTTP server v1.3.29 (linux x86)"/>
      <package name="Apache Wrapper"/>
    </packages>
  </component>
</component>
```
Fractal ADL

A J2EE 3-tier application

S. Krakowiak

BCS, Dec. 2009
Fractal ADL

A J2EE 3-tier application

<!-- MYSQL -->
<!-- ================================== -->
<component name="mysql"
definition="fr.jade.resource.j2ee.mysql.MysqlResourceType">
  <attributes>
    <attribute name="resourceName" value="mysql"/>
    <attribute name="dirLocal" value="/tmp/j2ee"/>
    <attribute name="user" value="jlegrand"/>
  </attributes>
</component>
<!-- ---------------------------------- -->
<!-- BINDINGS -->
<!-- ---------------------------------- -->
<binding client="apache.worker1" server="tomcat1.resource"/>
<binding client="apache.worker2" server="tomcat2.resource"/>
<binding client="tomcat1.jdbc" server="mysql.resource"/>
<binding client="tomcat2.jdbc" server="mysql.resource"/>
<virtual-node name="node1"/>
</definition>
Both the managed system and Jade itself are organized as an assembly of Fractal components.

To manage legacy systems, one needs to wrap them into Fractal components.

The architecture of the managed system is described in Fractal ADL.
The Jade self-repair service

**Assumptions**

- The managed system runs on a cluster of nodes (with a pool of free nodes)
- In this version, only node failures (fail-stop) are considered

**Objectives**

- To provide self-repair for the managed system
- To provide self-repair for the self-repair service (self-self-repair)
Self-repair principles

✧ Repair policy
- Identify failed components and get their architectural state
- Substitute failed components by new ones and restore their architectural state
- Architectural state: the state captured in the meta-data

Diagram:
- Repair service
- Managed element
- Managed element
- Managed element
- Managed element

architectural state
The meta-data of failed components are lost (e.g., connections, etc.)

The system provides meta-data checkpointing

- Checkpointing architectural state (1)

[Diagram showing meta-data checkpointing and repair service]

- The system provides meta-data checkpointing.
- Restoration architectural state.

自我修复

- ME1
- ME2

managed element

- ME1 (repaired)

- Meta-data
- Self-repair
- Managed element

- Meta-data
- Managed element
Checkpointing architectural state (2)
Checkpointing architectural state (2)

Copy of the meta-data

Checkpoint

Application layer

Meta-data

Business code

Life Cycle, Attribute, Binding
Failed components are identified by comparing the current state of the layer with the checkpointed state. The current state is maintained using usual failure detection techniques (heartbeat).
Making the self-repair system robust (1)

- **Bases of self-repair**
  - Reflective components
  - Architectural state checkpointing
  - Failure detection

- **The self-repair system itself is a single point of failure...**

- **Self-self-repair**
  - The same algorithm is applied recursively
  - This is possible since the self-repair system is structured in reflective components
  - Recursion stops at this level (no self-self-self repair...)

S. Krakowiak  BCS, Dec. 2009
Making the self-repair system robust (2)

- Apply the repair algorithm on the components of self-repair system
Making the self-repair system robust (2)

- Apply the repair algorithm on the components of self-repair system
- Conceptual view
Making the self-repair system robust (2)

- Apply the repair algorithm on the components of self-repair system

- Conceptual view

- Implementation view
Making the self-repair system robust (2)

- Apply the repair algorithm on the components of self-repair system

- Conceptual view

- Implementation view

- Mutual control of replicas
  Similar to classical process pairs (Tandem, etc.)
  Each replica works as a component
Putting it all together

The managed application
Putting it all together

The managed application
The self-repair service and the checkpoint layer
Putting it all together

The managed application
The self-repair service and the checkpoint layer
Self self-repair
The repair algorithm in action
The repair algorithm in action
The repair algorithm in action

Repair Service
- Analyze
  - Monitor
  - Execute

Checkpoint

Application Server
- Tomcat

Web Server
- Apache

Database Server
- MySQL
The repair algorithm in action

- **Repair Service**
  - Analyze
  - Monitor
  - Execute

- **Checkpoint**

- **Application Servers**
  - Tomcat

- **Web Server**
  - Apache

- **Database Server**
  - MySQL

Node failure
The repair algorithm in action
The repair algorithm in action

Repair Service
- Analyze
  - Monitor
  - Execute

Checkpoint

Application Server
- Tomcat

Web Server
- Apache

Database Server
- MySQL
The repair algorithm in action
The repair algorithm in action
The repair algorithm in action

Repair Service
- Analyze
- Monitor
- Execute

Checkpoint

newInstance(…)

Web Server
- Apache

Application Server
- Tomcat

Database Server
- MySQL
The repair algorithm in action
The repair algorithm in action
The repair algorithm in action
Self-repair summary

- **Main results**
  
  Architecture-based repair is feasible
  
  Components as units of confinement
  
  Reflection is important (inspection / reconfiguration)

- **Open issues**
  
  Efficient failure detection
  
  in time and space
  
  Handling dynamic architectures (e.g., mobile, etc.)

- **Some related work**
  
  Rainbow (Carnegie Mellon Univ.)
  
  a framework for architecture-based management
  
  PinPoint / JAGR (ROC project, Berkeley-Stanford)
  
  pinpointing software errors, repairing by micro-reboot
  
  Partial availability
  
  performability measures
Conclusion & Perspectives

- A new paradigm for systems management
  Beyond the Manager-Agent model

- Main ingredients
  Architectural system description
  Reflection
  both at component and architecture level

- Towards more formal models

- Towards a wider use of feedback control techniques
References

General


Case studies

• SmartFrog: http://www.smartfrog.org/
• EDOS: http://www.edos-project.org/ see also Mancoosi: http://www.mancoosi.org/
• Nix: http://nixos.org/
• Rainbow: http://rainbow.self-adapt.org/