

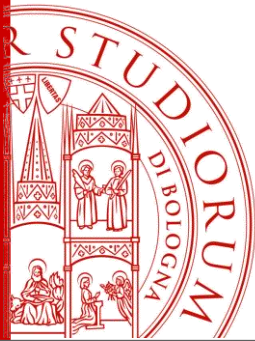
Architectural styles of adaptive systems

Dott. Francesco Poggi - fpoggi@cs.unibo.it

Corso di Architettura del Software

CdL M Informatica

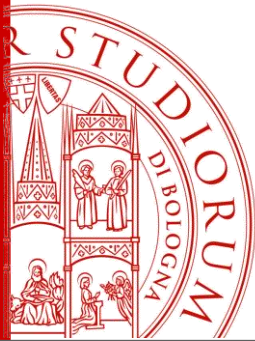
Università di Bologna



Agenda

- Self adaptive systems
- Autonomic computing
- Architectural styles for adaptive systems
- Reasoning on adaptive systems

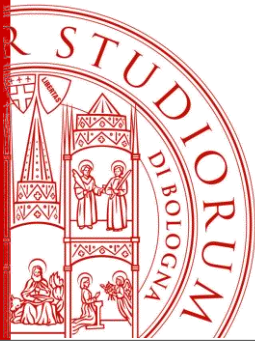
Based on a presentation by P. Ciancarini, S.Guinea and L.Baresi



Self-adaptive software

1997: defined by Laddaga in a DARPA announcement as:

“...software that evaluates its own behavior and changes when the evaluation indicates that it is not accomplishing what the software is intended to do...”



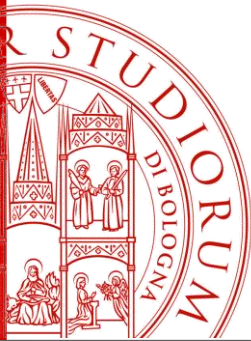
Self-adaptive software

Self adaptive software can modify its own development artifacts and configuration attributes in response to changes in

The **self**, that is, the whole body of the software, usually implemented in several layers

The **context**, that is, everything in the operating environment that affects the system's properties and behavior

The development lifecycle of an adaptive software system continues after its deployment and initial setup



Self-* properties

Self-adaptiveness

general
level

Self-configuring

Self-healing

major
level

Self-optimizing

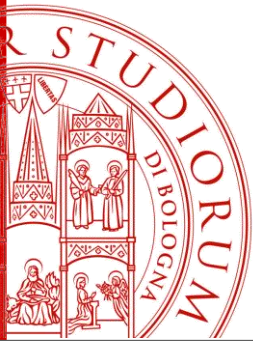
Self-protecting

Self-awareness

Context-awareness

primitive
level

M. Salehie and L. Tahvildari. Self-adaptive software:
Landscape and research challenges. TAAS 4(2): (2009)



Open vs. closed systems

Closed systems

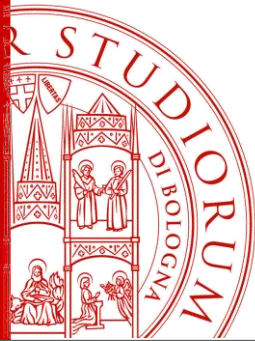
Fixed set of elements

Adaptation can only act on them to keep the system on track

Open systems

Elements can **appear** and **disappear**

Adaptation must both “**discover**” existing elements and act on them to keep the system on track



Anticipated vs. un-anticipated

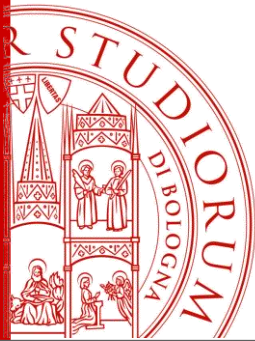
Anticipated adaption (closed adaptive)

Situations to be accommodated at run-time are known at design-time

Un-anticipated adaption (open adaptive)

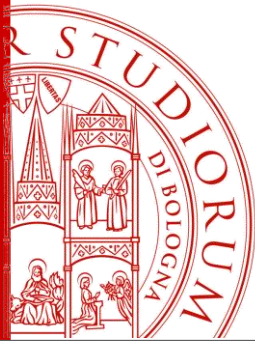
Possibilities are recognized and computed at run-time

Decisions are computed by using self-awareness and environmental context information



Externalized adaptation

- One or more models of the system are maintained at runtime
 - They are used to identify and resolve problems
- Changes are described as operations on the model
- Changes to the model affect changes onto the underlying system



Different needs

Topology

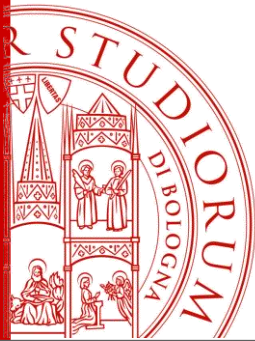
- Different interactions among the same elements
- New elements enter the system

Behavior

- Same elements start behaving differently
- New elements are injected in the system

Control

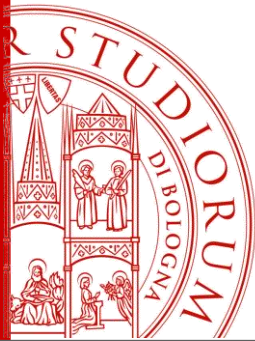
- MAPE elements must be added
- Reliability and robustness must be enforced



Adaptability

Components

- Give each component a single, clearly defined purpose
- Minimize component interdependencies
- Avoid burdening components with interaction responsibilities
- Separate processing from data
- Separate data from meta-data



Adaptability

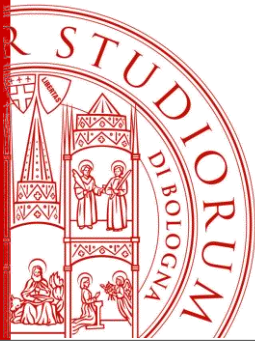
Connectors

- Give each connector a clearly defined responsibility
- Make the connector flexible
- Support connector composability
- Be aware of differences between direct and indirect dependencies

Configurations

- Leverage explicit connectors
- Try to make distribution transparent
- Use appropriate architectural styles

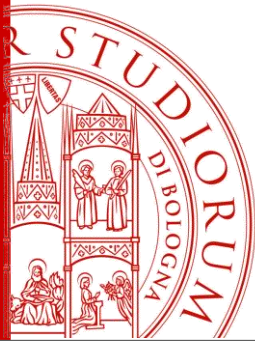
Software Architecture: Foundations, Theory, and Practice; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; 2008 John Wiley & Sons



Why Autonomic Computing?

Current software systems attributes

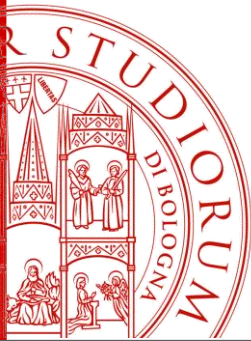
- Inherent **complexity**
- Computational **distribution**
- A set of different behaviors (**variability**)
- **Interaction** among different “components”
 - Software
 - Hardware (in general, sensors and effectors)
 - Humans
- Run-time adaptivity to unpredicted circumstances



Autonomic Computing and self-* properties

- Autonomic Computing: systems that **can manage themselves** given **high-level objectives** from administrators [Hor01]
- Self-* properties
 - Self-(re)configuration
 - Self-healing
 - Self-optimization
 - Self-protection
- Self-reconfiguration and self-*
 - Is self-reconfiguration the **mechanism** to enact the other self-* properties?



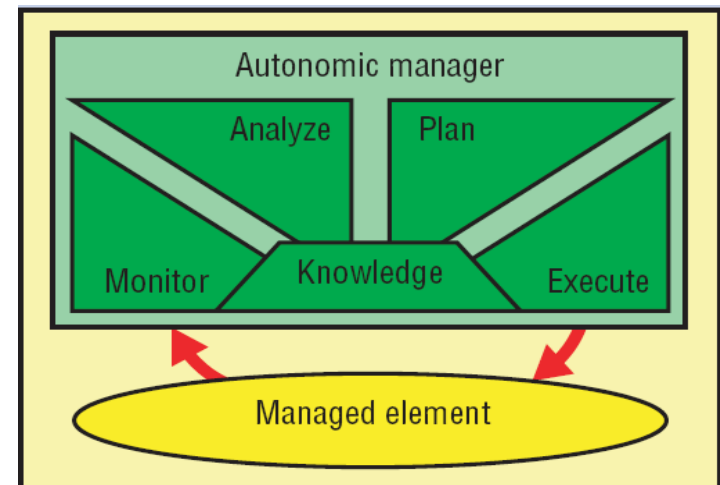


The MAPE-K paradigm

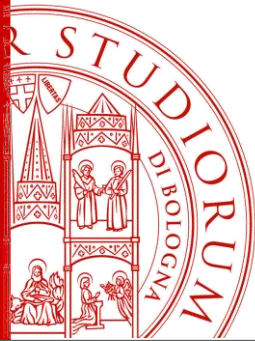
Autonomic Computing systems are made up of several **autonomic elements** [Kep03]

MAPE-K cycle:

- **Monitor**
What changed/happened?
 - **Analyze**
How to face the change
 - **Plan**
Define the actions to perform
 - **Execute**
Perform actions in response to the change
- + **Knowledge** (an high-level representation of the observed system)



Self-reconfiguration is tightly linked to MAPE-K



Key ingredients to autonomic software

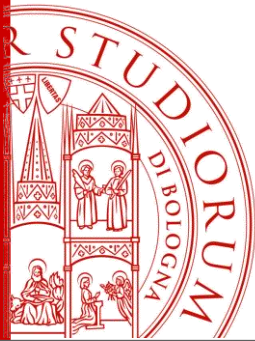
1. Architecture

- (Web)-Services or Agents to support distribution
- Adapt established infrastructures
- Guarantee reuse, extensibility, and scalability

2. Algorithms and reconfiguration mechanisms

- The following questions should be answered:
 - When reconfiguring?
 - Why reconfiguring?
 - Which behavior should be selected?
 - How to enact compensation if something fails?

3. Integration with a sw development methodology



A look at research

- **Phase 1: 1991-1999**

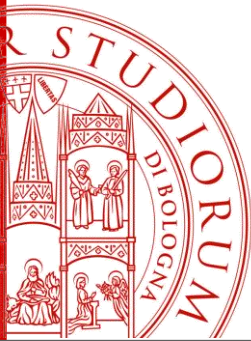
Software Architecture as a Design-time tool for Systems that Need to be Adaptive

- **Phase 2: 2000-2007**

Software Architecture for Self-Adaptive Systems

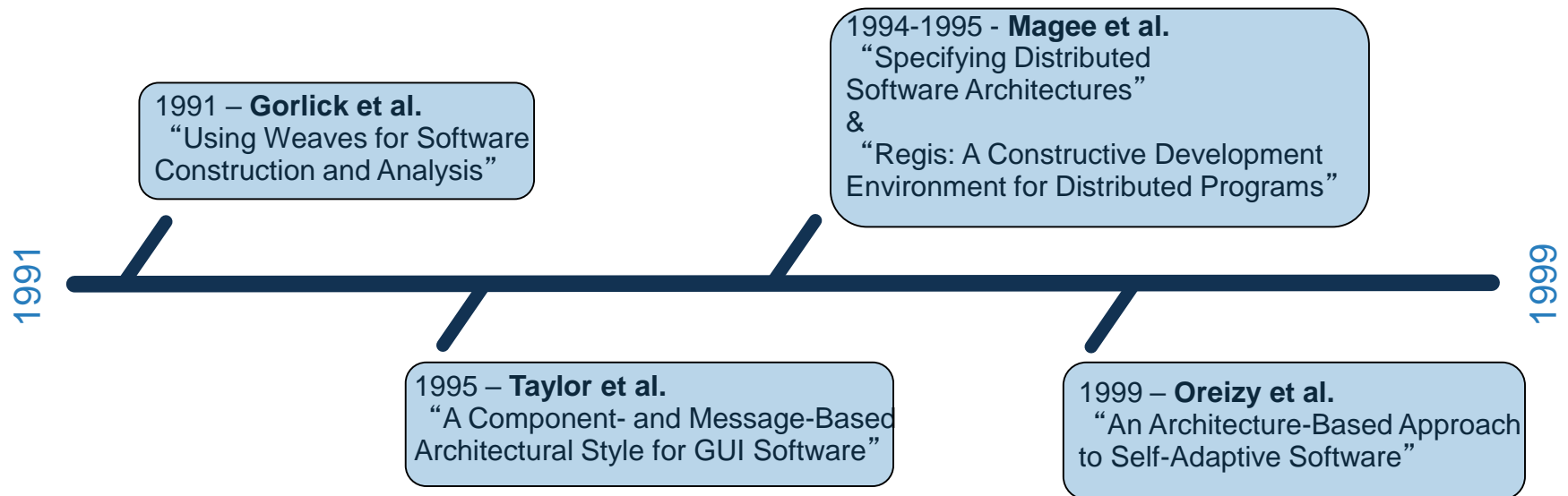
- **Phase 3: 2008-today**

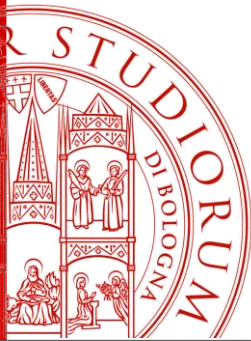
Ongoing Research on Software Architecture for Self-Adaptive Systems



A look at research Phase 1: 1991-1999

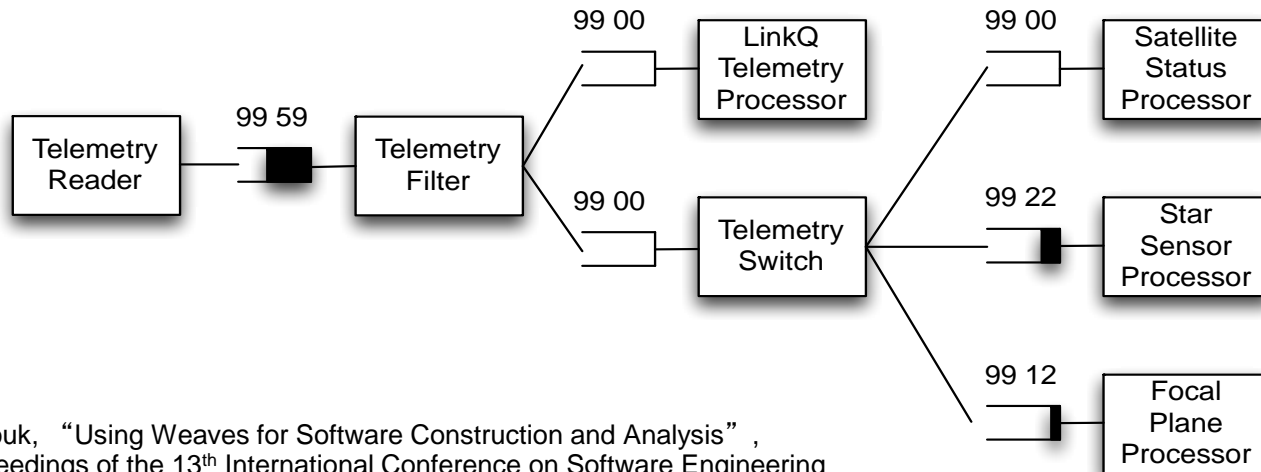
Software Architecture as a Design-time tool for Systems that Need to be Adaptive



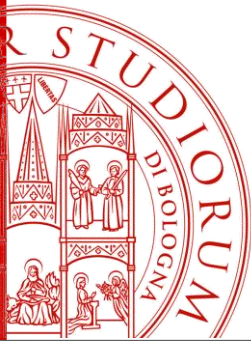


1.1 - Weaves

- Architectural style with accompanying notation
- Pipe and Filter with three differences:
 1. Tool fragments process object streams (NO byte streams)
 2. Connectors are explicitly sized queues
 3. Tool fragments can have multiple inputs and outputs



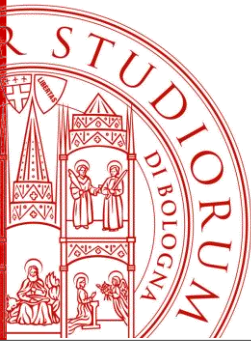
Gorlick and Razouk, "Using Weaves for Software Construction and Analysis",
ICSE '91, Proceedings of the 13th International Conference on Software Engineering



1.1 - Weaves

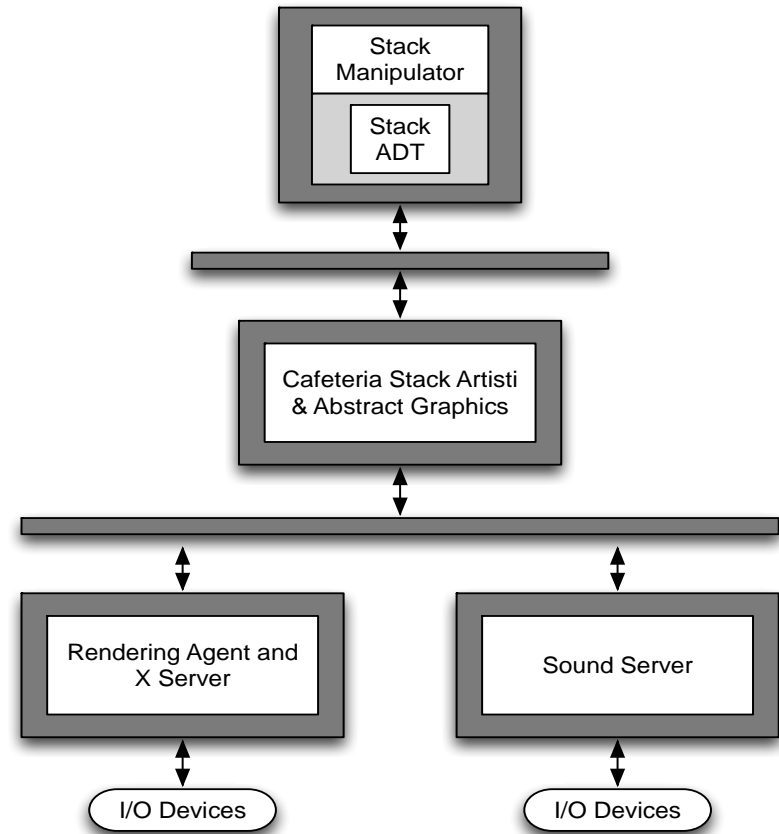
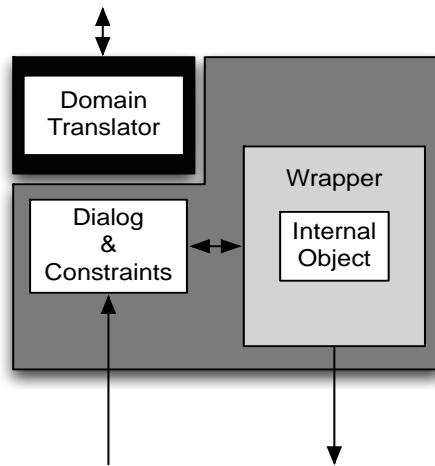
- Ports and queues are blind, type-indifferent, two-layer transport services
 - Increases the flexibility of interconnectivity
 - Allows for Location transparency
 - Specialized ports help solve output -> input incompatibilities
 - A full queue sends an error to the sender, which waits and tries again
- Tool fragments
 - Lifecycle management: start, suspend, resume, sleep, abort
- Analysis
 - Self-metric tool fragments
 - Instruments (specialized tool fragments) inserted in the weaving
 - Observers (separation between data capture and analysis)

Gorlick and Razouk, "Using Weaves for Software Construction and Analysis",
ICSE '91, Proceedings of the 13th International Conference on Software Engineering

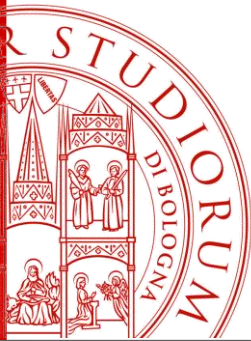


1.2 - C2

- Obtain the benefits of MVC in a distributed and heterogeneous setting
- Layered network of concurrent components hooked together by explicit message-based connectors

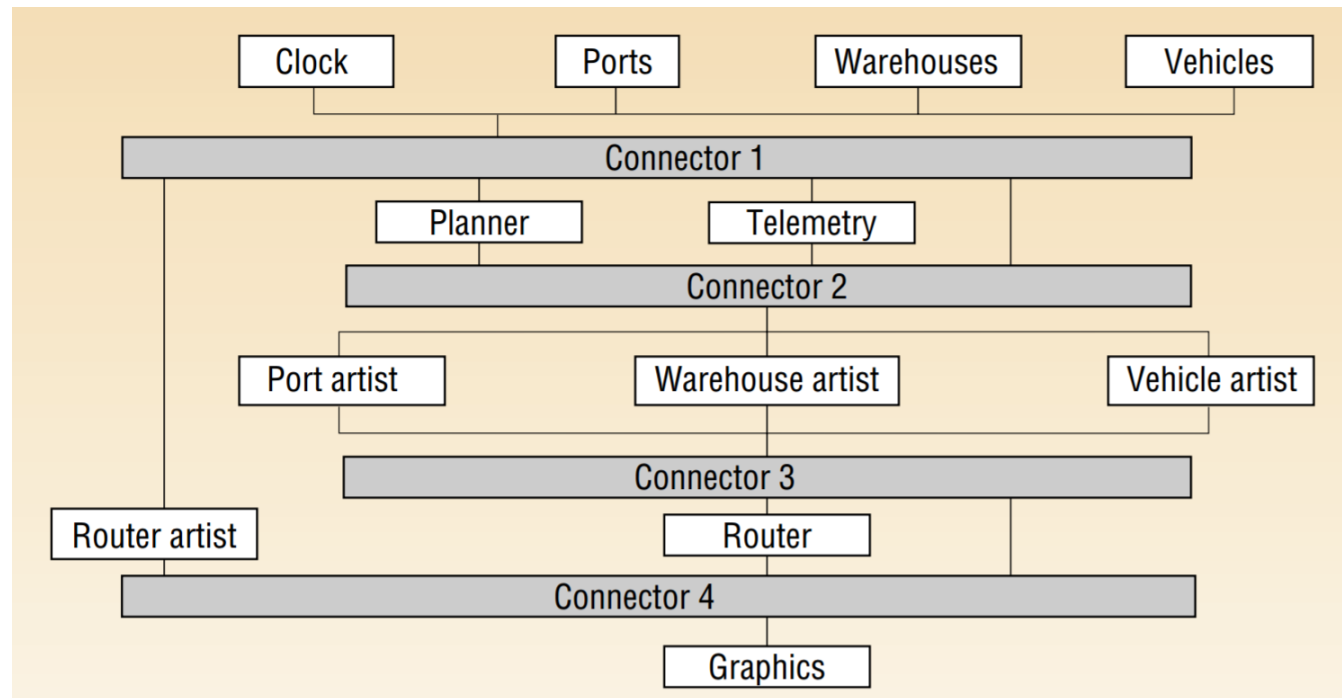


Taylor, Medvidovic, Anderson, Whitehead, and Robbins, "A Component- and Message-Based Architectural Style for GUI Software", ICSE '95 Proceedings of the 17th International Conference on Software Engineering 20

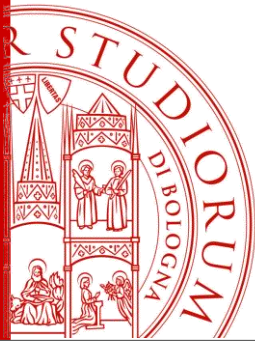


1.2 - C2

- Obtain the benefits of MVC in a distributed and heterogeneous setting
- Layered network of concurrent components hooked together by explicit message-based connectors

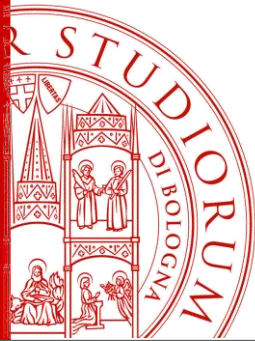


Taylor, Medvidovic, Anderson, Whitehead, and Robbins, “A Component- and Message-Based Architectural Style for GUI Software” , ICSE ‘95 Proceedings of the 17th International Conference on Software Engineering 21



1.2 - C2

- C2 composes systems as a hierarchy of concurrent components bound together by connectors (message-routing devices)
- a component is only aware of components “above” it, and is completely unaware of components residing at the same level or “beneath” it.
- a component explicitly utilizes the services of components above it by sending request message.

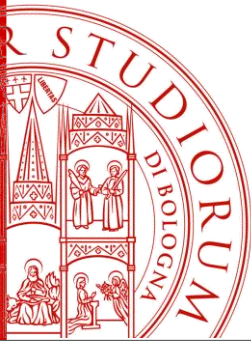


1.2 - C2

Benefits:

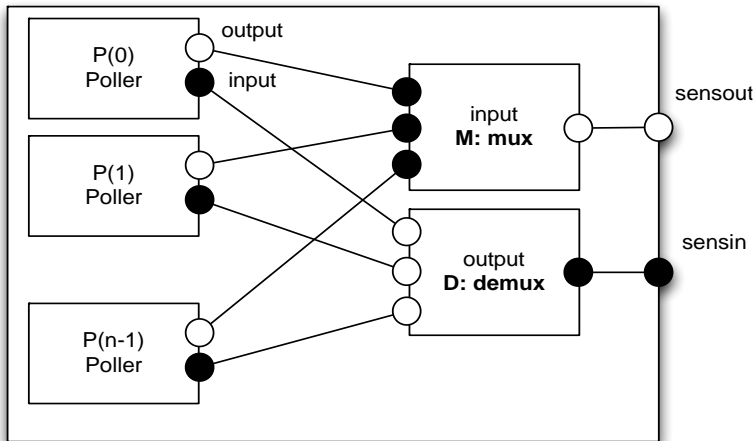
- Substrate independence
- Accommodating heterogeneity
- Can easily support for component substitution
- Design in a MVC style
- Support for concurrent components
- Support for network-distributed systems
- Smart connectors can support filtering policies

Taylor, Medvidovic, Anderson, Whitehead, and Robbins, "A Component- and Message-Based Architectural Style for GUI Software", ICSE '95 Proceedings of the 17th International Conference on Software Engineering



1.3 - Darwin and Regis

- Configuration Programming
 - Separate the description of the program structure from the programming of the functional components
 - Manage growing complexity
- Based on the notion of **provided** and **required** interfaces



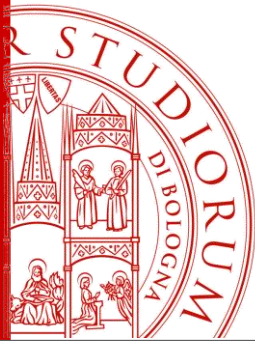
```
Component sensornet (int n){  
  provide sensin <port msg>;  
  require sensout <port msg>;
```

```
  array P[n]:poller;  
  inst  
  M:mux;  
  D:demux;
```

```
  forall i:0..n-1 {  
    inst P[i] @i+1;  
    bind  
    P[i].output -- M.input[i];  
    D.output[i] - P[i].input;  
  }
```

```
  bind  
  M.output - sensout;  
  sensin - D.input;  
}
```

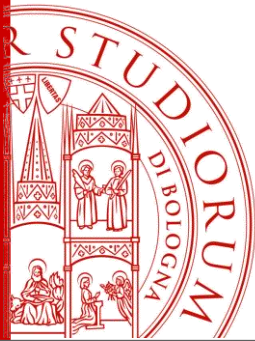
Magee, Dulay, and Kramer, "Regis: a Constructive Development Environment for Distributed Programs", Distributed Systems Engineering, Volume 1, Number 5, 1994



1.3 - Darwin and Regis

- Hierarchical configuration allows for a scalable solution
- Regis provides C++ automatically generated templates for implementing communication and processing components
- Can be modeled in pi-calculus for checking internal consistency
- Allows for Dynamic Configuration meaning the system's structure can change over time
 - Dynamic instantiation
 - Lazy instantiation

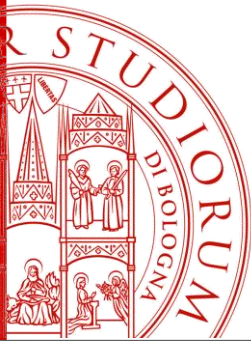
Magee, Dulay, and Kramer, "Regis: a Constructive Development Environment for Distributed Programs", Distributed Systems Engineering, Volume 1, Number 5, 1994



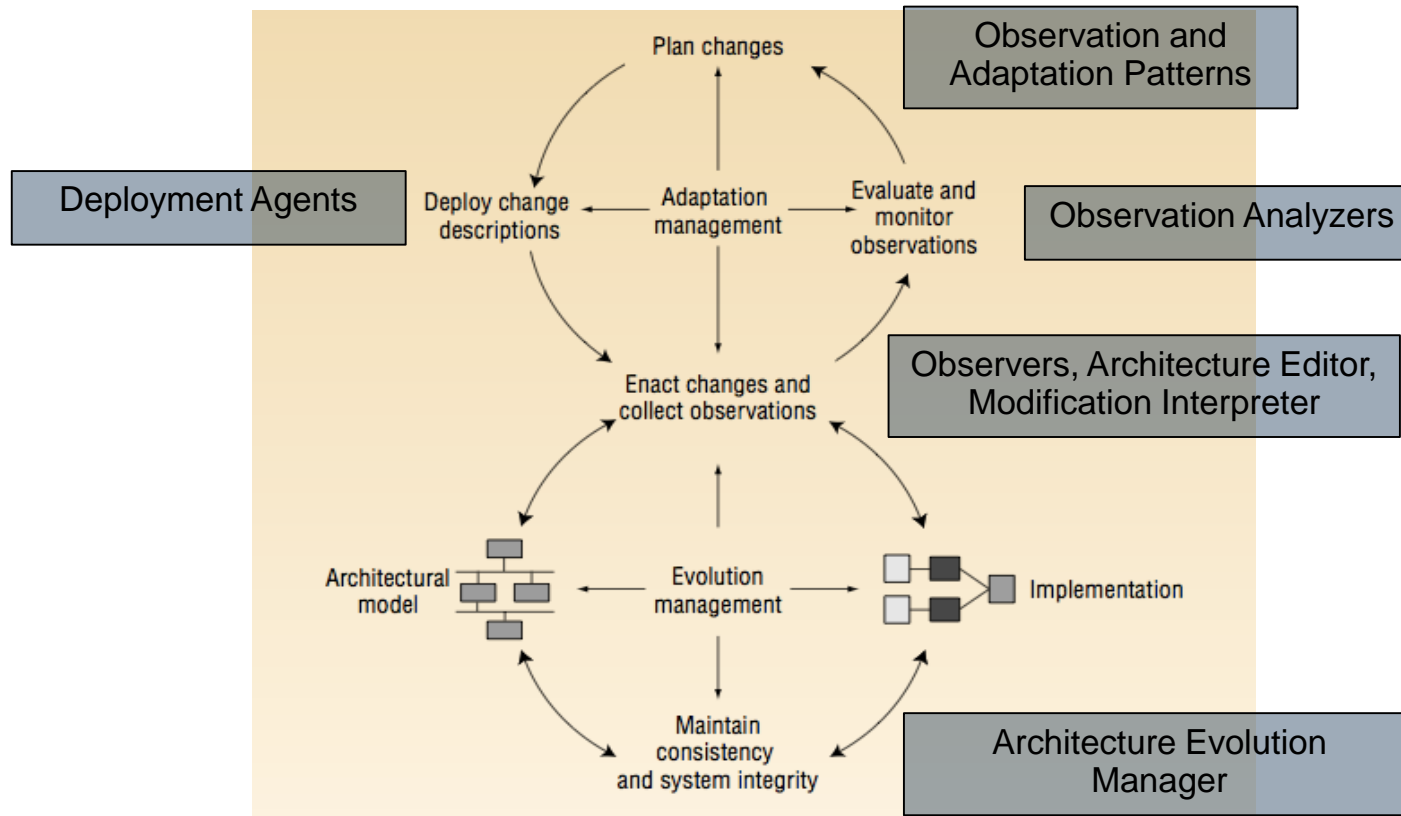
1.4 - The need for Self-Adaptation

- Unmanned Aerial Vehicles (UAVs) are used to disable an enemy airfield
 - Midway, intelligence finds that Surface-to-Air Missile (SAMs) are defending the airspace
 - Autonomous re-planning leads to two groups of UAVs (a SAM-suppression unit and a airfield suppression unit)
 - This leads to the automatic deploy of new SAM recognition algorithms
- Components are added to fielded and heterogeneous systems with no downtime
- **Required assurances: consistency, correctness, and distributed change coordination**

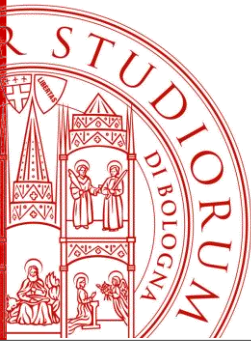
Oreizy, Gorlick, Taylor, Heimbinger, Johnson, Medvidovic, Quilici, Rosenblum, and Wolf, "An Architecture-Based Approach to Self-Adaptive Software", IEEE Intelligent Systems, Volume 14 Issue 3, May 1999



1.4. - A Methodology



Oreizy, Gorlick, Taylor, Heimbinger, Johnson, Medvidovic, Quilici, Rosenblum, and Wolf, "An Architecture-Based Approach to Self-Adaptive Software", IEEE Intelligent Systems, Volume 14 Issue 3, May 1999

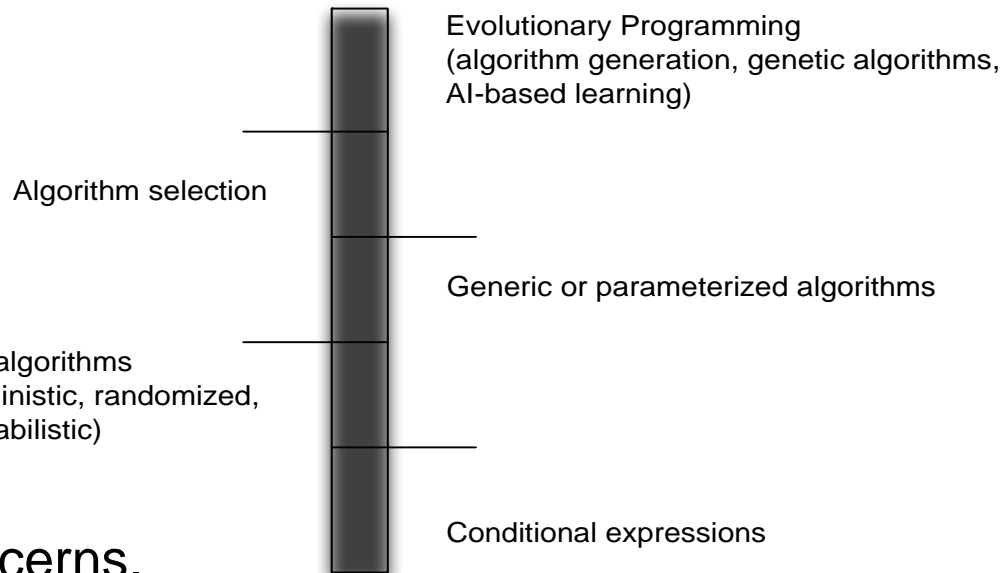


1.4 - Reasoning on Self-Adaptive Software

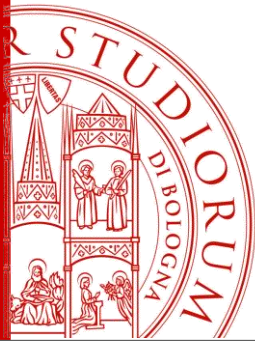
- What conditions?
- Open- or closed-adaptation?
- Type of autonomy?
- Frequencies?
- Cost-effectiveness?
- Information type and accuracy?

Approaches near the bottom select among predetermined alternatives, support **localized change**, and lack separation of concerns.

Approaches near the top support **unprecedented changes** and provide a clearer separation of software-adaptation concerns.

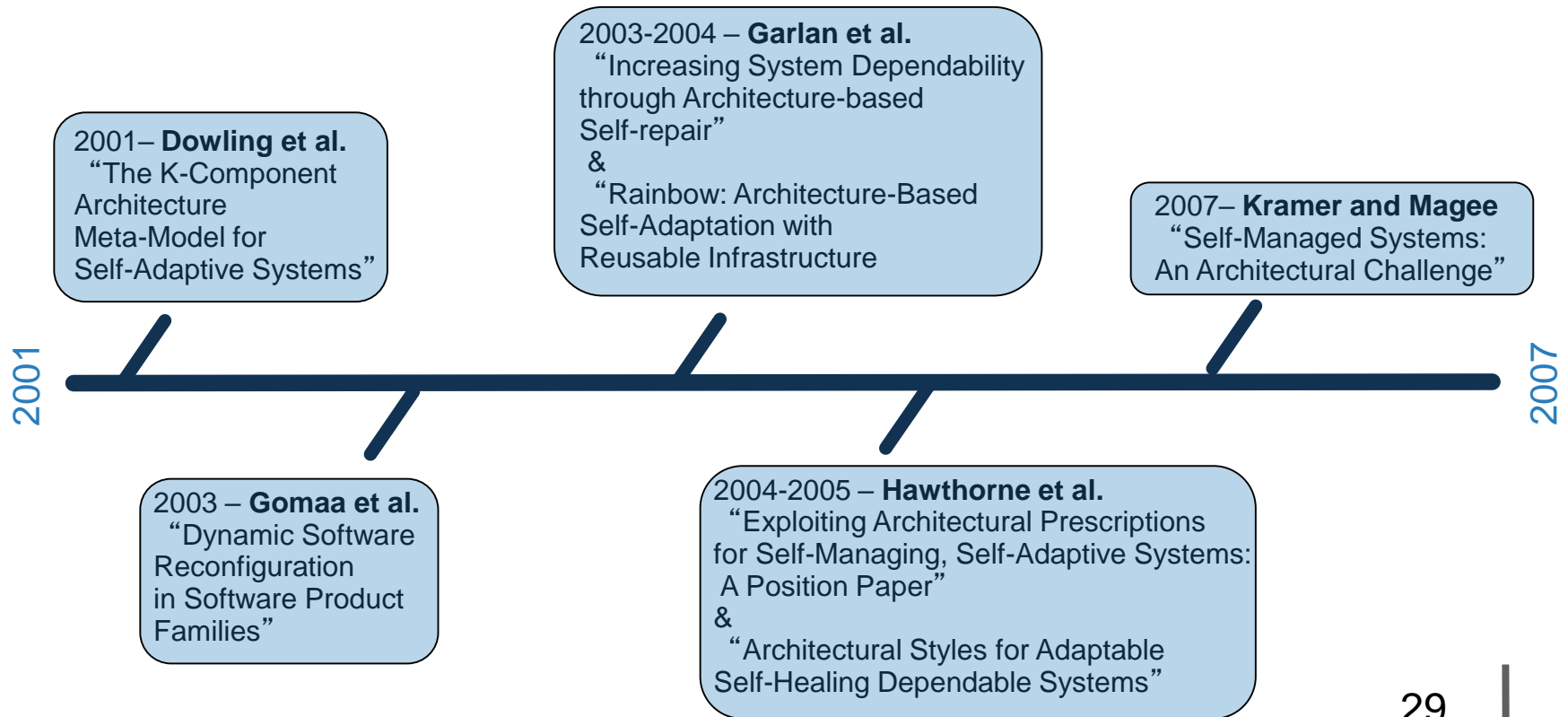


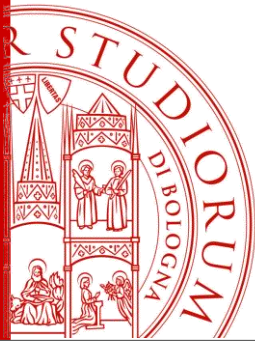
Oreizy, Gorlick, Taylor, Heimbinger, Johnson, Medvidovic, Quilici, Rosenblum, and Wolf, "An Architecture-Based Approach to Self-Adaptive Software", IEEE Intelligent Systems, Volume 14 Issue 3, May 1999



Phase 2

Software Architecture for Self-Adaptive Systems

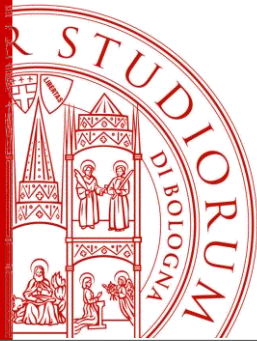




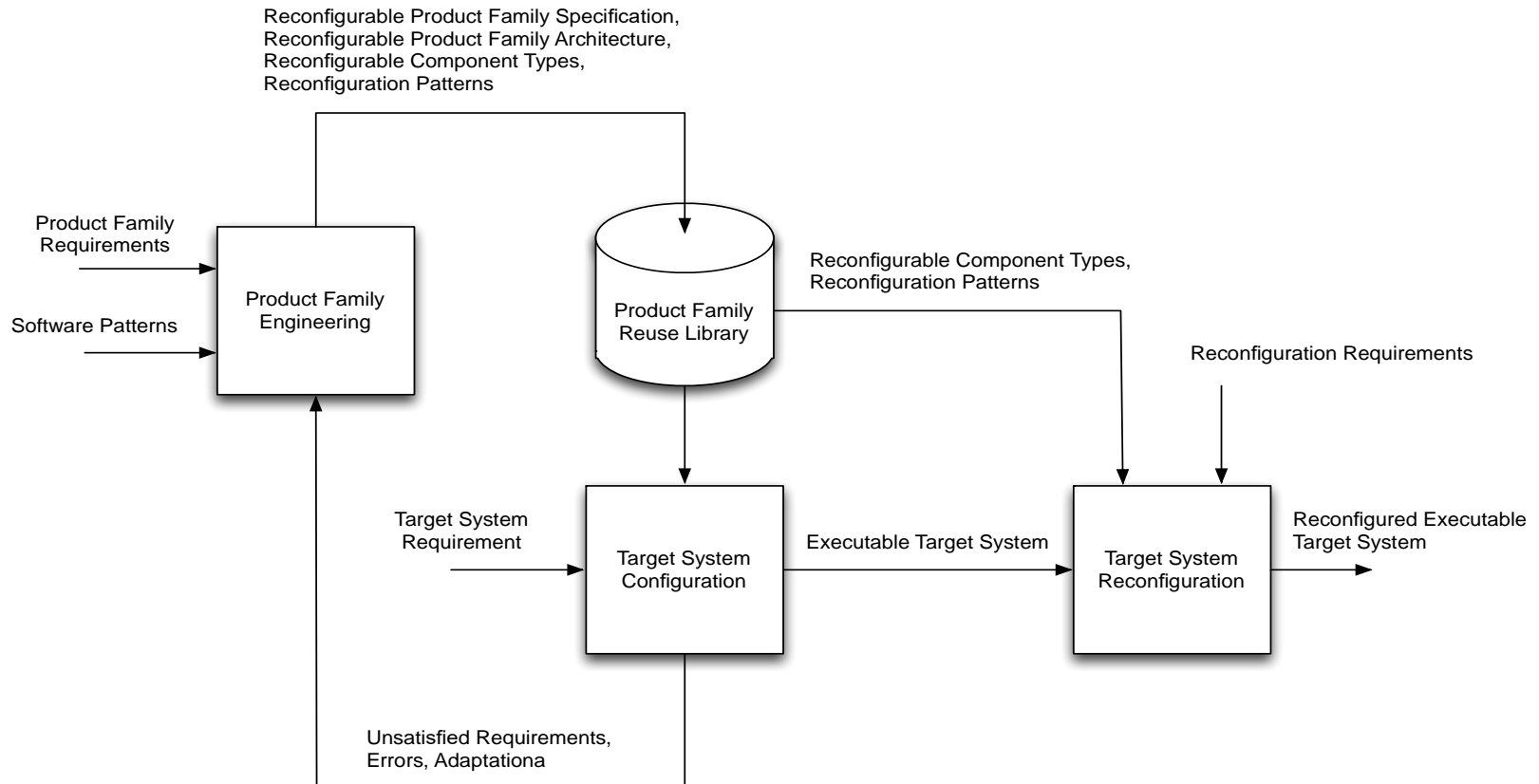
2.1 - Software Product Families

- Software Product Family
 - A Software Architecture that characterizes the similarities and variations that are allowed among the members of a product “family”.
- Software Configuration
 - Process of adapting the architecture of the product family to create the architecture of a specific product member
- **Dynamic System Reconfiguration**
 - No interference with the parts that are not affected
 - Components should complete their activities prior to reconfiguration
 - Separation of Reconfiguration and Application concerns

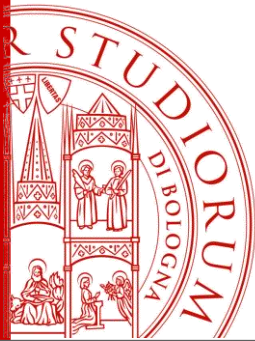
Gomaa and Hussein, “Dynamic Software Reconfiguration in Software Product Families” , Software Product-Family Engineering, 5th International Workshop, 2003



2.1 - Reconfigurable Evolutionary Product Family Life Cycle



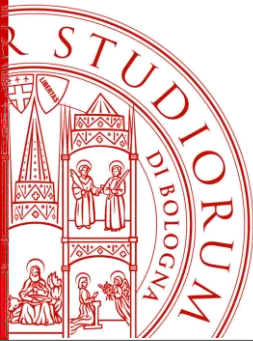
Gomaa and Hussein, "Dynamic Software Reconfiguration in Software Product Families", Software Product-Family Engineering, 5th International Workshop, 2003



2.1 - Reconfiguration Patterns

- How do components cooperate to change their configuration using reconfiguration commands?
- Each component has
 - An **operating** statechart
 - operational transactions
 - A main **reconfiguration** statechart
 - explains how the component passes through active, passivating, passive, and quiescent states during reconfiguration
 - One or more **operating with reconfiguration** statecharts
 - For handling reconfiguration events in the operating statechart
 - One or more **neighbor component state tracking** statecharts
- Everything is brought together by a **Change Management Model**

Gomaa and Hussein, "Dynamic Software Reconfiguration in Software Product Families", Software Product-Family Engineering, 5th International Workshop, 2003



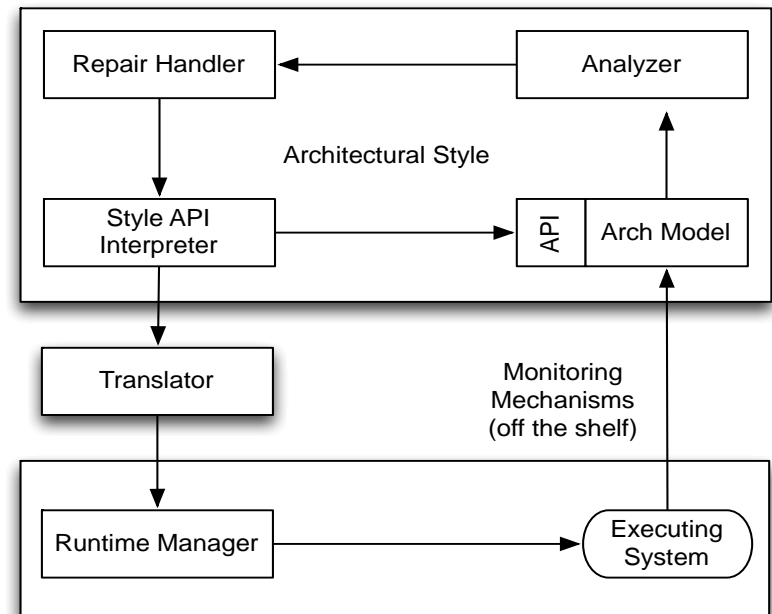
2.1 - Change Management Model

- Change Rules
 - A component can only be removed if quiescent
 - Interconnections can be unlinked if the component is quiescent with respect to those links
 - Etc.
- Change Transaction Model
 - Impacted Sets (of components)
 - Components that must be brought to quiescence
- Reconfiguration Commands
 - Passivate, checkpoint, unlink, remove, create, link, activate, restore, reactivate

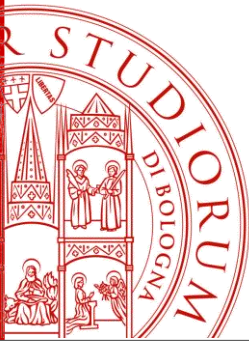
Gomaa and Hussein, "Dynamic Software Reconfiguration in Software Product Families" , Software Product-Family Engineering, 5th International Workshop, 2003

2.2 - Architecture-based Self-Repair

- Provides a generalization of architecture-based self-adaptation
- The architectural style becomes a **first-class run-time entity**
- The style determines
 - What needs to be monitored
 - What constraints need to be evaluated
 - What to do when there is a violation
 - How to perform the repair
- Augment the style with
 - Style-specific architectural operators
 - Collection of repair strategies



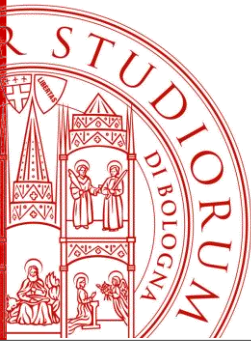
Gomaa and Hussein, "Dynamic Software Reconfiguration in Software Product Families", Software Product-Family Engineering, 5th International Workshop, 2003



2.2 - Architecture-based Self-Repair

- The generic model comprises **Components** and **Connectors** with explicit interfaces
 - **Ports** are component interfaces
 - **Roles** are connector interfaces
- Components can be further refined through **representations**
- Semantic properties are described through graph annotation
- Style types are defined using Acme, a generic ADL
- Style constraints are defined in Armani, first-order predicate logic
- A **repair strategy** determines a problem's cause and how to fix it
 - Defined as a transactional sequence of **tactics**
 - plus a policy for solving tactic conflict
- A tactic has a **pre-condition** and a **repair script**

Garlan, Cheng, and Schmerl, "Increasing System Dependability through Architecture-based Self-repair",
Architecting dependable systems



2.2 - A Web-based Server-Client System

```

Family PerformanceClientServerFam extends ClientServerFam with {
  Component Type PAClientT extends ClientT with {
    Properties {
      Requests : sequence <any>;
      ResponseTime : float;
      ServiceTime : float;
    };
  };
  ComponentType PAServerT extends ServerT with {...}
  Connector Type PALinkT extends LinkT with {
    Properties {
      DelayTime : float; };
  };
  Component Type PAServerGroupT extends ServerGroupT with {
    Properties {
      Replication : int <<default : int = 1;>>;
      Requests : sequence <any>;
      ResponseTime : float;
      ServiceTime : float;
      AvgLoad : float;
    };
    Invariant AvgLoad > minLoad; };
  Role Type PAClientRoleT extends ClientRoleT with {
    Property averageLatency : float;
    Invariant averageLatency < maxLatency;
  };
  Property maxLatency : float;
  Property minLoad : float;
};
  
```

```

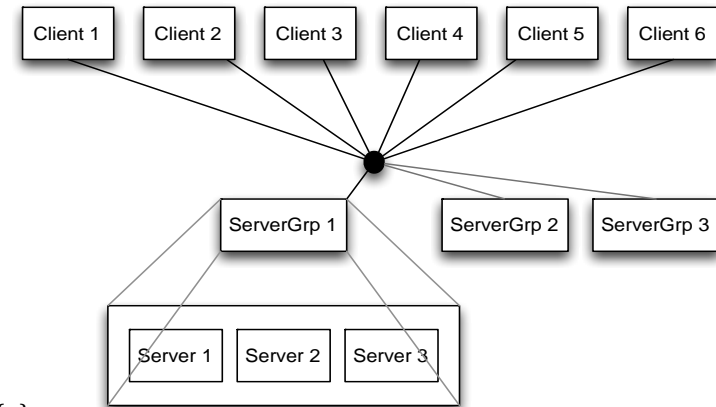
Family ClientServerFam = {
  Component Type ClientT = {...};
  Component Type ServerT = {...};
  Component Type ServerGroupT = {...};
  
```

```

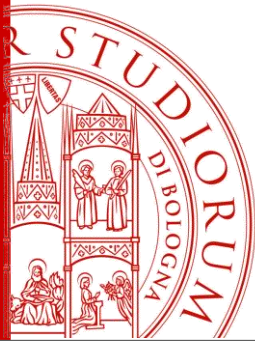
Role Type ClientRoleT = {...};
Role Type ServerRoleT = {...};
  
```

```

Connector Type LinkT = {
  invariant size(select r : role in Self.Roles | declaresType(r, ServerRoleT)) == 1;
  invariant size(select r : role in Self.Roles | declaresType(r, ClientRoleT)) >= 1;
  Role ClientRole1 : ClientRoleT;
  Role ServerRole : ServerRoleT;
};
  
```



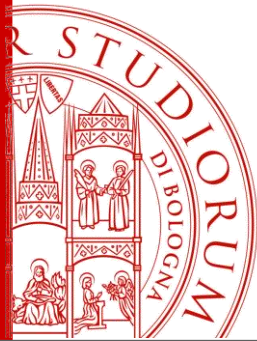
Garlan, Cheng, and Schmerl, "Increasing System Dependability through Architecture-based Self-repair", Architecting dependable systems



2.2 - Operations and Strategies

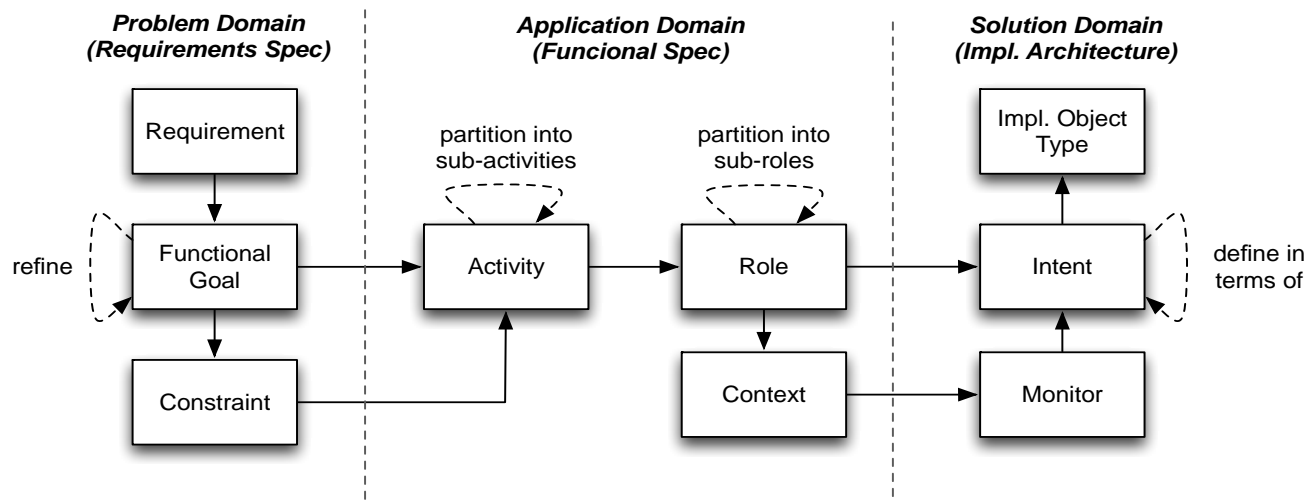
- Adaptation Operations
 - `addServer()`
 - `move(to:serverGroupT)`
 - `remove()`
 - `findGoodSGroup(cl:ClientT, bw:float)`
- Strategies
 - `fixLatency`
 - Pre-condition: averageLatency is NOT less or equal to maxLatency
 - Tactic 1: server group is overloaded -> create a new server in the group
 - Tactic 2: there is communication delay -> find the best server group and move the client-server connector to that group

Garlan, Cheng, and Schmerl, "Increasing System Dependability through Architecture-based Self-repair",
Architecting dependable systems

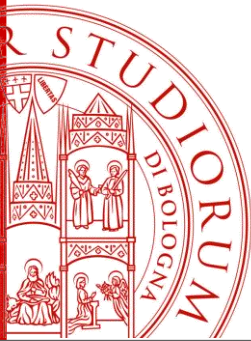


2.3 - From Requirements to Architectures

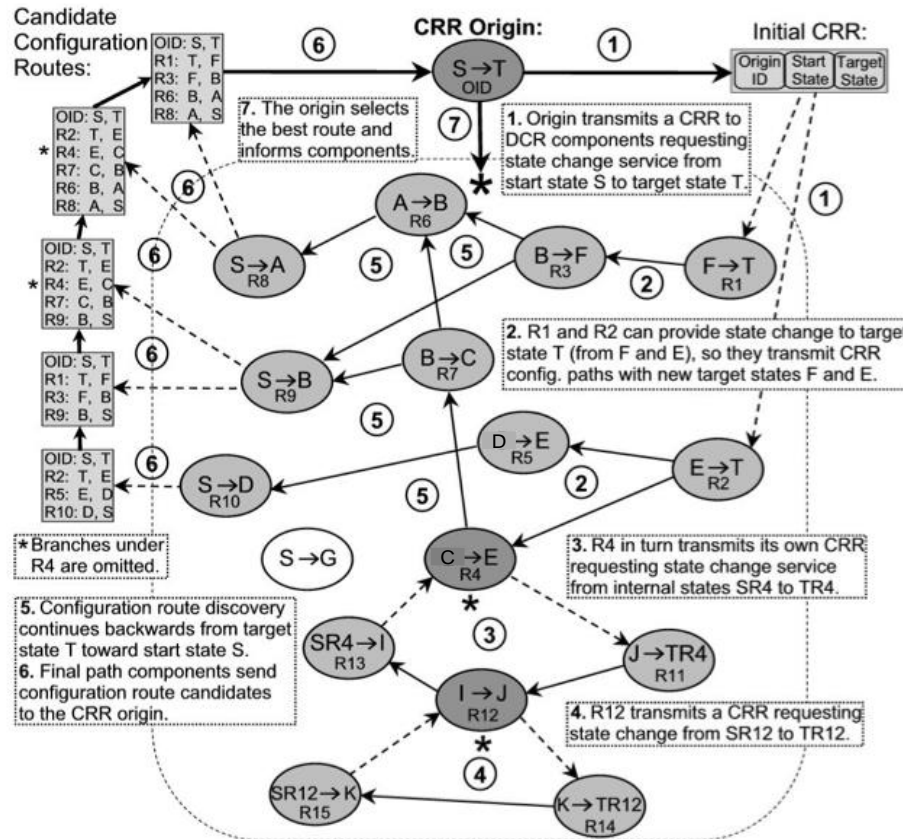
- The goal is to bridge the gap between requirements engineering and software architecture
 - **Activities** - what must be done to satisfy a functional goal
 - **Roles** - abstractions of the roles the objects play to reach a goal
 - **Intents** - capture the essence of an object's purpose and functionality
 - They provide a behavioral abstraction (state change model)



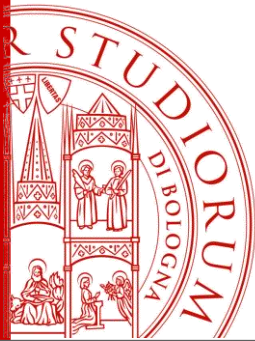
Hawthorne and Perry, "Exploiting Architectural Prescriptions for Self-Managing, Self-Adaptive Systems: A Position Paper", Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, 2004



2.3 - Distributed Configuration Routing



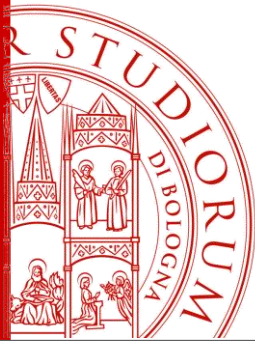
Hawthorne and Perry, "Exploiting Architectural Prescriptions for Self-Managing, Self-Adaptive Systems: A Position Paper", Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, 2004



2.4 - An Architectural Challenge

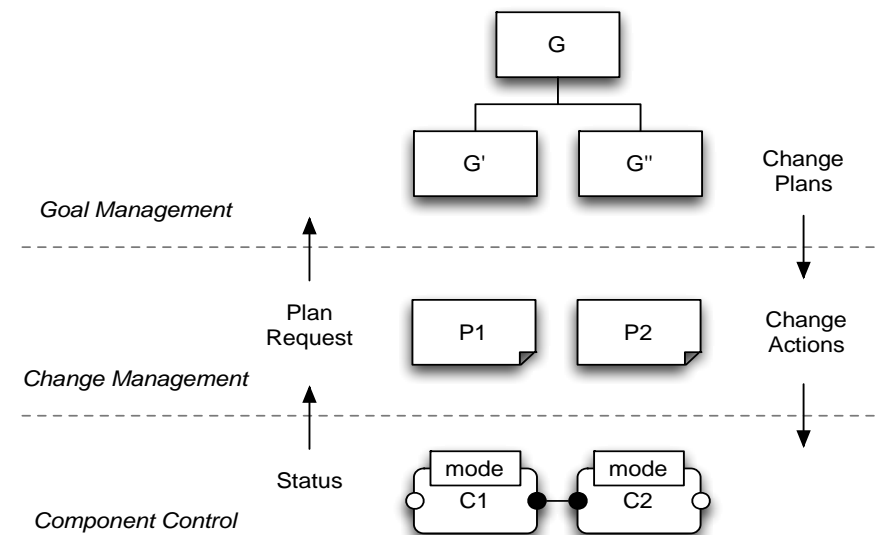
- Architecture provides the required level abstraction and generality to deal with Self-management
 - Can help with scalability
 - Build on existing work
 - Potential for an integrated approach
- The goal is to **minimize the degree of explicit management necessary for construction and subsequent evolution whilst preserving the architectural properties implied by its specification**
- They propose a three-layer architecture based on Gat' s three layer architecture for self-managing robots

Kramer and Magee, "Self-Managed Systems: an Architectural Challenge" , Future of Software Engineering 2007

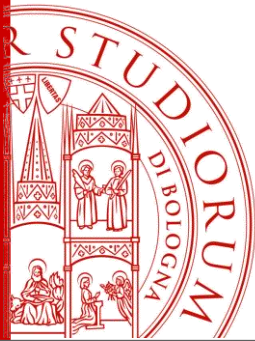


2.4 - An Architectural Challenge

- Component Control
 - How can we preserve safe application operation during change?
 - How can we ensure safety properties are never violated?
- Change Management
 - How can we deal with distribution and decentralization?
 - How can we preserve global consistency and guarantee local autonomy?
- Goal Management
 - How can we achieve goal specification that is both comprehensible and machine readable?
 - How can we decompose goals and generate operationalized plans?

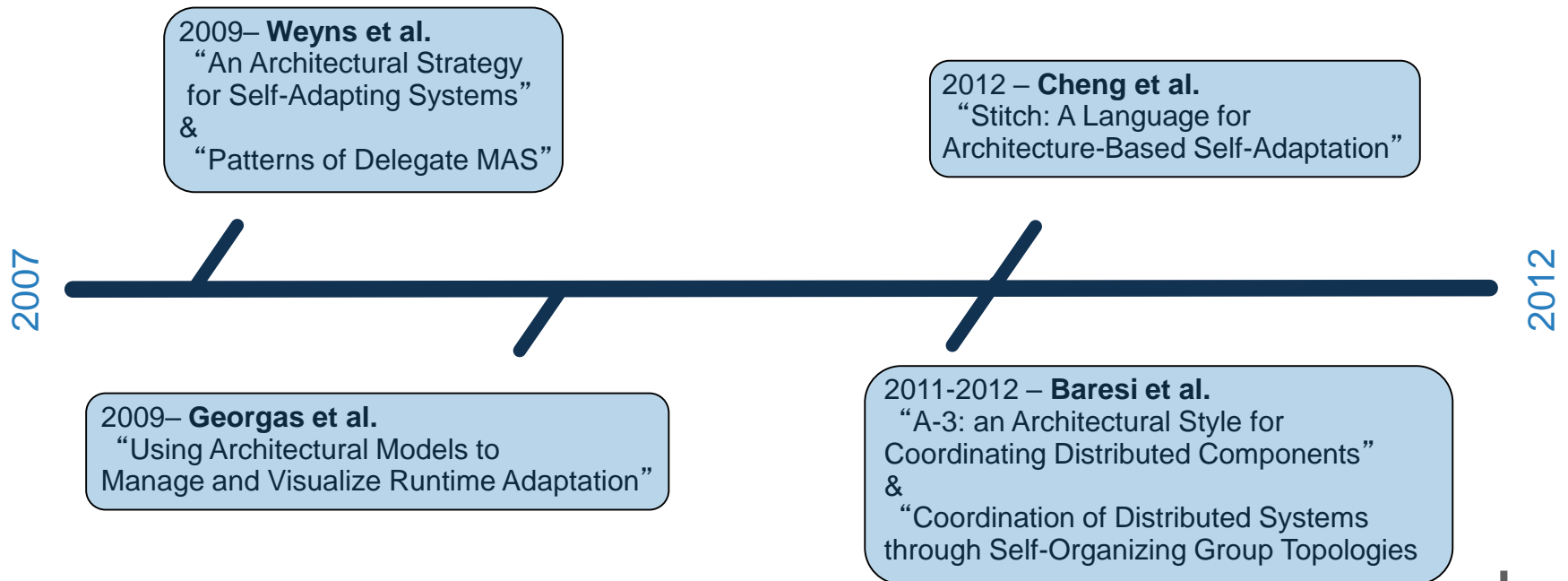


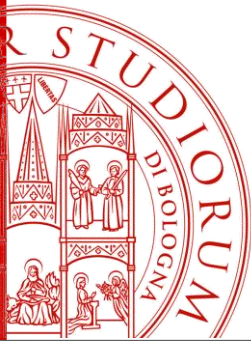
Kramer and Magee, "Self-Managed Systems: an Architectural Challenge", Future of Software Engineering 2007



Phase 3

Ongoing Research on Software Architecture for Self-Adaptive Systems

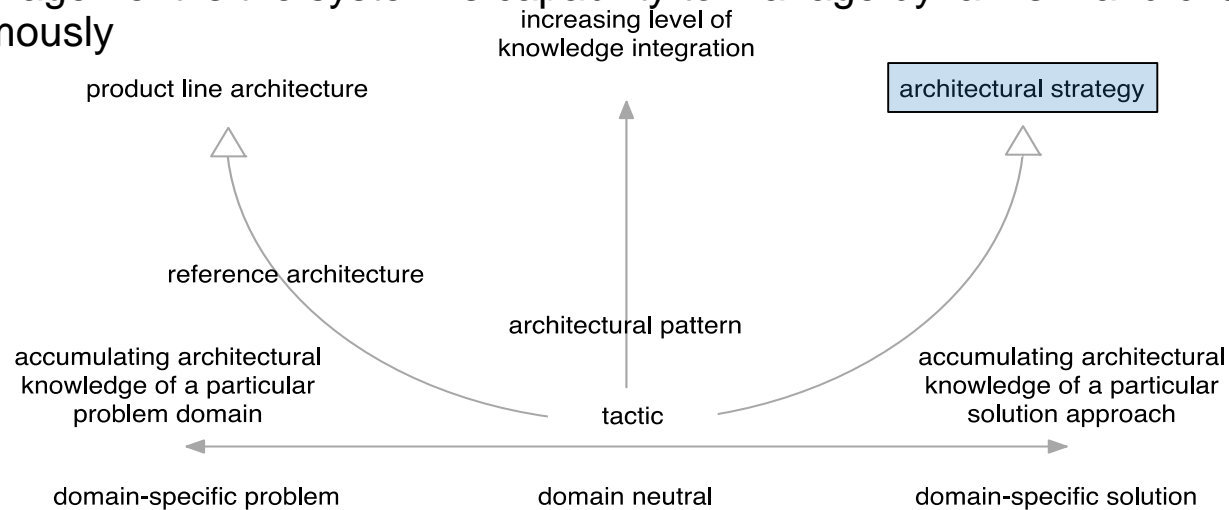




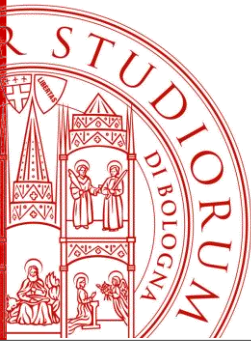
3.1 - Situated Multi-Agent Systems

The system is structured as interacting autonomous entities that are situated in an environment

- They employ the environment to share information and coordinate their behavior
- **Control is decentralized**
- Self-management is the system's capability to manage dynamism and change autonomously

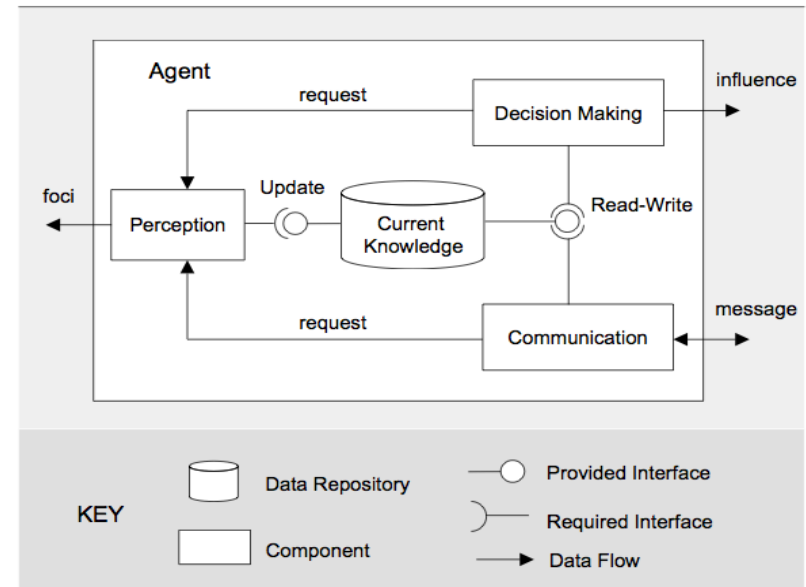


Weyns and Holvoet, "An Architectural Strategy for Self-Adapting Systems", Software Engineering for Adaptive and Self-Managing Systems, 2007

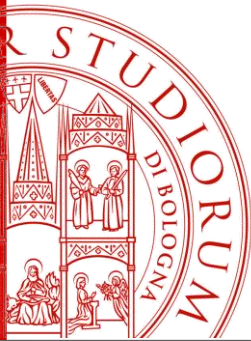


3.1 - Situated Multi-Agent Systems

- **Perception** – a filtered sensing of the environment
- **Decision Making** – action selection through the *influence-reaction* model
- **Communication** – communicative interaction with other agents
- Communication and decision making are kept separate
 - Clear separation of concerns
 - Both functions can act in parallel and proceed at different paces



Weyns and Holvoet, “An Architectural Strategy for Self-Adapting Systems” , Software Engineering for Adaptive and Self-Managing Systems, 2007



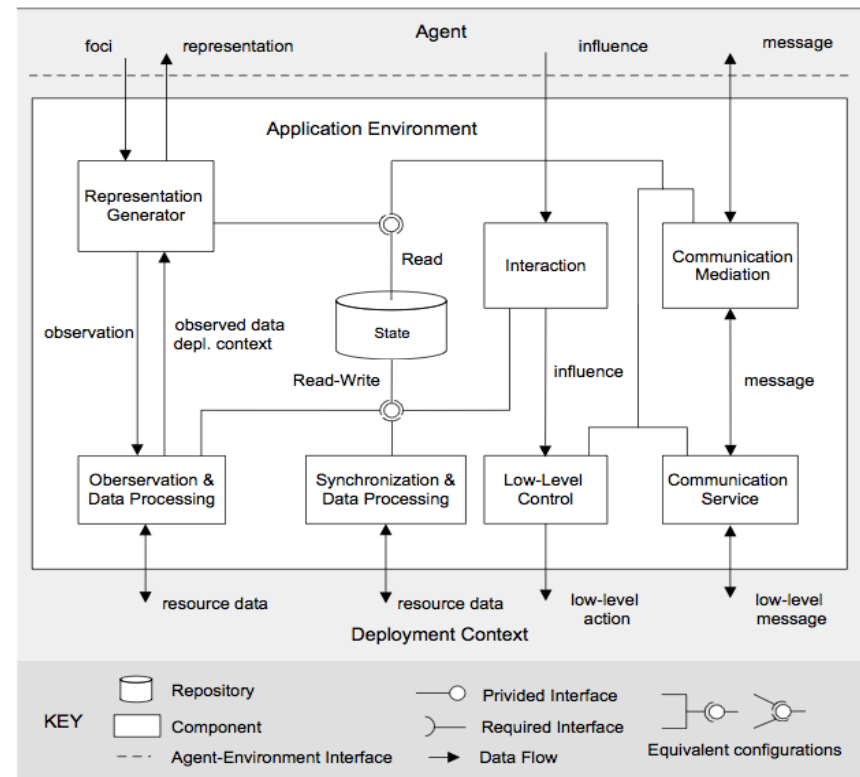
3.1 - Situated Multi-Agent Systems

Horizontal Decomposition

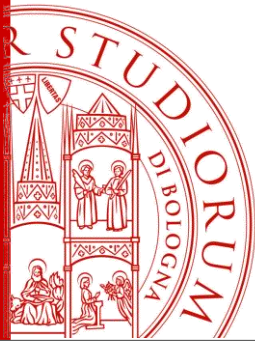
- Representation Generator
- Interaction
- Communication Mediation

Vertical Decomposition

- Observation and Data Processing
- Low-level Control
- Communication Service
- Synchronization and Data Processing



Weyns and Holvoet, "An Architectural Strategy for Self-Adapting Systems", Software Engineering for Adaptive and Self-Managing Systems, 2007



3.1 - Patterns for Delegate MAS

Three bio-inspired, light-weight, ant-like agents that assist domain agents in their coordination

The underlying communication environment is a dynamic graph topology

Problems:

- Global-to-local and local-to-global information dissemination

Information is needed for decision making

- Stability

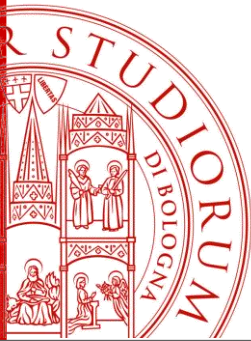
Decisions need continuous revision due to new possibilities or problems

Solutions:

- **Smart Messages**

Information retrieval is delegated to smart messages

Weyns and Holvoet, “An Architectural Strategy for Self-Adapting Systems” , Software Engineering for Adaptive and Self-Managing Systems, 2007



3.1 - Solutions

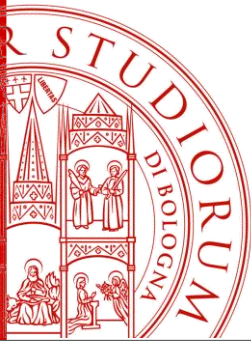
Smart messages mix **state** and **behavior**

- Behavior is executed at every node to determine:
 - How it interacts with the node
 - How it will “move” from there
 - If message cloning is required

Delegate MAS

- About effectively using a conglomerate of smart messages to solve the problem of repeated interactions
 - Specification of interactions, frequency of interactions, and aggregation and processing of the interaction results
- It is a behavior module that includes
 - A policy for creating smart messages with their own parameterised behaviors and initial states

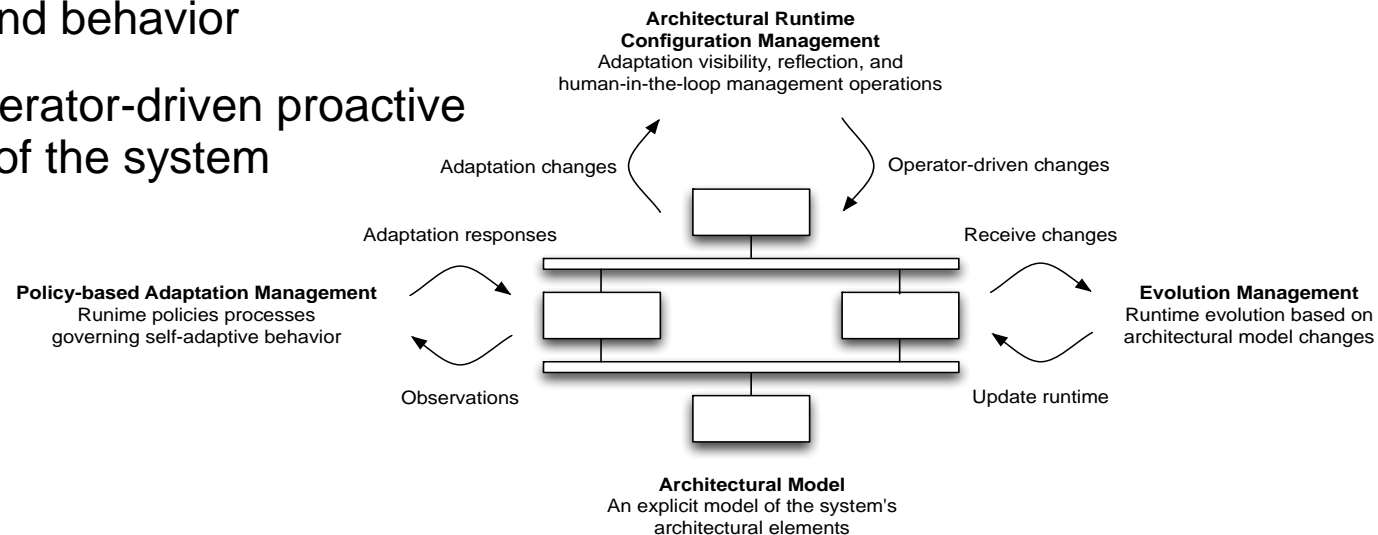
Weyns and Holvoet, “An Architectural Strategy for Self-Adapting Systems” , Software Engineering for Adaptive and Self-Managing Systems, 2007



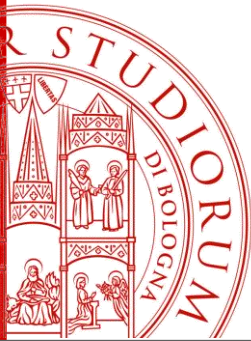
3.2 - Management and Visualization

An Operations Control Center to

- Contextualize current and past behavior with respect to the system configurations that resulted in these behaviors
- Support retroactive analyses of **historical information** about a system's composition and behavior
- Connect to operator-driven proactive management of the system



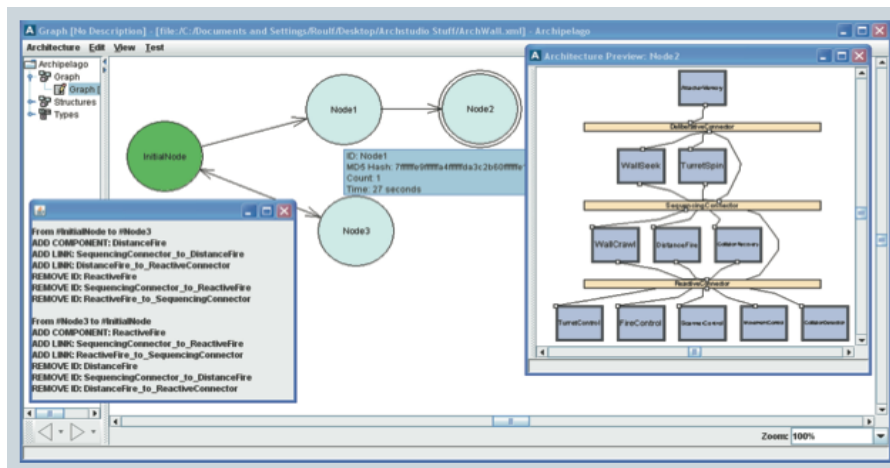
Georgas, van der Hoek, and Taylor, "Using Architectural Models to Manage and visualize Runtime Adaptation", IEEE Computer, 2009



3.2 - Management and Visualization

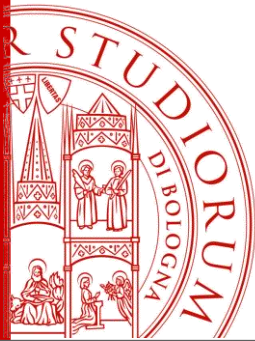
The main result is a historical graph of architectural configurations

- **Visibility** – to see what happened
- **Understandability** – to improve adaptation
- **Management** – to rollback or push the system into an existing configuration



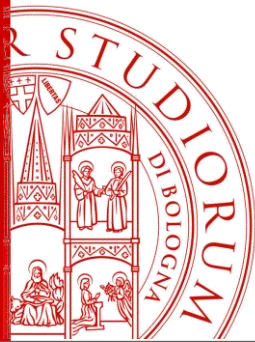
- Directed cyclic graph $G=(N, E)$
 - N is a set of nodes
 - E is a set of unidirectional edges between nodes
- Each n in N defines a specific architectural configuration
- The tool provides bidirectional **diffs** for each edge in the graph

Georgas, van der Hoek, and Taylor, “Using Architectural Models to Manage and visualize Runtime Adaptation” , IEEE Computer, 2009



3.3 - Stitch

- A language for defining and automating the execution of adaptation strategies in an architecture-based self-adaptation framework (see slides 55-58)
- Requirements for Stitch
 - Adaptation decision processes should be able to choose the next action depending on the outcome of previous ones
 - When evaluating the result of an adaptation action the language should take into account that effects could be susceptible to delay
 - Strategies should be “guarded” by activation conditions
 - Should be possible to determine the best strategy to execute if there is more than one – this must depend on the context of execution
 - Past successes or failures to adapt should contribute to the overall process
- Stitch defines repair decision trees, together with business objectives to guide the strategy selection



3.3 - Stitch

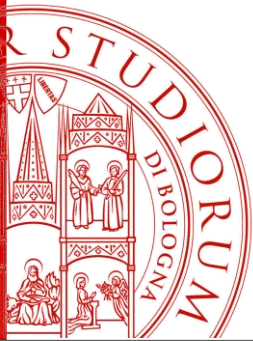
Operator - primitive unit of adaptation

- The operators are determined by the architectural style

Tactic – an abstraction that packages operators into larger units of change. A tactic contains:

- A sequence of operator calls
- Activation pre-conditions
- A definition of the effects that it is attempting to achieve
- An impact vector that specifies how it will impact the system's quality dimensions

```
tactic switchToTextualMode () {  
  condition {  
    exists c:T.ClientT in M.components | c.expRspTime >M.MAX_RSPTIME;  
  }  
  action {  
    svrs = { select s : T.ServerT | ! s.isTextualMode };  
    for (T.ServerT s : svrs) { Sys.setTextualMode(s, true); }  
  }  
  effect {  
    forall c:T.ClientT in M.components | c.expRspTime ≤ M.MAX_RSPTIME;  
    forall s:T.ServerT in M.components | s.isTextualMode ;  
  }  
}
```



3.3 - Stitch

Strategy – each step is the condition execution of a tactic

- It is a tree of **condition-action-delay** decision nodes
- Each strategy has a context-based activation condition
- Allows for the calculation of an aggregate utility function

```
define boolean styleApplies = Model.hasType(M,"ClientT")//.."ServerT";
define boolean cViolation = exists c:T.ClientT in M.components | c.expRspTime >M.MAX_RSPTIME;

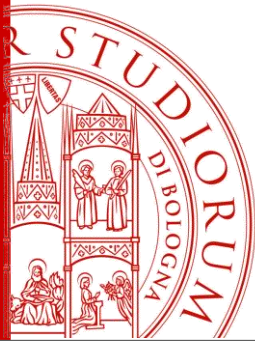
strategy SimpleReduceResponseTime [ styleApplies && cViolation ] {
  define boolean hiLatency = exists k:T.HttpConnT in M. connectors | k.latency > M.MAX_LATENCY;
  define boolean hiLoad = exists s :T. ServerT in M. components | s . load > M.MAX_UTIL;

  t1: (#[Pr{t1}] hiLatency) -> switchToTextualMode() @[1000/*ms*/] {
    t1a: (success) -> done ;
  }

  t2: (#[Pr{t2}] hiLoad) -> enlistServer(1) @[2000/*ms*/] {
    t2a: (!hiLoad) -> done ;
    t2b: (!success) -> do [1] t1 ;
  }

  t3: ( default ) -> fail ;
}
```

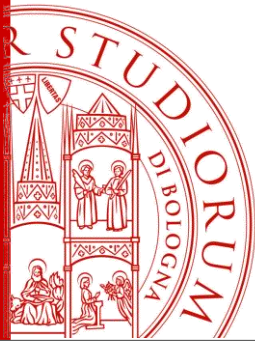
Cheng, Garlan, and Schmerl, “Stitch: A Language for Architectural-based Self-Adaptation” , Journal of Systems and Software, Special Issue on State of the Art in Self-Adaptive Systems, 2012



3. Stitch

Strategy Selection – chooses the strategy with the highest utility. This is achieved through the definition of:

- Quality dimensions
 - Identifier, label, description, mapping to architectural property, utility function definition
- Utility preferences
 - Used to define business priorities over quality dimensions
- Impact vectors
 - Costs and benefits that a tactic has on each quality dimension, defined as deltas
- Branch probabilities
 - Captures the fact that a tactic could not achieve its effect, or that a tactic's activation condition might not be met

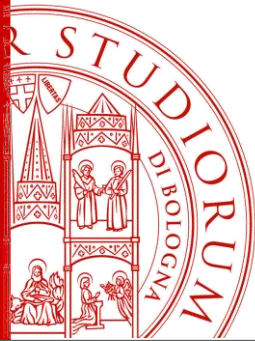


Reasoning on adaptive systems

Our department has an ongoing collaboration with CeSIA, the center responsible for the whole IT infrastructure of the University of Bologna

We cooperate with an ongoing effort to **model** the complexity of the CeSIA systems and **reason** about the models at run-time

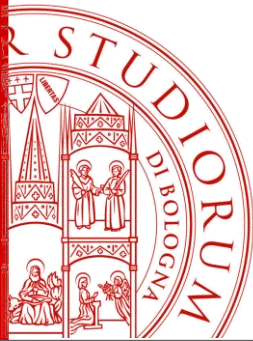
- we are developing an adaptation engine based on semantic models for the management of this vast infrastructure



Case Study

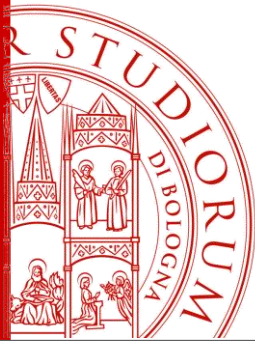
Enterprise: UniBo IT infrastructure @ CeSIA

- 510 Km optical fiber, 160 centers
- ~500 servers (90% virtual) - 264 Cores, 1408 GB Ram, 450 TB data)
- 480 websites (12.400.000 visitors/year, 137.000.000 page hits)
- 100.000 students
- 6.200 employees (teaching/administrative/technical staff)
- HelpDesk: 50.000 tickets/year




Research Challenges

- **CH1 - heterogeneity management**
 - each component of legacy systems uses different programming languages, technologies, architectural styles, etc.
 - *which general framework for managing this variability and supporting dynamic adjustments?*
- **CH2 - need for runtime queryable models**
 - model-based adaptation approaches should guarantee easy and uniform mechanisms to interrogate the information about the observed systems
 - *how can we provide up-to-date queryable models, ready to be processed by both humans and adaptation engines?*

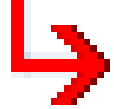


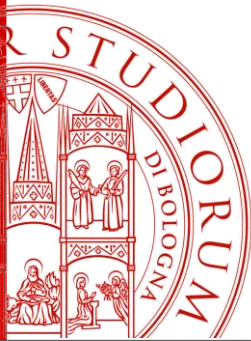
Approach

- **CH1 - heterogeneity management**

 **ONTOLOGIES** (for providing a shared and unified representation of complex and heterogeneous domains of interest)
+
REASONERS (for driving complex systems' adaptation)

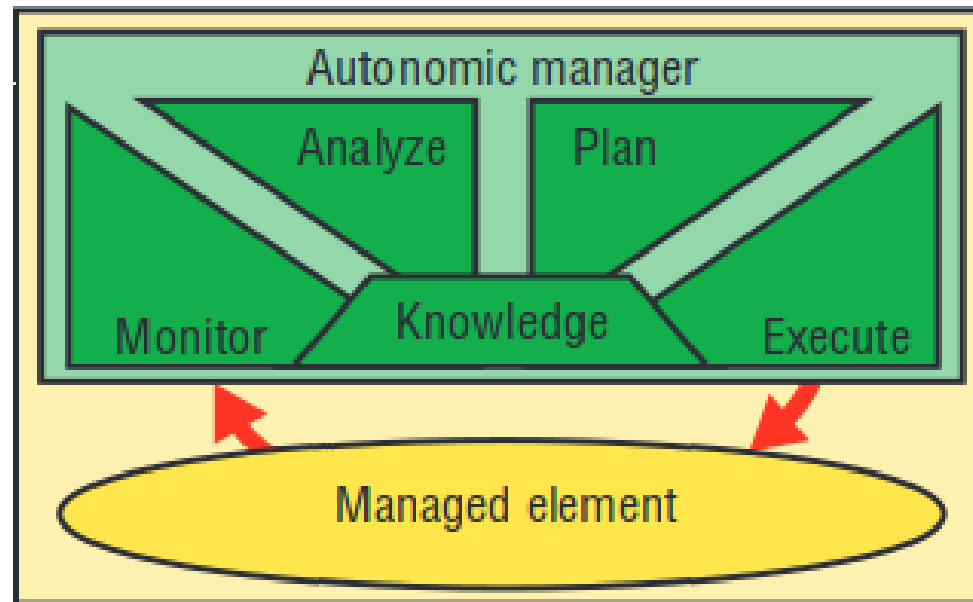
- **CH2 - need for runtime queryable models**

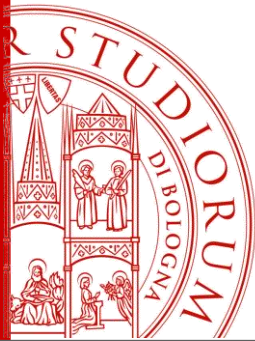
 **TRIPLESTORES** (for providing standard access mechanisms to semantic models at runtime)



System Architecture: the MAPE-K Paradigm

- For developing the adaptation logic, we used an external approach.
- The engine has been implemented as a feedback loop following the MAPE-K paradigm .

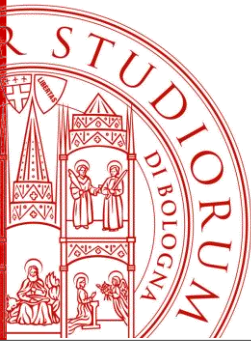




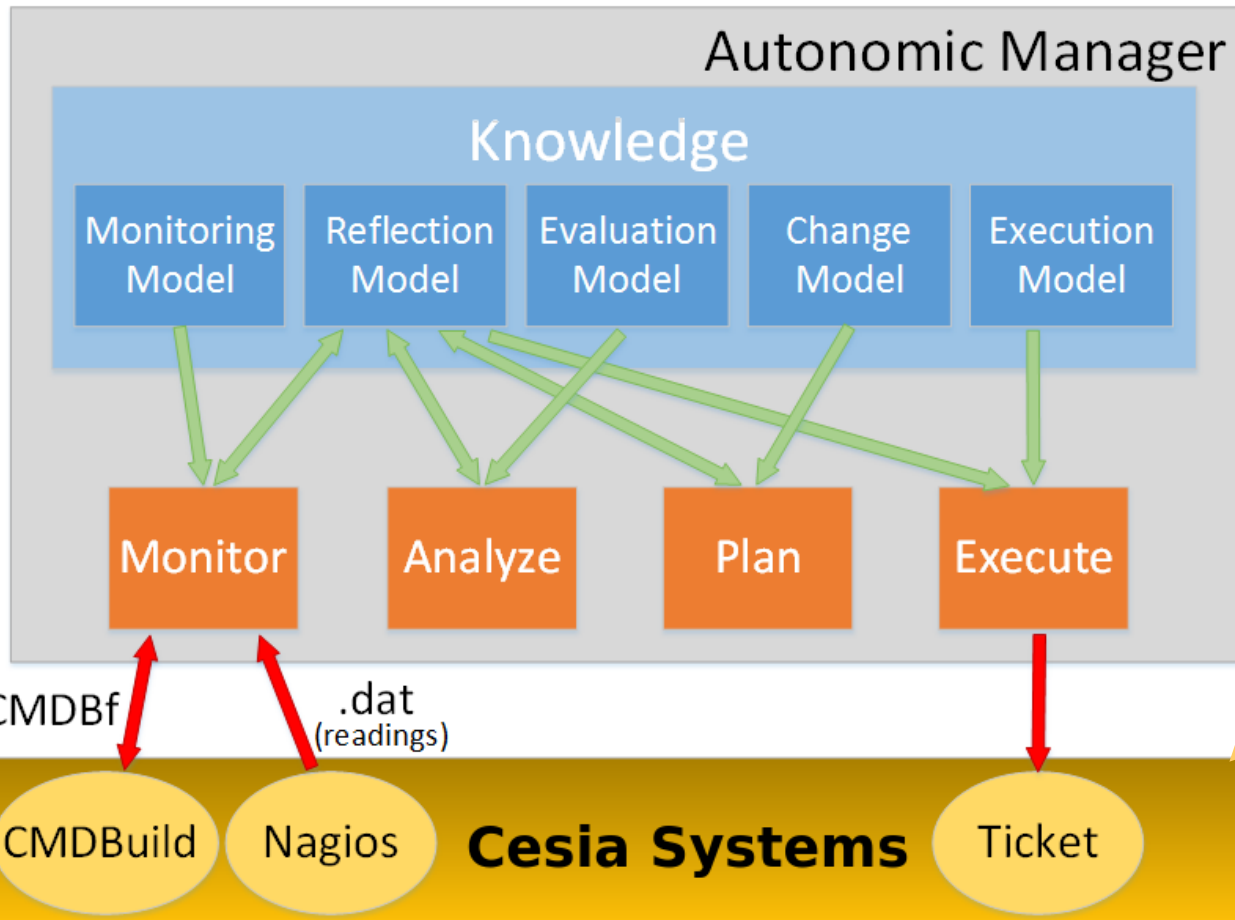
System Architecture

The Shared Knowledge

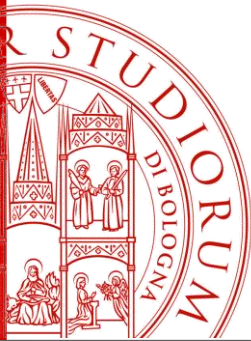
- The shared knowledge is refined to the following set of semantic models:
 - **Reflection Models:** which reflect the adaptable software and its environment, mapping system-level observations to an higher level of abstraction. Reflection Models accommodate both static and dynamic information;
 - **Monitoring Models:** for mapping the system-level observations to the abstraction level of the Reflection Models;
 - **Requirement Models:** define the system expected behavior;
 - **Evaluation Models:** specify the reasoning, e.g., by defining constraints that are checked on the Reflection Models;
 - **Change Models:** devise the most appropriate reconfiguration plans if adaptation needs have been identified;
 - **Execution Models:** define mappings between reconfigurations and system-level adaptations.



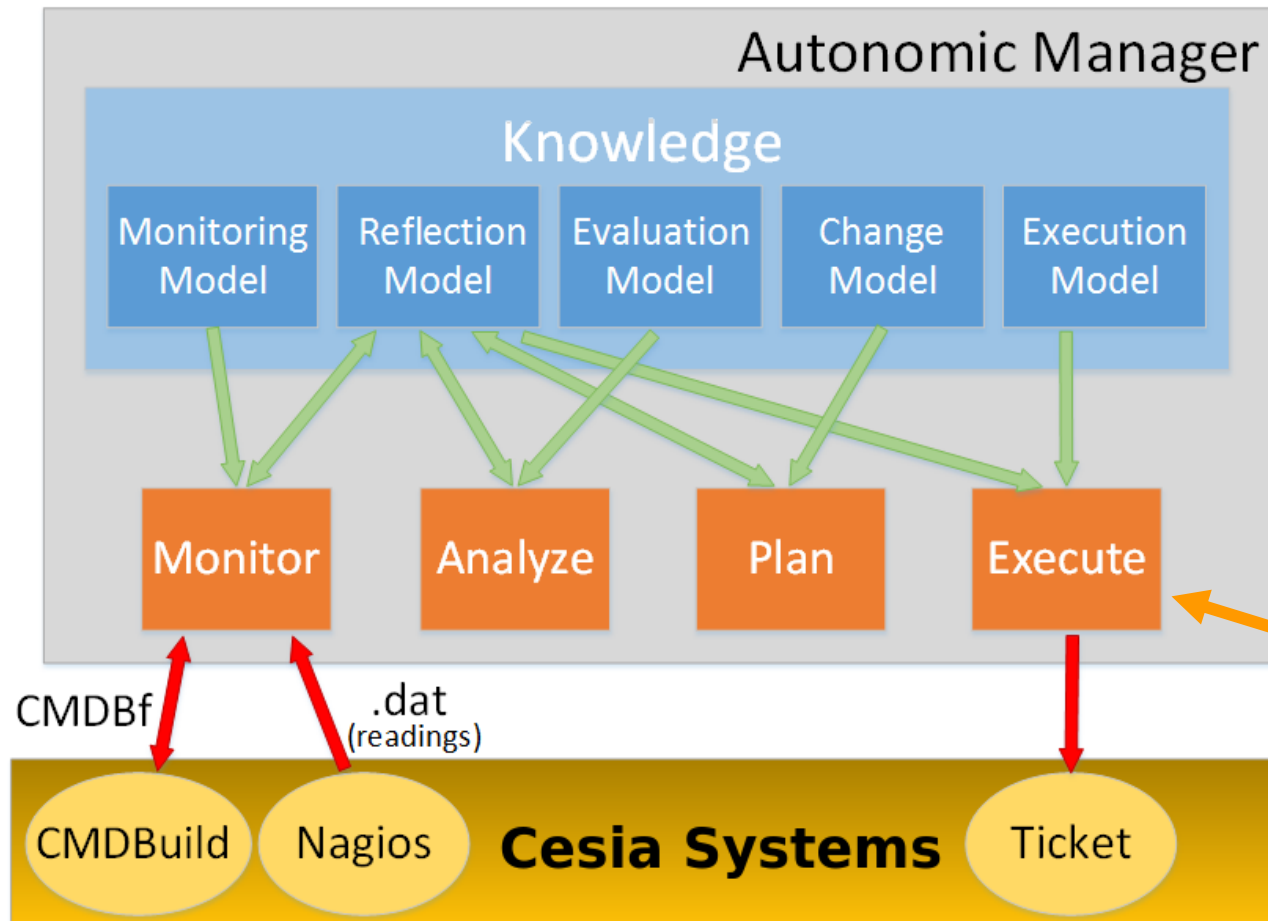
System Architecture: the Observed system



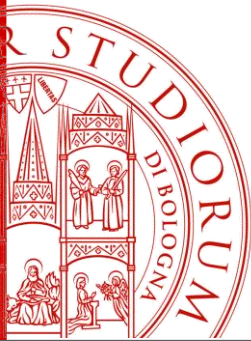
- **CMDBuild:** structural (*static*) information (i.e. servers, applications, services & their relationships)
- **Nagios:** the CeSIA monitoring infrastructure (*dynamic* behaviors)
- **Ticketing system:** collects detailed descriptions of the reconfiguration policies that need to be carried out



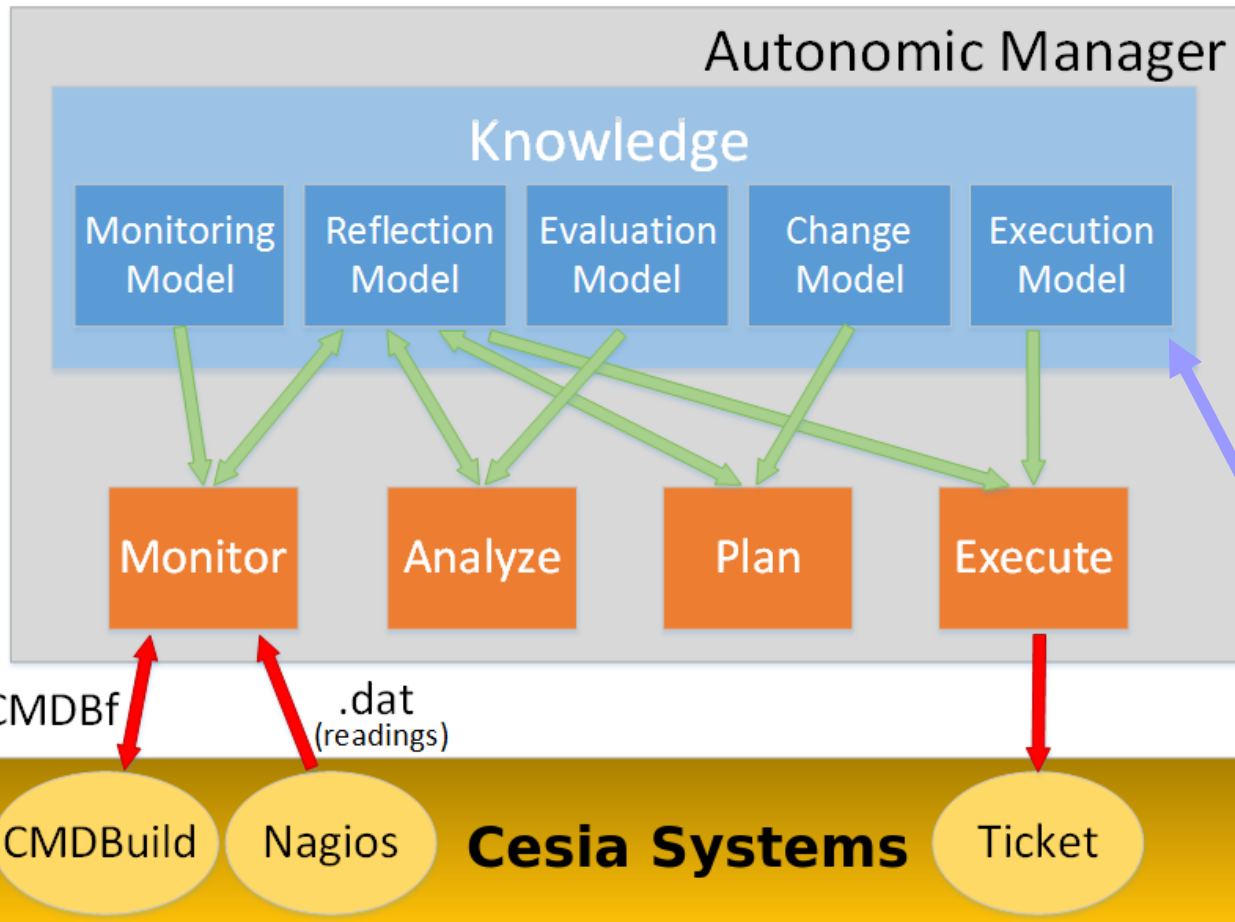
System Architecture: the 4 MAPE Phases



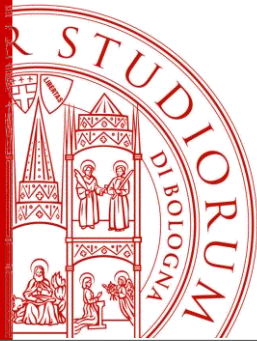
- Concurrent Java components implement the 4 MAPE-K phases
- They coordinate by passing a control token through shared queues
- An external thread supervises their activities, intervening when needed (e.g. anomalous behaviors, excessive delays, etc.)



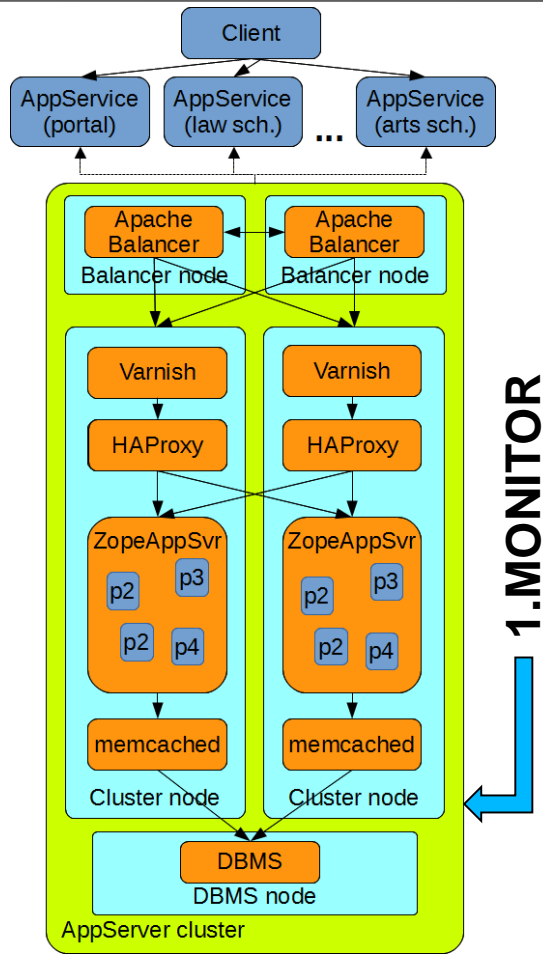
System Architecture: the Shared Knowledge



- A **set of semantic models** refines the shared knowledge, with all the information required to implement any adaptation activities
- These models are implemented as **OWL ontologies**, stored in an external triplestore
- The 4 phases communicate by means of the the triplestore (through standard SPARQL queries)



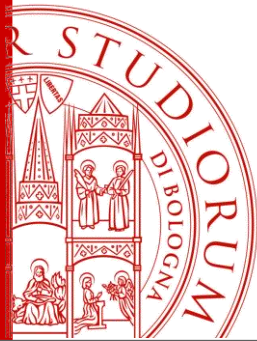
Monitor and Analyze Phases



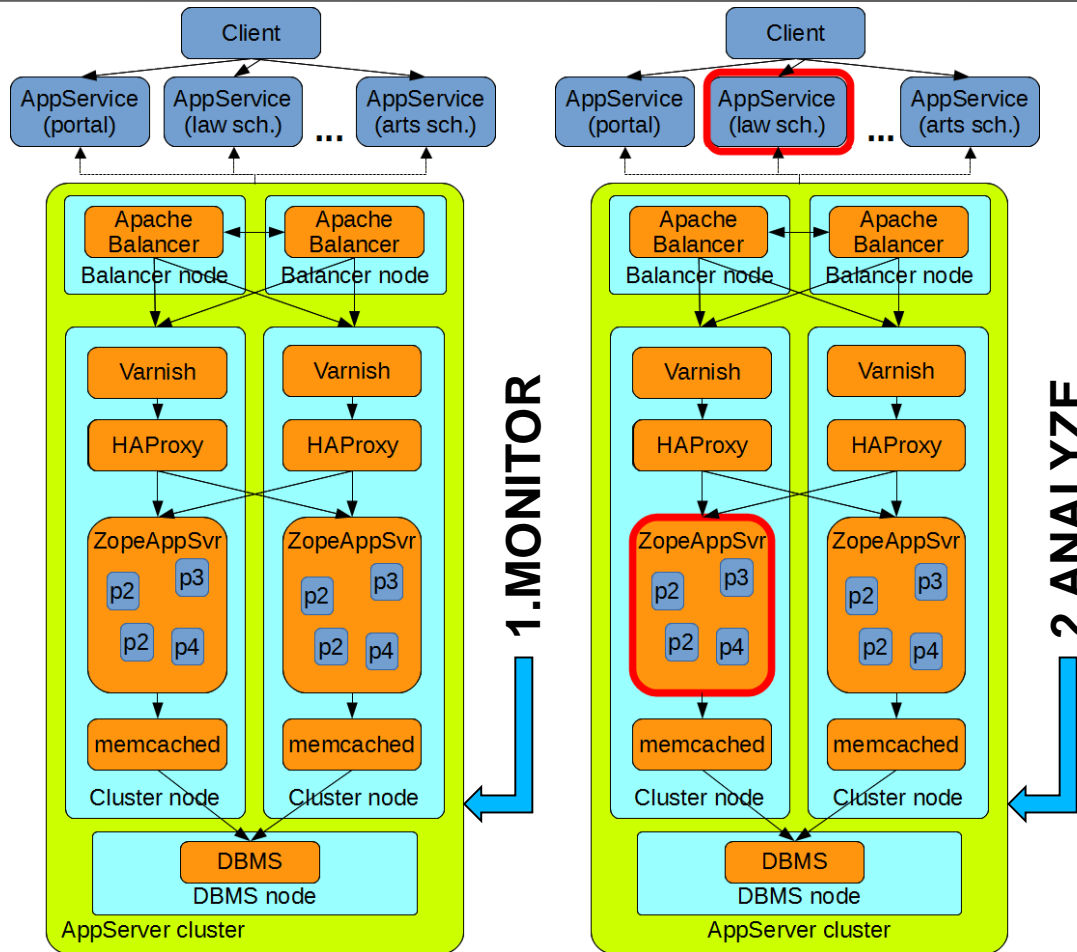
1. Monitor

Goal: provide an updated view of the system state:

- Static information
- Dynamic information



Monitor and Analyze Phases



1. Monitor

Goal: provide an updated view of the system state:

- Static information
- Dynamic information

2. Analyze

Goal: recognize and characterize all the critical situations that should be resolved by a system reconfiguration. E.g.:

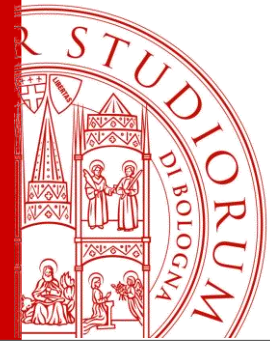
Slow application:

response time > 10 sec.

Overloaded application server:

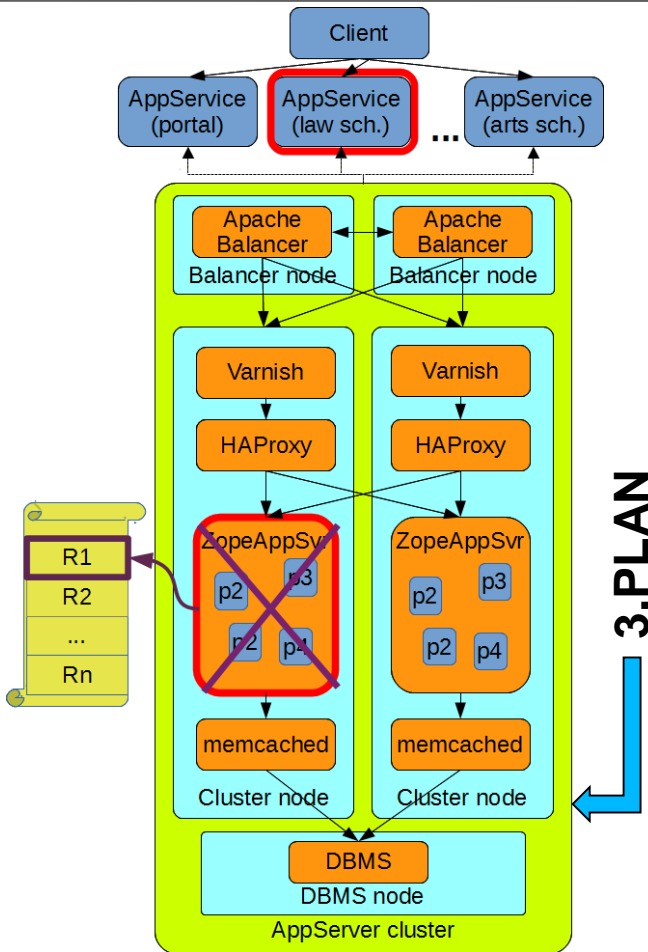
#queued requests > 20

Plan and Execute Phases

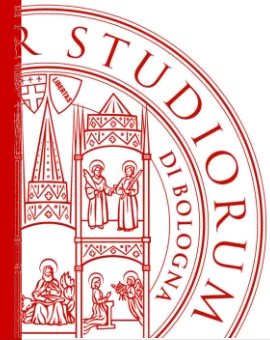


3. Plan

Goal: devise, if needed, a reconfiguration plan for reacting to the actual issues, by select the most suitable reconfiguration policies for the selected components



Plan and Execute Phases

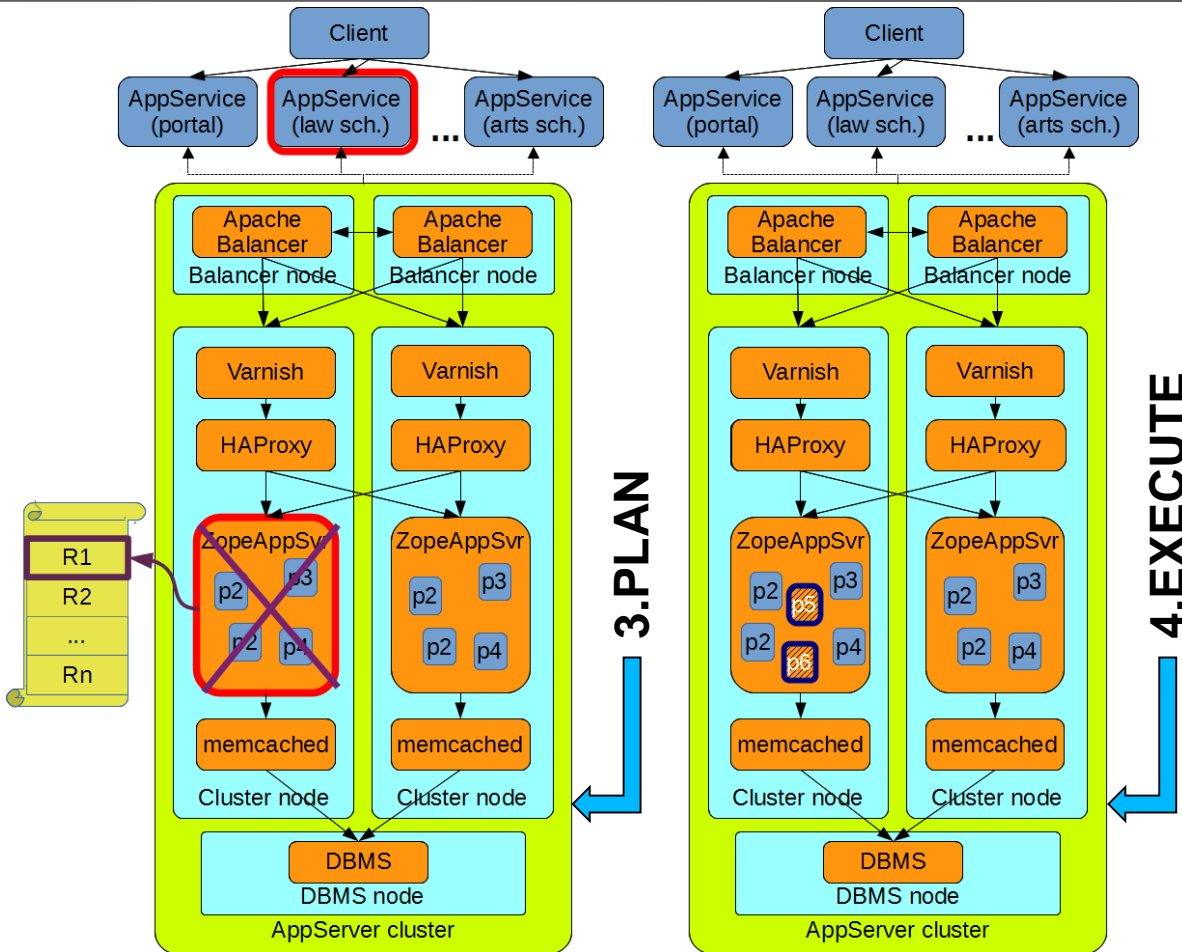


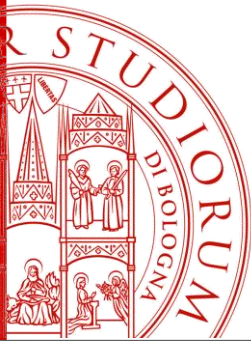
3. Plan

Goal: devise, if needed, a reconfiguration plan for reacting to the actual issues, by select the most suitable reconfiguration policies for the selected components

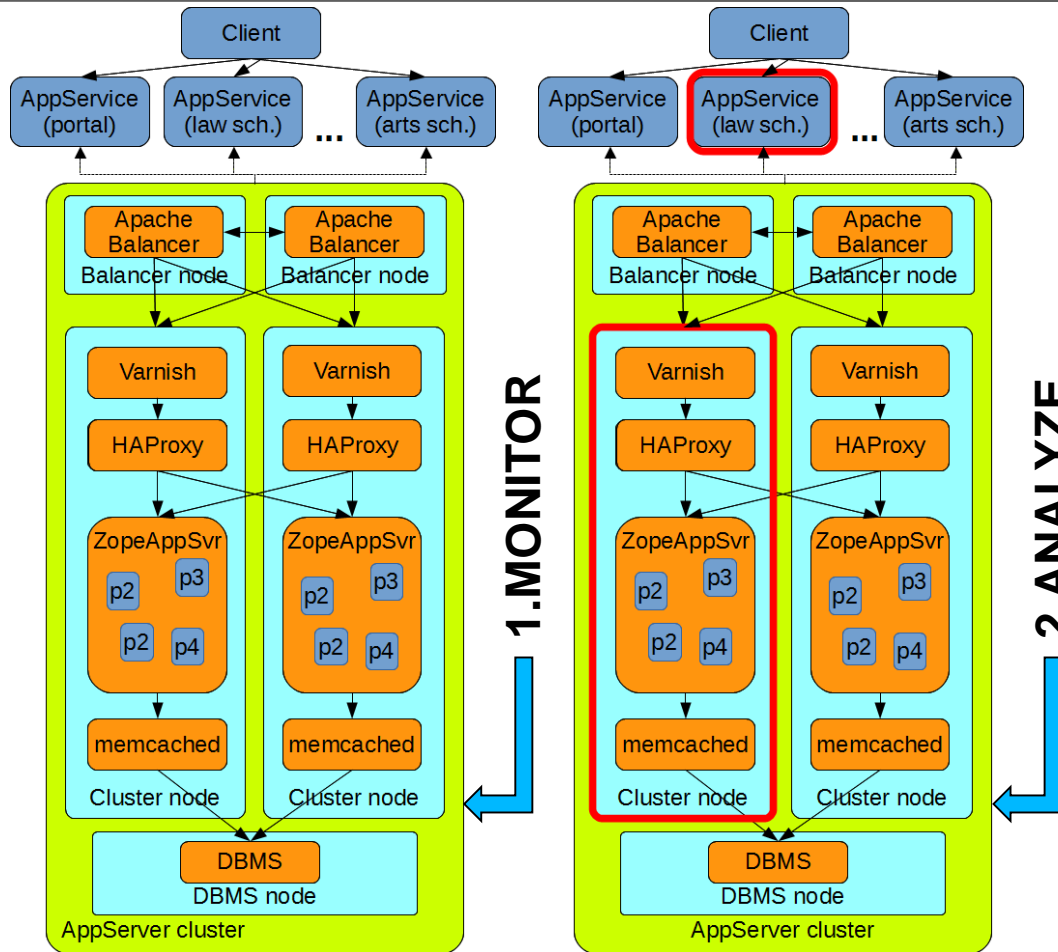
4. Execute

Goal: convert the model-level adaption into the system level, and re-synchronize the Reflection Model (in the KB) with the system state





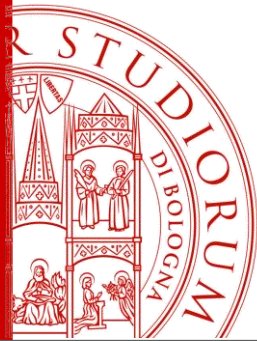
Another Example



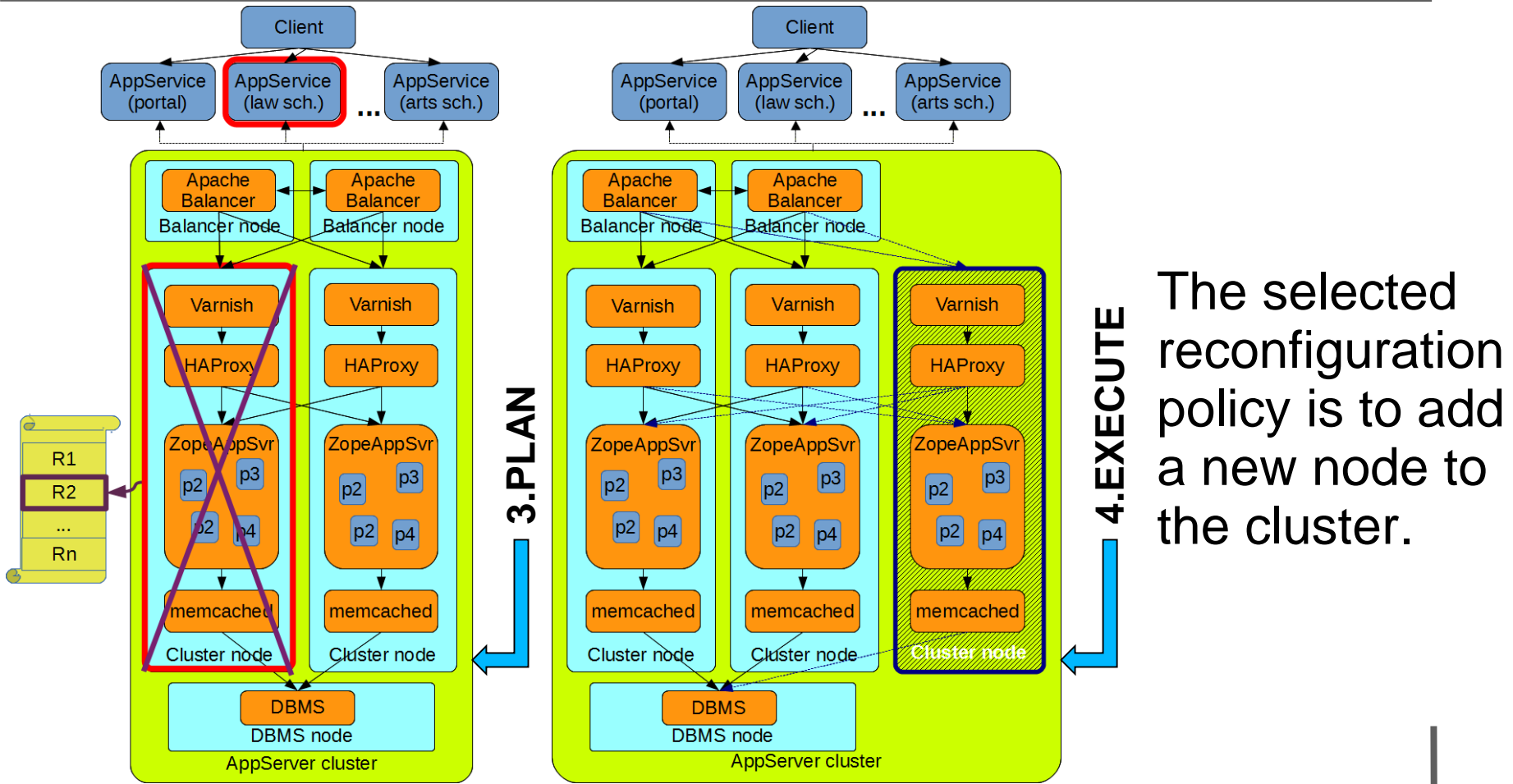
In this case the delays in the portal of the School of Law are caused by an overloaded cluster node:

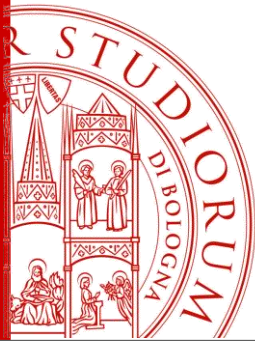
Overloaded cluster node:

CPU Load > 95%



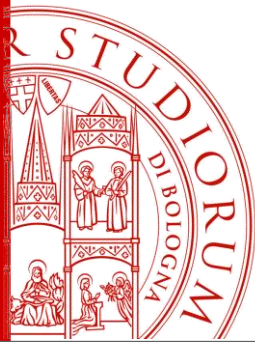
Another Example





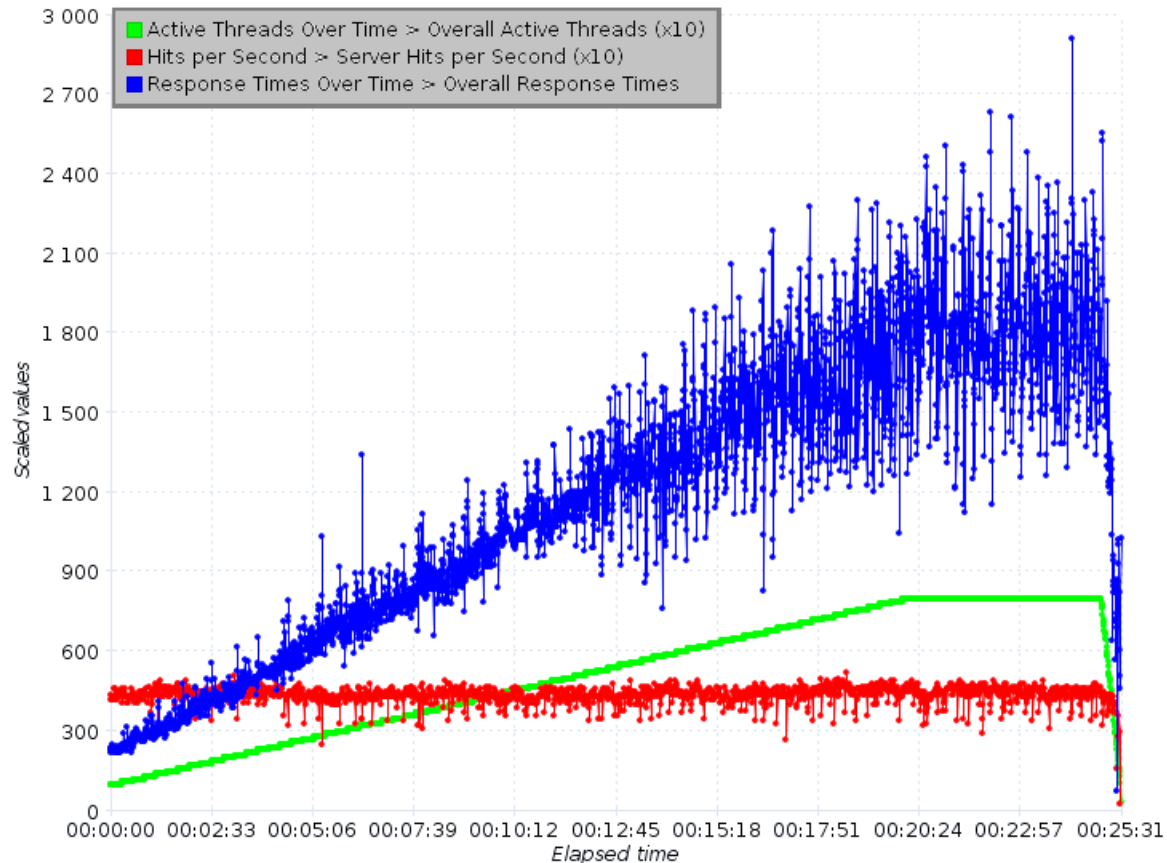
Queryable Models

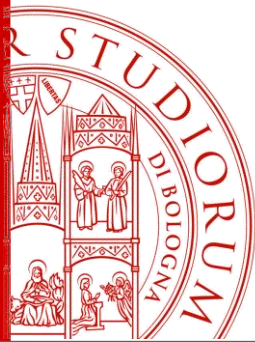
- Our approach traces information about
 - system states
 - behaviors
 - adaptation strategies
- that can also be used for other - maybe unexpected at design time - purposes.
- SPARQL provides a single interface to:
 - query the CeSIA model
 - integrate with external datasources
 - e.g. can be joined with UniBo and CeSIA calendars to improve the adaptation strategies



Experimental Results

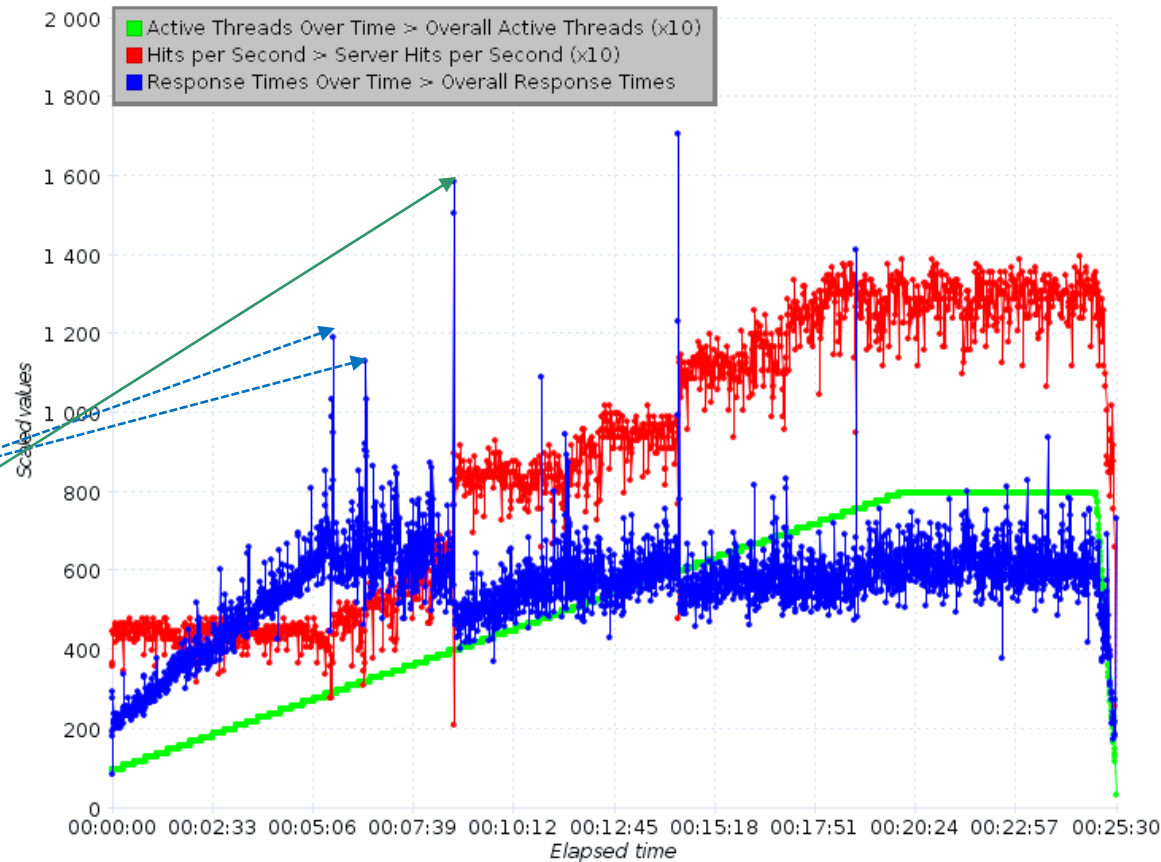
- Artificial load generator: raising number of connections;
- With the autonomic manager **turned off**, the CeSIA system doesn't meet the SLAs
 - e.g. responseTime < 1.2 sec

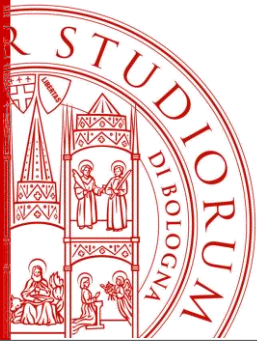




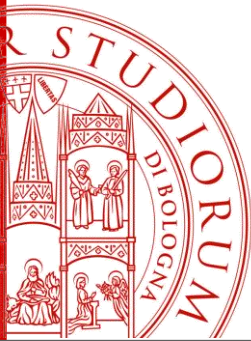
Experimental Results

- Autonomic manager **turned on**
 - responseTime threshold set to 600 ms
- Two reconfiguration strategies:
 - adding new **processes**
 - adding new **nodes**
- ResponseTime is kept around 600 ms (even when the load increases)



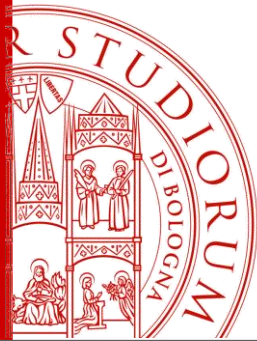


Conclusions



Software Architecture

- Software Architecture provides a level of abstraction that allows us to
 - Scale our understanding of complex systems, and
 - Manage adaptation (and the self-* properties of our systems)
- Programming in the large vs. programming in the small
- A plethora of styles for many different system requirements
 - Yet, it is still possible to generalize and understand what helps make a system adaptable, or self-adaptable



Research in Adaptive Software Architecture

- Software Architecture as a Design-time tool for Systems that Need to be Adaptive
- Software Architecture for Self-Adaptive Systems
- Architecture as a living entity supporting adaptation frameworks

