

# MetaSelf – An Architecture and a Development Method for Dependable Self-\* Systems

Giovanna Di Marzò  
Serugendo  
Dept of Computer Science  
and Information Systems  
Birkbeck College, London, UK  
dimarzo@dcs.bbk.ac.uk

John Fitzgerald  
School of Computing Science  
Newcastle University  
Newcastle Upon Tyne, UK  
John.Fitzgerald@ncl.ac.uk

Alexander Romanovsky  
School of Computing Science  
Newcastle University  
Newcastle Upon Tyne, UK  
Alexander.Romanovsky@ncl.ac.uk

## ABSTRACT

This paper proposes a software architecture and a development process for engineering dependable and controllable self-organising (SO) systems. Our approach addresses dependability by exploiting metadata to support decision making and adaptation based on the dynamic enforcement of explicitly defined policies. Control is obtained by actively modifying metadata, policies or components. We show how this applies to two different systems: (1) a dynamically resilient Web service system; and (2) an industrial assembly system with self-adaptive and SO capabilities.

## Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architecture

## General Terms

Design-time and run-time adaptation, control loop

## Keywords

Self-organisation, self-adaptation, metadata, policies

## 1. INTRODUCTION

Self-organisation is the ability of a system to organise itself without explicitly being instructed to do so. Self-organisation principles, as observed in natural systems (e.g. ant colonies, flocks of birds) are increasingly applied to artificial systems. They provide robustness to systems that must cope with volatile environments, while retaining simplicity in individual system components. Most of the results achieved to date focus on ad hoc implementations in domains such as optimisation problems, robotics, services, or MANETs. Though displaying great potential, artificial self-organising (SO) systems are not yet trustworthy (in the sense of being dependable *and* displaying evidence to support claims of dependability) or controllable. Their actual

(emergent) behaviour is often difficult to predict and control (e.g. stop, reset or guide); they show latency when adapting to changes, sensitivity to initial conditions and instability. Approaches to dependable SO systems range from multi-layer reference architectures for self-adaptive systems [10], to analysis guidelines and specific collaborative agents solutions for SO systems [14], and design of SO systems through information flows [5]. Despite the large body of work in this area, there are as yet no systematic techniques for engineering (designing and implementing) trustworthy and controllable SO systems. To overcome these limitations, we have been working on a software architecture and development method combining design-time and run-time features which permit the definition and analysis at design-time of mechanisms that both ensure and constrain the run-time behaviour of a SO system, thereby providing some assurance of its self-\* capabilities.

Our approach, called *MetaSelf*, involves loosely coupled autonomous components which are the building blocks of the SO system, dynamically updated and retrieved metadata, rules for self-organisation, and dependability policies instantiating generic SO and resilience mechanisms. At design time, SO mechanisms are identified and corresponding generic dependability policies specified. At run-time, both the components and the run-time infrastructure exploit metadata to support decision-making and adaptation based on the dynamic enforcement of instantiated policies.

In this paper, we present the *MetaSelf software architecture and development method*. Section 2 briefly reviews SO systems and dependability. Sections 3 and 4 describe the *MetaSelf* architecture and development process respectively. Section 5 discusses how *MetaSelf* supports dependability and control in the SO systems. Two examples are presented in Sections 6 and 7. Related works are discussed in Section 8.

## 2. SELF-ORGANISING SYSTEMS

Self-organising applications are generally made of multiple autonomous components that locally interact (directly or indirectly) with each other to produce a result. Typical examples of natural self-organisation are swarms (ant, flocks of birds, wasps, etc.). Artificial (engineered) systems include unmanned vehicles, swarms of robots, P2P systems, immune computer or trust-based access control. Leveraging from [6], we consider that the *design elements* of a SO system are: the *environment* in which the system evolves; the *autonomous components* that constitute the system itself (software agent, robots, peer nodes, services); *self-organising mech-*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'10 March 22–26, 2010, Sierre, Switzerland  
Copyright 2010 ACM 978-1-60558-638-0/10/03 ...\$10.00.

*anisms* governing the interactions among the components (generally defined by rules for self-organisation that each component applies); and *artifacts* – passive entities maintained by the environment, created, modified and/or sensed by the agents (e.g. digital pheromone spread in the environment or information exchanged among agents). Faults in a SO system arise from faults or changes in one of the four design elements listed above. For instance, changes in the environment, malicious components, design errors in the SO mechanism, etc. Dependability in the SO context means that a system satisfies its properties (invariants, convergence, stability, availability, etc.) in the face of such faults. Dependability in SO systems is obtained either through the SO mechanism itself, by devising an enhanced SO mechanism (better design), or by applying dependability policies.

### 3. METASELF ARCHITECTURE

MetaSelf follows a service-oriented architecture (Figure 1). It exploits metadata to support decision-making and adaptation based on the dynamic enforcement of explicitly expressed policies. Metadata and policies are themselves managed by appropriate services. The main elements of the MetaSelf architecture are the following:

**Self-Describing Components/Services/Agents.** As we have seen above, autonomous components (e.g. software agents, robots, peers, services) are the active entities at the heart of a SO system. Interoperability is fundamental when different service providers are involved in the same system. Decoupling components (software programs) from descriptions of their capability, QoS, requirements and constraints is thus a solution for solving interoperability and deriving run-time solutions in case of unexpected condition.

**Acquired, Updated & Monitored Metadata.** Sensing and acting is a fundamental activity in SO systems. This requires appropriate metadata that may be published; that is permanently acquired, updated and monitored at run-time by *both* the system's components (for sensing, acting) and the supporting infrastructure (for monitoring activities). Different types of metadata are available: component descriptions (possibly including interface information), environment related metadata (possibly supporting coordination and self-organisation), metadata related to either individual components (e.g. availability level, efficiency) or to groups of components (supporting dependability policies).

**Self-\* Mechanisms.** Self-\* mechanisms describe how adaptation is triggered on the basis of metadata. It consists of both the SO mechanism driving the interactions among the autonomous components and any self-adaption required for ensuring dependability, e.g. replacement of a component with a functionally equivalent one if the performance of the first deteriorates; an adaptive fault tolerance structure in which replicas are dynamically configured; what coordination action to take next on the basis of locally sensed information (stigmergy). These patterns are implemented through *rules for self-organisation* or through *dependability policies*. Rules for self-organisation are followed by all components and applied regularly. Dependability policies apply punctually to recover from or anticipate an error by taking appropriate steps. Dependability policies are available at run-time to both the run-time infrastructure and the components themselves. Policies come in different categories, and may apply at system level (guiding policies), component level (monitoring policies), or refer to environmental

constraints (bounding policies).

**Enforcement of Policies.** The run-time infrastructure is equipped with services responsible for enforcing policies on the basis of current metadata values and changes in metadata values. These services may act directly on components by performing replacements and reconfigurations. Each service provides tasks related to processing of metadata, such as comparison/matching, determination of equivalence and metadata composition. They also encompass automated reasoning over policies and metadata.

**Coordination/Adaptation.** This service manages the list of components, seamlessly activates or connects the ones that will be used according to the specified rules for self-organisation (coordination) or dependability policies (adaptation). It encompasses automated reasoning on adaptation policies.

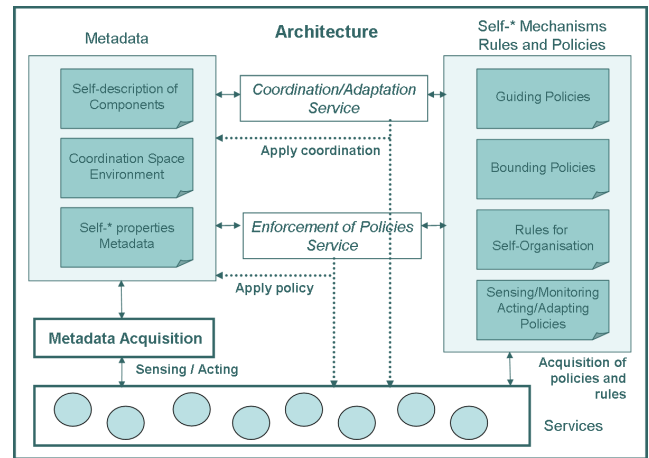


Figure 1: Run-time Generic Infrastructure

It is worth noting that the architecture is not necessarily centralised; the services providing access to the description of components, or monitoring and acquisition of metadata can reside at different locations and work autonomously. In addition, metadata and policies have either a local or global scope, and can be locally attached to a component. The actual implementation depends on the application.

### 4. METASELF DEVELOPMENT METHOD

The MetaSelf development process consists of 4 phases (Figure 2). The **Requirement and Analysis phase** identifies the functionality of the system along with self-\* requirements specifying where and when self-organisation is needed or desired. The required quality of service is determined. The **Design phase** consists of two sub-phases: **D1:** the designer chooses architectural patterns (e.g. autonomous manager or observer/controller architecture) and self-\* mechanisms (governing the interactions and behaviour of autonomous components (e.g. trust, gossip, or stigmergy). Generic rules for self-organisation and dependability policies are defined. In the second part; **D2:** the individual autonomous components (services, agents, etc.) are designed. The necessary metadata and policies are selected and described. The self-\* mechanisms are simulated and possibly adapted or improved. The **Implementation phase** produces the

run-time infrastructure (Figure 1) including agents, meta-data and executable policies. In the **Verification phase**, the designer makes sure that agents, the environment, artefacts and mechanisms work as desired. Potential faults and their consequences are identified, similar to the way *failure modes and effects analysis (FMEA)* [11] works. Corrective measures (redesign or dependability policies) to tolerate or remove the identified faults are taken accordingly.

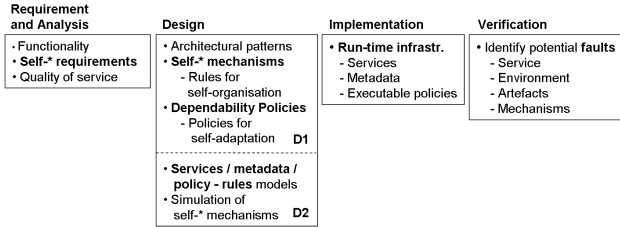


Figure 2: MetaSelf Development Process

## 5. DEPENDABILITY AND CONTROL

**Dependability.** Means for achieving dependability encompass prevention of faults, fault tolerance, fault removal and fault forecasting [2]. MetaSelf supports dependability in the following ways. Prevention of faults is provided by simulations (phase D2 of the design) that allow designers to establish rules for self-organisation and dependability policies. Fault tolerance is provided by the SO mechanism and by setting any additional and punctual dependability policy that complements it. Fault removal is achieved by revised design of rules for self-organisation and dependability policies. Finally, fault forecasting is obtained through dependability policies and monitored metadata (an action is taken before an error occurs, the decision is based on metadata values).

**Control and feedback loop.** Figure 3 shows the different levels of control supported by the architecture. *Internal or low-level control* is a result of the activity of the components. Components sense and retrieve metadata and policies. Their behaviour causes metadata changes which in turn causes components to individually adapt to the new situation. *External or high-level control* is provided in three ways. The run-time infrastructure itself is active and through reasoning services enforces active external or human/admin control by direct reconfiguration of components; modification of the metadata used by the components to sense their environment; and modification of the policies used by the components for driving (changing) their behaviour on-the-fly. Loose coupling is crucial. Dynamically changing a policy or metadata occurs without modifying/stopping the components, their new value immediately affects the components behaviour. This is useful when devices change context (e.g. a PDA moving around), or when global policies change (e.g. rights are denied to some user).

## 6. DEPENDABLE WEB SERVICES

In this section we briefly show how the proposed architecture has been applied in developing a Web Service architecture called WS-Mediator intended as a novel approach to im-

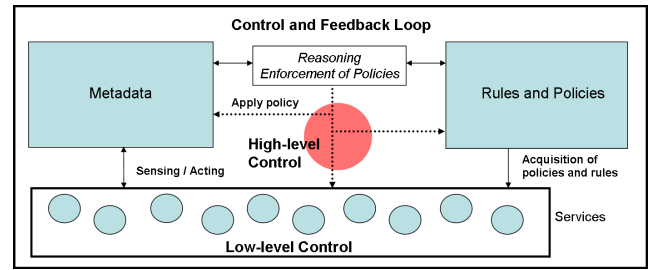


Figure 3: Control and Feedback Loop

proving dependability of Web service composition [4]. WS-Mediator relies on an off-the-shelf mediator solution implementing run-time dependability monitoring and assessment, resilience-explicit computing and fault tolerance mechanisms used together to achieve dependable dynamic Web service integration. A Java WS-Mediator framework has been developed and applied in experiments conducted in the bioinformatics and virtual organisation domains.

The WS-Mediator architecture is developed for the Web service domain in which components, i.e. Web services, meet our requirements for decoupling from their descriptions and from the underlying coordination infrastructure. Self-awareness is architected as an overlay network of dedicated components monitoring the application services and collecting dynamic information about their behaviour as the metadata ( $m, r, f$ ) described below. The WS-Mediator system consists of a set of interconnected functionally-identical Sub-Mediators distributed over the Internet to monitor the dependability of Web services and to provide accurate metadata, presenting Web service dependability characteristics from the client's perspective. Each service is periodically invoked and, if a valid result is returned, an availability score ( $m$ ) increases. The round-trip response time of the invocation is recorded for calculating the average response time ( $r$ ) of the service. If the service returns an invalid response, the value of  $m$  decreases, and the error message is logged in the database for statistic ( $f$ ) recording types of failure. If the service fails to respond, or if an exception arises during the invocation,  $m$  also decreases, and the type of the exception is also logged for  $f$ .

The collected metadata are dynamically analysed and used to choose and enforce one of the fault tolerance policies. These policies are statically defined by the system architect to represent the most efficient strategies of tolerating a wide range of erroneous conditions in the application services and of the communication media. The policies utilise the types of redundancy available in such global setting, including path diversity and application service diversity, and allow synchronous and asynchronous invocation of the application services. The most typical fault tolerance modes use backward recovery (retry and try of a similar service [15]), masking diversity (invoking several similar services concurrently [1]) and path diversity (routing the service invocation through different routes). A basic reasoning engine is implemented as part of each WS-SubMediator to ensure the best choice for the current situation in the network and the application services as observed from the location of this WS-SubMediator.

The WS-Mediator architecture has been evaluated in two

major sets of experiments. First we verified the validity of dependability monitoring using two Web services provided to us by the GOLD project<sup>1</sup>. Second, in order to demonstrate the applicability and effectiveness of the WS-Mediator approach, we experimented with three Web services implementing an algorithm which is commonly used in *in silico* experiments in bioinformatics to search for gene and protein sequences that are similar to a given input query sequence (so-called BLAST services<sup>234</sup>).

*Proof-of-concept:* this example shows that a dynamically resilient system can be implemented using the notions of (monitored) metadata and policies. It is a self-adaptive system, where globally available and up-to-date information about Web services behaviour is used at run-time for overcoming different types of errors (e.g. non responsive or overloaded Web service).

## 7. INDUSTRIAL ASSEMBLY SYSTEMS

This section shows how the MetaSelf approach has been applied to enhance assembly systems with self-organisation and self-adaptation. It also highlights the steps necessary to produce the design of a SO and self-adapting system. Additional descriptions of this example can be found in [8].

An *assembly system* is an industrial installation that receives parts and assembles them in a coherent way to form a final product. It consists of a set of equipment items (modules) such as conveyors, pallets, simple robotic axes for translation and rotation as well as more sophisticated industrial robots, grippers, sensors of various types, etc.

Our goal is to design *self-organising assembly systems*. Given an order for a specified product, the system's modules spontaneously select each other (preferred partners) and their positions in the assembly system layout. They also program themselves (instructions for robots' movements). The result of this SO process is a new or reconfigured assembly system that will assemble the product ordered. The appropriate assembly system *emerges* from the self-organisation process among the modules. This automated process does not stop at layout formation. During production time, whenever a failure or weakness occurs in one or more of the current elements of the system, either the current modules adapt their behaviour (change speed, force, task distribution, etc.) in order to cope with the current failure, possibly degrading performance but maintaining functionality; or may decide to trigger a reconfiguration to effect a repair.

**Components.** The components (and services) of the system are provided by several agents. The *order agent* carries the generic assembly plan for building the product. This is a series of instructions specific to the products being assembled, but independent of both the specific modules that will carry the assembly and their positioning layout. The *product agents* carry the layout-specific assembly instructions (micro-instructions) which the visited resource modules will be asked to execute. This is a translation of the generic assembly plan specific to the modules that participate in the layout and to their position. The *manufacturing resource*

*agents* include the conveyor units, robots, grippers, feeders, etc. and the *part agents* are the the parts to be assembled.

**Self-\* Requirements.** *Layout Formation.* Any new assembly plan triggers a self-organisation process leading to a new configuration (layout), i.e. self-selection of modules and their partners, and establishment of the layout-specific assembly instructions. *Task coordination at production time.* This encompasses task sequencing (modules coordinate their work according to the current status of the product being built); and collision avoidance (modules with overlapping workspace must maintain a minimum separation while moving). *Self-Adaptation.* Self-healing: During production, if a module failure is detected (blocked gripper, lost part), by the module itself or one of its partners, either the modules can solve the problem by themselves (software restart, open and close a gripper, etc.) or alert the user. Self-optimization: During production, the speed of processing the product parts is adjusted to the queuing level.

**Self-\* Mechanisms.** *Self-organisation mechanism.* The mechanism that allows modules and product parts to reorganise when a new assembly plan arrives, or when a change in a module is requested, is inspired by the chemical abstract machine (CHAM) [3]. In this model components are parts of a chemical solution, rules for self-organisation are chemical reactions for binding physically compatible modules and for matching modules skills with expected services. *Coordination mechanism.* During production, the coordination of tasks for each individual product item is done through indirect communication by storing the current advancement of the assembly in a shared place (a RFID attached to the product item). *Self-adaptation mechanism / Dependability.* Self-adaptation, e.g. resilience to failures, to the effects of fatigue and collision avoidance, is performed by modules monitoring their own or their neighbour's behaviour.

**Policies and Metadata.** CHAM rules for layout formation include: matching of modules skills to requested assembly task (rotation or vertical movement); physical compatibility among modules (shapes and sizes) and rewriting of generic assembly tasks (screw) into specific ones (which module performs the screwing operation and exact movement positions). Some examples of self-healing policies include: if the target position after a movement has not been reached correctly, take corrective measures (advance more or less, ask for maintenance); in case of malfunctioning, request a replacement from a coalition partner; if no replacement found, ask for a re-configuration of the layout. Metadata include: optimal speed of operation; own precision (movements on every axis); precision of partners (neighbours); quality of assembled product.

**Implementation.** A specific decentralized instantiation of the architecture in Figure 1 has been developed [8]. The CHAM rules are designed using K-Maude. Implementation of this example is reported in [9]: an appropriate ontology has been developed with Protégé and is loaded at run-time using Jena, agents are developed with Jade, rules and policies are enforced with Jess.

*Proof-of-concept:* this example combines self-adaptation (adaptation to production conditions) and self-organisation (layout formation) features, for which different self-\* requirements and mechanisms have been identified. This examples demonstrates how to apply the MetaSelf approach when designing such a system: identification and selection of self-\* requirements and corresponding self-\* mechanisms.

<sup>1</sup><http://www.neresc.ac.uk/projects/GOLD/projectdescription.html>

<sup>2</sup><http://www.ebi.ac.uk/>

<sup>3</sup><http://hc.ims.u-tokyo.ac.jp/JSBi/journal/GIW00/GIW00P072/index.html>

<sup>4</sup><http://pathport.vbi.vt.edu/main/home.php>

## 8. RELATED WORK

The “Observer-Controller” is a generic paradigm architecture [13] attaching to individual components or groups of components, an “observer” component responsible for monitoring events and states, and a “controller” component responsible for acting in response to the observer’s results. MetaSelf can be viewed as an instantiation of this paradigm. The run-time enforcement of policies acts as a controller, while acquisition and monitoring of metadata act as an observer. White et al. [16] propose a uniform representation and composition of autonomic elements utilising a service-oriented architecture supporting their interactions, preliminary design patterns and policies. The notions of registry, brokers and monitoring are similar to those described in our architecture. Self-Managed Cells (SMC) [7] consist of heterogeneous hardware and software elements, and management services, integrated through a common publish/subscribe event bus. The SMC concept is very close to the approach advocated in this paper. The main differences are that SMC elements have well defined expected interfaces, limiting the possibility for new elements to join the system, especially if they have not been designed by the same team. SMCs do not specifically use metadata, even though elements are monitored, which implies that some metadata are collected about their behaviour. A proposal in [12] is specifically intended for autonomic systems, and shares with our approach the ideas of a service-oriented architecture, of description of services, and use of metadata. The proposed autonomic service-oriented architecture is a three layer architecture (process, service, and application) driven by the process layer, and services are autonomous and monitored by the system.

## 9. CONCLUSION

We have discussed MetaSelf, a software architecture and a development method for engineering dependable SO systems. The key point resides in combining both rules for self-organisation that autonomous components abide to with dependability policies enforced at run-time on the basis of updated metadata. We have briefly described the key elements of the architecture and its application on two examples: dependable Web services and industrial assembly systems. Future works concentrate on actual implementation of the industrial manufacturing system with actual industrial modules and thorough investigation of dependability policies addressing the different types of faults in SO systems.

## Acknowledgements

We are grateful to Yuhui Chen for his work on WS-Mediator. This work has been partially supported by the EPSRC Platform Grant on Trustworthy Ambient Systems (TrAmS).

## 10. REFERENCES

- [1] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Press*, 11:1491–1501, 1985.
- [2] A. Avizienis, J.-C. Laprie, B. Randell, and Landwehr C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.
- [3] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.
- [4] Y. Chen. *WS-Mediator for Improving Dependability of Service Composition*. PhD thesis, Newcastle University (UK), 2008.
- [5] T. De Wolf. *Analysing and engineering self-organising emergent applications*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2007.
- [6] G. Di Marzo Serugendo. Robustness and dependability of self-organising systems - a safety engineering perspective. In *The 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009)*, number 5873 in LNCS, pages 254–268. Springer-Verlag, 2009.
- [7] N. Dulay, E. Lupu, M. Sloman, J. Sventek, N. Badr, and S. Heeps. Self-managed cells for ubiquitous systems. In *Proceedings of the Third International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security, MMM-ACNS 2005, St. Petersburg, Russia, September 25-27, 2005, Proceedings*, volume 3685 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 2005.
- [8] G. Frei, B. Ferreira, G. Di Marzo Serugendo, and J. Barata. An architecture for self-managing evolvable assembly systems. In *IEEE International Conference on Systems, Man, and Cybernetics (SMC 2009)*. IEEE, 2009.
- [9] R. Frei, N. Pereira, J. Belo, J. Barata, and G. Di Marzo Serugendo. Self-Awareness in Evolvable Assembly Systems. Technical Report BBKCS-09-08, School of Computer Science and Information Systems, Birkbeck College, London, UK, 2009.
- [10] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering (FOSE’07)*, pages 259–268. IEEE Computer Society, 2007.
- [11] N. G. Leveson. *Safeware: System Safety and Computers*. Addison Wesley, July 1995.
- [12] L. Liu and H. Schmeck. A roadmap towards autonomic service-oriented architectures. In *International Service Availability Symposium (ISAS 2006)*, pages 193–205, 2006.
- [13] C. Müller-Schloer. Organic computing: on the feasibility of controlled emergence. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2004*, pages 2–5. ACM, 2004.
- [14] G. Picard and M.-P. Gleizes. The ADELFE Methodology - Designing Adaptive Cooperative Multi-Agent Systems. In *Methodologies and Software Engineering for Agent Systems: The Agent-oriented Software Engineering Handbook*, pages 157–176. Kluwer Publishing, 2004.
- [15] B. Randell and J. Xu. The evolution of the recovery block concept. In *Software Fault Tolerance*, pages 1–22. J. Wiley. New York, 1994.
- [16] S. White, J. Hanson, I. Whalley, D. Chess, and J. Kephart. An architectural approach to autonomic computing. In J. Kephart and M. Parashar, editors, *International Conference on Autonomic Computing (ICAC’04)*, pages 2–9. IEEE Computer Society, 2004.