

Dependable Self-organising Software Architectures - An Approach for Self-Managing Systems

Giovanna Di Marzo Serugendo¹, John Fitzgerald², Alexander Romanovsky², and Nicolas Guelfi³

¹ School of Computer Science and Information Systems, Birkbeck College, London, UK
`dimarzo@dcs.bbk.ac.uk`

² School of Computing Science, University of Newcastle, NE1 7RU Newcastle upon Tyne, UK
`{John.Fitzgerald}{Alexander.Romanovsky}@newcastle.ac.uk`

³ Software Engineering Competence Center, University of Luxembourg, Luxembourg
`Nicolas.Guelfi@uni.lu`

Abstract. We argue that principles from the design of dependable software, especially separation of concerns and the use of formality, can be applied beneficially in the construction of self-managing systems. We illustrate this approach by presenting an experimental architecture for dynamic and resilient computer-based systems which utilises component metadata to govern reconfigurations in accordance with formally stated policies. Initial experiments with the architecture are described. We argue that the architecture describes a self-organising system and, further, provides a basis for self-managing systems.

1 Introduction

Large network-enabled computing systems are becoming integral to society, security and economy [25] and are likely to become more pervasive as technology is embedded in a wider range of products in the environment [11]. The openness and flexibility of such systems pose challenges to the maintenance of predictable levels of dependability, but also provide opportunities for self-organisation and self-management in response to threats. Delivering architectures for systems that are predictably dependable, yet open and flexible, is a significant challenge, albeit only part of the wider interdisciplinary response that is required [16].

The response to this challenge from the software engineering community addresses two main issues: *system architectures* that contain the potential for reconfiguration in response to events, and *formal descriptions* of components and interaction mechanisms that are strong enough to allow prediction of the dependability characteristics of the resulting dynamic systems. Here, a software architecture is a high-level structure of a software system described in terms of components and component interactions. A key aspect of the architectures for predictable dependability is *separation of concerns*: the code implementing a component is separated from the more abstract formal description of its functionality; the semantics of the architectural connectors are independent of the particular formal specifications of the components.

In the field of self-organising and self-managing systems, we observe two trends. On the one hand bio-inspired algorithms and models are applied for developing self-organising and self-managing systems. These algorithms and models provide a high-level of robustness and adaptation to the corresponding applications, but usually they are not driven by software engineering concerns, such as validation or correctness. On the other hand, recent proposals for self-managing systems architectures tend to go in the direction of adaptive software architectures. Self-organising software architectures are being defined in order to provide infrastructures where “components automatically configure their interactions in a way compatible with an overall architectural specification” [13]. Liu et al. [19] propose a framework enabling the development of individual autonomic components, as well as their dynamic composition and management. This framework relies on the separation of the components’ computational behaviour from interaction and organisation aspects. Similarly, the CASA approach [21] proposes and implements the idea of an application’s contract described independently of the autonomic application itself. Such a contract serves as a basis for adaptation policies at run-time.

In this paper we argue that building on software engineering approaches, either as an alternative to bio-inspired techniques or as a complementary approach, may enhance the predictability of self-organising systems, especially self-managing systems which must be resilient.

We first motivate our work by describing the challenge of achieving dependability and resilience in open and dynamic systems (Section 2). We present a model architecture for resilient dynamic systems (Section 3) based on the separation of concerns between the code that implements individual components and the metadata that describe their functional, and non-functional properties. This separation allows metadata to be used at run-time to permit adaptation, resilience, and self-management. We argue that such a system has characteristics of self-organisation (Section 4) and that the engineering approach taken in the model enables self-management (Section 5). Finally, we consider in a more generic way the relationship among self-organisation, emergent behaviour and self-management (Section 6).

2 Dynamically Resilient Computer-based Systems

Our work is motivated by the need to provide predictable dependability in new generation open and dynamic computing systems. By *dependability* we mean the ability of the system to deliver a service that can justifiably be trusted [3]. Central to this definition is the notion that it is possible to provide a justification for placing trust in a system. In practice this justification often takes the form of a dependability case which may include test evidence, development process arguments and mathematical proofs.

Future systems, based on network capabilities, wireless and nano-technologies, will be open and dynamic. A key characteristic is that they will allow *dynamic binding*, i.e. the selection and use of components at run-time. They will therefore not consist simply of components selected during an off-line design activity. Instead, they will be open to components arriving, departing or being modified. They will be dynamic in order to provide services on a continuous basis, and to do so even when components or the environment change.

Resilience is the ability of a system to maintain dependability while assimilating change without dysfunction. The characteristics of being open and dynamic mean that these systems can, and should, evolve by dynamically adapting to changes in components, infrastructure and environment (including accidental and malicious behaviours), as well as by more conventional deliberate off-line design, using established design and validation techniques e.g. to fabricate and install updates.

Some technologies already exist to help us build and deploy dependable computing systems. They rely on evolution through deliberate design, because the verification of resilience involves analysis that has generally been done off-line prior to deployment. However, next generation systems open the possibility of making dependability-maintaining adaptations at run-time. Consider an example scenario:

Emergency services use information and communication technologies in the environment to coordinate a response to a major road crash. Many different components are involved including on-board computers in rescue vehicles (and maybe in the crash vehicles), communications providers and sources of data (GPS, UMTS, GPRS, GSM). In this highly volatile environment, one component, a GPS data source, for example, may degrade, perhaps by losing availability.

One classical approach [3, 24] to the provision of dependability here would involve a design-time decision to use a fault tolerance strategy such as deploying a fixed number of diverse but compatible GPS sources instead of relying on a single one (Figure 1 shows a compound structure of three GPS sources). The selection of the component services would be based on information about the available services such as known availability levels and failure rates, and formal descriptions of the functionality provided by the components. Such classical approaches, when applied in a more open and dynamic environment, restrict dynamic flexibility. In the scenario, for example, the architectural decision to employ several diverse GPS services has to be made using the information available at design time. At run-time, a different set of compatible services may be available, allowing service to be maintained either at lower cost or with reduced degradation.

In an alternative, more resilient, approach the components that rely on the GPS service could reconfigure dynamically (Figure 2). There are many options, and they could pursue a policy of trying various alternatives: they could switch to an alternate GPS with higher availability, or use another service in parallel with the low availability one, or ultimately signal a failure. The main point is that the system could evolve dynamically to offer continued, if necessary (predictably) degraded, service using the resources available at the time the negative event is detected.

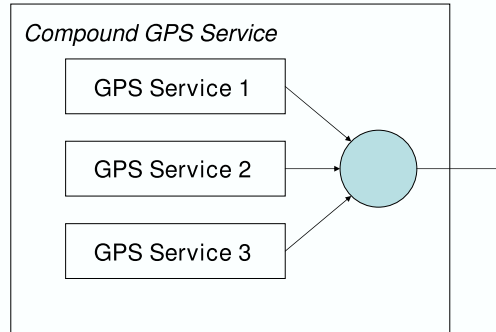


Fig. 1. Design-time solution: fixed choice of parallel GPS services

Our challenge is therefore to provide system architectures that permit dynamic decision-making about the selection and use of components in order to ensure predictable dependability.

Self-organising and self-managing systems The development of self-organising and self-managing systems raises several issues that are similar to those in resilient dynamic computing systems. Self-organising systems must adapt dynamically to environmental changes by (re-)organising themselves, this is similar to the dynamic binding mentioned above. In addition to (re-)organisational needs, self-managing systems are required to cope with specific tasks, such as ensuring seamless integration of new or updated components, (possibly degraded) functioning despite failure, and security aspects. This latter requirement, among others, entails adaptation to, and development of, corresponding policies at run-time. Our main argument here, later developed in this paper, is that software architectures for dependable resilient systems, in addition to having self-organisation properties, are also suitable for building self-managing systems.

3 An Architectural Model for Dynamically Resilient Systems

In this section, we present an architectural approach to the problem of maintaining the dependability of computing systems in the face of change. We give an overview of an architectural model that exploits component metadata to support decision-making and reconfiguration. We briefly describe some of the characteristics of an experimental implementation of the main parts of the model. We then indicate the central role of formal specifications in such a model. Finally, we present two implementations works, acting as proofs of concept, and describe preliminary experiments in self-management.

3.1 A Model for Dynamically Resilient Systems

As suggested by the example scenario above, certain features are essential to providing dynamic resilience in the open and flexible systems that are envisaged in the future. Perhaps the key feature is the availability at run-time of *dependability metadata* – information about system components, sufficient to govern decision-making about dynamic reconfiguration. Such metadata can be used to guide reconfiguration in accordance with policies (e.g. to increase the number of alternate services if availability starts to decline). We will call these policies *reconfiguration schema*. Finally, we require an environment for the acquisition, maintenance

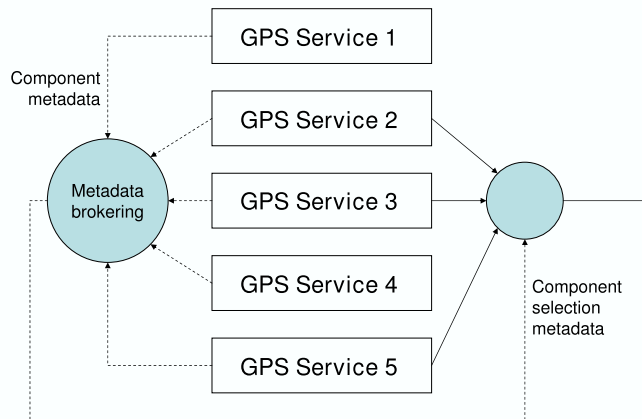


Fig. 2. Dynamic solution: run-time metadata-based selection of component services

and publication of metadata, including services for performing reconfigurations in accordance with policies. We call this the *component management environment*. These three main features are summarised in Figure 3. Each component is considered in more detail below.

Dependability Metadata *Metadata* are distinct from the data used by components in the course of their normal operation, and distinct from the code that implements component services. Rather they are data describing functional and non-functional properties of components themselves. We will use the term *dependability metadata* to refer to metadata that is relevant to system dependability.

Examples of *non-functional dependability* metadata include: availability or reliability measures (e.g. mean time to fail, mean time to repair); description of needed resources (CPU, network bandwidth, memory) for the component to work properly; or trust-based information on other components.

Examples of *functional dependability* metadata include logical specifications of component behaviour, for example, characterisation of known component failure modes or, more sophisticated still, formal specifications of pre-conditions on the use of a service and post-conditions characterising the service response to a valid input. These metadata are similar to the information used to make design-time decisions, but here we consider them as data that may be published and maintained at run-time.

Reconfiguration Schema Reconfiguration schemas are descriptions of metadata-based changes to the running system, which may include structural alterations. For example, a policy may be to attempt to find a replacement component for one whose failure rate exceeds a threshold, or if none exists to find two components of lower reliability that can be used in harness together. Such policies include (re-)configuration aspects, as described above, as well as security related policies, such access constraints, or service delivery conditions. Reconfiguration schema, when enacted, use search, reasoning and reconfiguration services provided by the underlying component management environment.

Component Management Environment The run-time environment follows a service-oriented architecture. It must provide essential services, including publication, acquisition and maintenance of metadata; execution of metadata oriented components; reasoning over metadata; run-time adaptation of components in response to operational changes including physical faults, design faults, user mistakes and attacks as well as component upgrading and gradual changes such as degradation of sensor data.

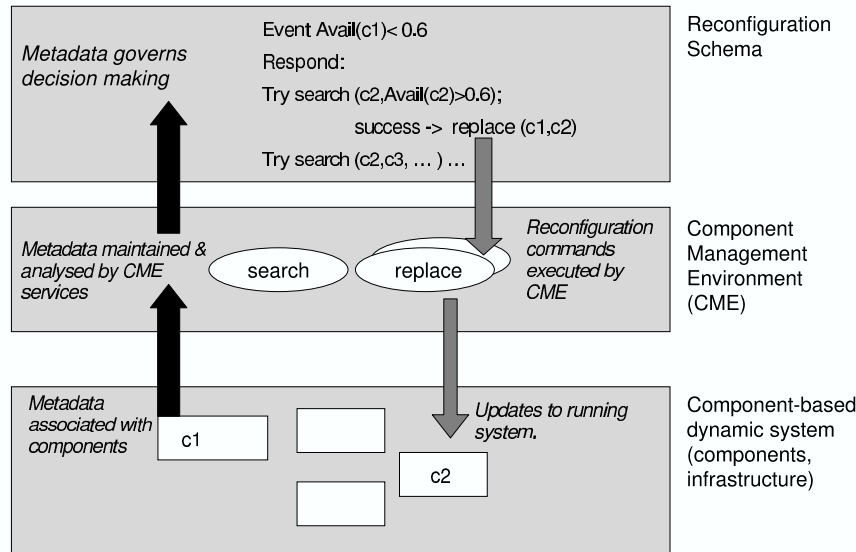


Fig. 3. An architecture for dynamic resilience

Experience with previously developed environments allowing dynamic replacement of code [15] or communication of components through semi-formal specifications [23], show that in order to avoid stopping applications running on top of the environment once the environment itself needs an update, it is important that the underlying run-time environment is itself as resilient as possible and that its updates are done seamlessly for the supported applications. This implies that the different run-time environment functionalities have to be developed, as far as possible, as different (independent) components, or services. The services constituting the run-time environment identified so far are illustrated in Figure 4:

Core Execution Part This component supports retrieval, activation, and execution, on the basis of metadata information, of components of the application's components or of the run-time environment services. The core execution part activates the services described below for realising its functionality.

Metadata Publication Service This service allows an application's components to register and publish its metadata (both functional description and value-based).

Lookup/Request Service This component retrieves and identifies a set of components corresponding to a specific metadata information request.

Metadata Reasoning Tool Closely linked with the Lookup Service, this service performs tasks related to the processing of dependability metadata such as comparison/matching of metadata, determination of equivalent metadata information, composition of metadata, etc.

Metadata Registry This service stores published metadata, and maintains the link with the corresponding component.

Metadata Maintenance / Acquisition Service This service updates the metadata Registry on the basis of availability of components.

General Reconfiguration - Replacement Strategies Service This service manages the list of components, seamlessly activates or connects the ones that will be used and which correspond to a resilient-based metadata request, disconnects the ones which are no longer used or available, possibly following a specific strategy. This service supports reconfiguration schemas.

Dynamic Replacement of Code Service Provides additional support for data transfer in case of dynamic replacement of code of an application’s component.

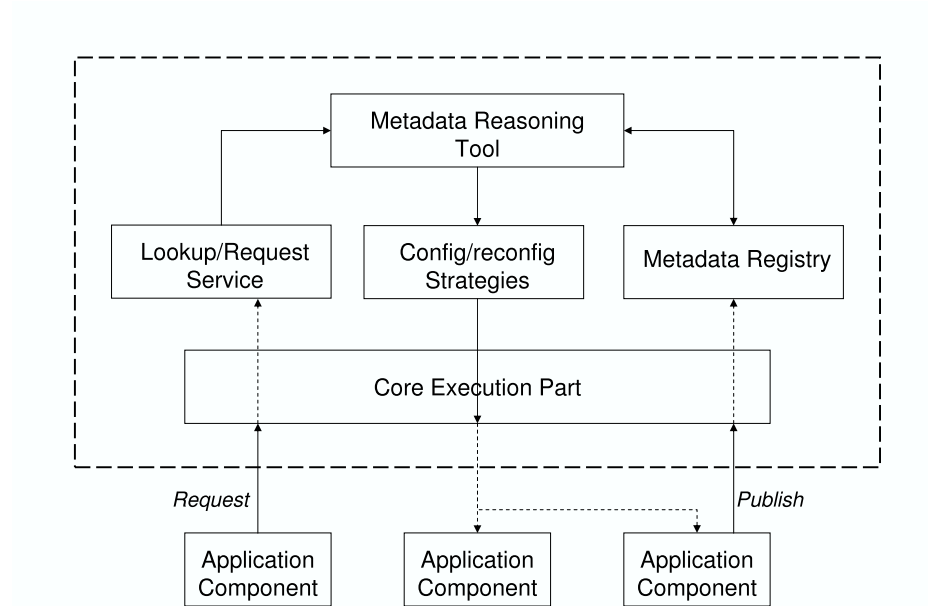


Fig. 4. Component management environment (Metadata acquisition and code replacement services not shown)

3.2 The Role of Formal Specifications

Formal descriptions play a significant role in the model described above. Respecting the separation of concerns principle, we identify a need for formal descriptions of dependability metadata, composition and reconfiguration schema. A description is regarded as formal if both its syntax and semantics are defined to a very high level of rigour, enabling machine-assisted analysis of properties up to and including formal mathematical proof.

Describing Metadata It may be helpful to distinguish between value-based metadata, such as availability distributions, probabilities of failure etc., and functional description metadata such as pre-conditions, post-conditions and descriptions of known failure modes.

Value-based metadata typically quantify some non-functional property and may be expressed as structured values drawn from a possibly ordered collection. Resilience mechanisms employing such metadata will allow explicit reasoning about reliability, performance, quality of service, availability of each component followed by system run-time evolution. This includes retrying after a dynamically-chosen period of time, signalling a failure when the component does not deliver the service of the required quality, switching from the less reliable component to a more reliable one, and, more generally, applying dynamic mechanisms based on the recovery block scheme [24] or employing a dynamic set of components as versions in N-version programming [2].

Component specifications may change as components evolve, but resilience may be maintained if it is possible to reason dynamically about the functional properties of components, including abnormal behaviours. Possible responsive adaptations include introducing wrappers or filtering mechanisms that compensate for changed characteristics of a component. An example might be a geographical route planning service that, over time, is known to be unreliable for destinations in a certain region. This is reflected in its metadata which would be updated to carry an advisory pre-condition on the service invocation. A possible reconfiguration schema would be to react to the change by building a wrapper that traps attempts to use the service outside the pre-condition and either return a known message, or apply to an alternative service that is known to work in the region, or even to apply to an alternative service as well as the original and compare the results.

Such dynamic resilience mechanisms require component specifications as metadata, including both specifications of services offered and services required. These can be given by logical expressions of pre/post-conditions (rely/guarantee conditions [10] in some cases), potential failure behaviours and responses when the components are used outside their pre-conditions. These are a very different form of metadata from the value-based examples. In particular, the on-line reasoning required to deal with such functional description metadata requires more complex logic than may be the case for value-based metadata.

There has been so far relatively little examination of resilience mechanisms that exploit component specifications as metadata in the dynamic system. However, the potential for publication of component specifications as metadata exists in several modern system architectures (such as service-oriented architectures [12]). Furthermore, advances in automated reasoning such as model-checking techniques [5], although generally only applied to design-time evolution, raise the potential for dynamic reconfiguration based on this metadata, at least for constrained cases, and make this an area well worth studying as basic research.

Describing Composition Within an architecture, system components are composed by connectors. In a web services environment, for example, the individual service invocations that make up an application are composed by means of combinators defined in a workflow language (e.g. parallel or serial composition). If a reconfiguration is to take place, it is necessary to know what the effects of the reconfiguration may be on the dependability properties of interest. In our GPS example, it is necessary to be able to predict the effects of parallel composition on service availability, in order to determine how many GPS services should be used, and which ones. This implies that the composition mechanism should have a semantics, at least in terms of the metadata of interest. This semantics is built in to the metadata reasoning tool, and so should be formal. For some existing architectural description languages [26, 1], such theories already exist, although they tend to deal primarily with the composition of functional rather than QoS properties.

Describing Reconfiguration Schema For many applications, we require that systems should be predictably dependable in the sense that degraded modes of operation and the circumstances in which they occur are understood in advance of deployment. This requirement for predictable dependability tends to lead towards the use of formal descriptions of system requirements, structural and security policies, as well as specifications of components. Reconfiguration schema bring together metadata and composition theories to describe the trigger conditions under which reconfigurations occur, and the reconfigurations themselves. Schema should be formal in the sense that their behaviour may be analysed in advance of deployment. In particular, the effects of interactions between multiple policies should be susceptible to some level of prediction.

3.3 Proofs of Concept

Proof of concept studies are being conducted for the metadata-based reconfiguration approach described above using service-oriented architectures. We describe two such studies. Study 1 focuses on the gathering of metadata and its use to reconfigure workflows in e-Science. In parallel, Study 2 concentrates on the issues of dynamic reconfiguration using the component management services described in Section 3.1.

Study 1: Acquisition and Use of Metadata In this preliminary work we have developed a tool that records in a database the dependability metadata derived from continuous observations of Web services.

The tool¹ measures the dependability of Web services by acting as a client to the Web service under investigation. It monitors a given Web service by tracking the following characteristics:

- Availability: The tool periodically makes dummy calls to the Web service to check whether it is running.
- Functionality: The tool makes calls to the Web service and checks the returned results to ensure the Web service is functioning properly.
- Performance: The tool monitors the round-trip time of a call to the Web Services producing and displaying real time statistics on service performance.
- Faults and exceptions: The tool logs faults and exceptions during the test period of the Web Service for further analysis.

We have applied this tool to monitoring two services implementing an algorithm used in the bioinformatics domain to search for nucleotide or protein sequences that are similar to a given query sequence.

We have been analyzing existing workflow languages used for composing e-science Web service applications and shown that it is possible to perform a simple reconfiguration of a workflow (specifically a SCUFL workflow developed in the Taverna environment²) by selecting a service or composition of services on the basis of availability metadata, to achieve the desired probability of successful completion of the task specified in the workflow. The metadata were collected and updated by the monitoring tool.

Study 2: Reconfigurable Service-oriented Architecture A restricted version of the architecture described in Section 3.1 has been implemented in a service-oriented framework. It supports essentially functional description of services, and a very limited form of non-functional QoS description. The underlying infrastructure is service-oriented middleware allowing registration of service descriptions and services request, matching of these descriptions, and seamless binding of components.

Principle. A service providing entity *registers* its specification to the run-time middleware that stores the specification in some repository. An entity requesting a service specifies this service through a specification, and asks the run-time middleware to *execute* a service corresponding to the specification.

Once it receives an execute request the run-time infrastructure activates a specification matcher that determines which of the registered services is actually able to satisfy the request (on the basis of its registered specification). The specification matcher establishes the list of all services whose semantics corresponds to the request.

Implementations. Two different implementations of the above architecture have been realised. The first implementation has been realised for specifications expressing: signatures of available operators whose parameters are Java primitive types; and required quality of service. Both operators name and quality of service are described using keywords. The resulting environment, a middleware called LuckyJ, allows server programs to deposit a specification of their own behaviour or of a requested behaviour at run-time. In the LuckyJ environment activation of services occurs anonymously and asynchronously. The service providing entity and the service requesting entity never enter in contact, communication is ensured by the LuckyJ middleware exclusively. The requesting entity is not blocked waiting for a service to be activated. The LuckyJ environment only allows the description of basic specification relying on ontology (keywords) shared among all the participating services [22]. Even though LuckyJ allows purely syntactical specifications, it nevertheless proved the viability of the approach under the form of a service-oriented architecture, and its usefulness for dynamic evolution of code.

In order to remove the need for interacting entities to rely on pre-defined keywords, a second implementation of the above architecture has been realised (see Figure 5). This architecture allows entities to carry specifications expressed using different kinds of specification language, and is modular enough to allow easy integration of additional specification languages [7, 8]. This architecture supports simple primitives for an entity to register its specifications or to request a service, and for the environment to execute the corresponding requested code once it has been found.

The current prototype supports specifications written either in Prolog, or as regular expressions. However it cannot check together specifications written in two different languages. In the case of Prolog, the

¹ <http://www.students.ncl.ac.uk/yuhui.chen/#Download>

² <http://taverna.sourceforge.net>

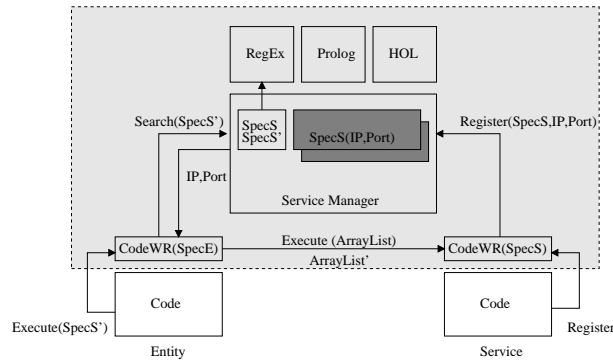


Fig. 5. Study 2: Semantic Service-Oriented Architecture

middleware calls SWI Prolog tool to decide about the conformance of two specifications, in the case of regular expressions we have implemented a tool that checks two regular expressions.

Discussion. Specification matching [20] encompasses *retrieval* of a component from a software library based on its semantics; *reuse* of a component from a software library in order to adapt it to the current needs; *substitution* of a component by another one without affecting the observable behaviour; and *subtyping*. The implementations described above are concerned with retrieval of a component whose semantics (registered specification) satisfies a query (specification request). There is no obligation that both specifications are equivalent; it is sufficient that the selected service specification *implies* the specification request.

In our current implementation using Prolog, specifications are registered as Prolog facts and rules, while specification requests are Prolog queries. In the case of regular expressions, the registered specification must match as a regular expression the specification request (but not necessarily the opposite).

3.4 Initial Self-Management Experiments

The above described implementations, even though serving as a proof-of-concept, have been turned to be useful to carry on the following self-managing functionalities.

Self-Configuration. Study 1, about the acquisition and use of metadata representing the dependability of diverse web services in the bioinformatics domain, has paved the way to building a self-configurable architecture which will be able to deal with new services by observing them and collecting metadata describing their characteristics for some period of time before they become available for use. When supported by the appropriate middleware services this feature will allow all basic activities related to self-configuration to be conducted seamlessly and without any involvement of the e-Scientist. In addition, Study 1 has shown the interest of the approach of separately describing execution flows and reconfiguration schemas to dynamically adapt the system to high-level user needs, in this case the needs of the scientist using the system.

For the particular case of automatic and seamless integration of new components, initial experiments with Study 2 have proven useful for dynamic run-time evolution of code [22]. Indeed, without stopping any specific entities or the whole system, it has been possible: to add in the system and seamlessly use additional features; to seamlessly replace updated entities without the calling entities noticing the replacement (even during a call). Since a specification request is the only element necessary for activating a service, inserting a new functionality then simply consists in registering its corresponding service to the middleware. Replacing an updated entity consists in having both the old and the new entities present at the same time in the system, and if necessary to transfer the state of the old entity to the new one before stopping the updated entity.

Self-Optimisation. Study 1 demonstrated that it was possible to choose the most suitable Web service or to employ/combine several Web services if the availability of individual Web services was not good enough. In addition, in Study 1 we extend the workflow language with an additional functionality to support a

search for the most dependable service out of the set of the service which are now monitored by the tool. This will allow us to add some degree of self-optimisation to the existing bioinformatics systems.

The architecture described for Study 2 implies that for each request, the middleware searches for all possible services realising the request. This turns out to be useful for self-optimisation. Indeed, as soon as a service realising a request is available (it simply needs to register itself), it can be selected by the middleware. If the request specifies that the most updated service is required, then the new service will be chosen. It is interesting to note that if the new service itself requires updated services to satisfy its needs, by a cascading effect a large part of the system will then use updated functionalities.

Self-Protection. Although no experiments on self-protection have yet been realised, a model for integrating trust information into specifications has been proposed as part of Study 2 [8]. In this schema, the specification is extended to incorporate trust and reputation information about a service provider, or about a client. This information, updated at run-time, then serves to accept or deny interaction requests.

4 Self-Organising Characteristics

Self-organisation essentially relates to the capacity of the system to spontaneously produce a new organisation in case of environmental changes (see Section 6 below). Software architectures relying on metadata as described in the previous section show self-organising features, since:

- components *dynamically bind* together under user requests or for pursuing their own goals;
- components *dynamically adapt* or conform to non-functional requirements, and policies of different nature (execution flows, resources description and access constraints).

In other words, the system taken as a whole (re-)organises itself as the result of components' own actions (without human or external assistance) under environmental changes: user requests, other components availability, changing policies, resources needs, etc.

The Nobel laureate Ilya Prigogine and his colleagues identified four necessary requirements for systems exhibiting a self-organising behaviour in thermodynamics environments [14]. In order to give a more precise description, we will identify for each point the corresponding behaviour in the software architecture model described above:

Mutual Causality “*At least two components of the system have a circular relationship, each influencing the other*” [6].

One of the aims of the architecture is to support semantic interaction of components that may not know each other in advance. This interaction necessarily implies a mutual causality in the sense that: on the one hand the service requester is satisfied or not by the service provider, thus affecting the requester's subsequent behaviour; and on the other hand the service provider has been activated under the service request. This may affect the provider in several ways: need to make some computation (resource consumption), possible modifications of local state due to the service request, etc.

Autocatalysis “*At least one of the components is causally influenced by another component, resulting in its own increase*” [6].

Interacting components may have an autocatalytic effect on the system, in the sense that a well satisfied request will cause the corresponding service to be solicited more than another service which may be less available or efficient. On the contrary, too much solicitation may lead to a degraded service satisfaction, and through dynamic regulation, according to specified reconfiguration schemas, more performant components will be selected to satisfy the request.

Far-from equilibrium condition “*The system imports a large amount of energy from outside the system, uses the energy to help renew its own structures (autopoiesis), and dissipates rather than accumulates, the accruing disorder (entropy) back into the environment*” [6].

Systems we consider are part of a highly changing environment. They need power supplies, network links, memory, etc. Components continually join and leave the system (inserted or removed by the environment, or no longer available because of lack of resources). The system then permanently and seamlessly integrates new or updated components, as well as environmental elements, such as additional CPU, power supply, etc. (autopoiesis).

Malfunctioning or non-responding components see their accesses or interactions denied from other components or from the system. In that sense, they leave the set of interacting components, maybe temporarily, and are then part of the environment, since they still consume some resources (entropy is pushed back into environment).

Morphogenetic changes “*At least one of the components of the system must be open to external random variations from outside the system. A system exhibits morphogenetic change when the components of the system are themselves changed*” [6].

Run-time evolution of components contributes to the continuous morphogenetic changes of interacting entities: components leave and join (appear/disappear); entities are upgraded (undertake software evolution). A given system may last for several years (old and recent portions of it working together) in a manner akin to the regeneration of animal cells.

In addition to changes in the components themselves, their executing environment may change: computing, networking, or storing resources may be removed, added, or changed.

5 Potential of Self-Organising Software Architectures for Self-Managing Systems

This section shows the interest of the above described architectural model for self-managing systems. More precisely, it describes the potential of such an approach for realising the four self-management concepts defining the autonomic computing view as explained in [17]. Essentially this potential relies on: dependability metadata for expressing functional behaviour of components, as well as their non-functional requirements; reconfiguration schemas for expressing different kinds of policies; and on the underlying run-time infrastructure (component management environment) supporting seamless interactions of components, dynamic binding, etc.

Self-Configuration. “*Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly.*” [17]

The notion of reconfiguration schema naturally serves to express high-level goals as stated by a (human) administrator. They constitute initial service requests triggering the rest of the system. At a lower level, service requests expressing high-level configuration policies may serve for automatic distribution of entities (e.g. placement on a Grid) of components participating in some computation. This leads to new reconfiguration schemas inserted into the system. Finally, at a local level, each component of an autonomic computing system may describe, through its non-functional dependability metadata, its own installation needs (e.g. CPU, Memory, or Network). In summary:

- High-level policies acting as high-level requests (goals) from human administrator (system installation needs) are expressed through reconfiguration schemas;
- High-level requests for configuration policies (Grid distribution) are expressed as a service requests in the form of reconfiguration schemas;
- Local-level configuration policies of individual components, express individual components’ installation needs (CPU, memory, etc.). They are described through non-functional metadata.

Self-Optimisation. “*Components and systems continually seek opportunities to improve their own performance and efficiency.*” [17]

The combined use of non-functional dependability metadata related to optimisation needs, and reconfiguration schemas for expressing optimisation policies serves self-optimisation purposes. Parameters to optimise may be described through metadata, and optimisation of parameters depending on the context may be expressed through reconfiguration schemas.

The underlying run-time infrastructure is then responsible for executing components' requests, while actually satisfying their requested optimisation parameters descriptions and optimisation policies. Since, at each service request, the underlying run-time infrastructure checks optimisation parameters and policies, the system then permanently improves its own performance given the current environmental context. For example, it uses updated components or more efficient available components for satisfying the ongoing requests; or it takes into account current environmental resources for providing efficient components executions. In summary:

- Non-functional metadata describe optimisation parameters;
- Reconfiguration schema describe optimisation of parameters depending on the context;
- Run-time infrastructure ensures the use of optimised service for satisfying the service requests.

Self-Healing. *“System automatically detects, diagnoses, and repairs localized software and hardware problems.”* [17]

Generally speaking, to make a system self-healing we need to apply appropriate fault tolerance techniques [18]. The choice here depends on the requirements, the resources available and the fault assumptions. Making evolvable and open systems self-healing requires advanced fault tolerance mechanisms which employ redundant resources in a dynamic and adaptable fashion. This typically involves dynamic system reconfiguration with deployment of new components.

The architectural model introduced in Section 3 for supporting dynamically resilient systems is a specific approach which enables building of self-healing systems capable of tolerating a wide range of faults, damaging events and changes in the systems, infrastructures and system environments. A dedicated architecture allows choices about the use of the specific fault tolerance mechanism and of the redundant resources to be made dynamically using metadata available in run-time.

Moreover, functional metadata serve for both detecting erroneous code (checking of code against functional specification), and for potentially replacing such erroneous code by an equivalent one (automatic generation of code). Indeed, from the one hand, the underlying run-time infrastructure may verify the adequacy of a code from its corresponding functional metadata (e.g. through proof-carrying code techniques). If recognised as erroneous by the run-time infrastructure, its functional and non-functional metadata is removed from the repository of available services (that service is no longer available). The whole system then automatically works with the remaining available services, maybe in a degraded manner, until a new code is inserted into the system and replaces the erroneous one. Alternatively, the architecture above can be extended to incorporate automatic generation of a code (recognised as erroneous) from its corresponding functional metadata description. In summary:

- The architectural model contains built-in support for dealing with a wide range of faults;
- The model can be extended for: 1. checking code against its functional metadata, thus detecting potential erroneous code; 2. using functional metadata as a basis for replacing erroneous code with another code having an equivalent functional metadata; 3. automatically generating correct code from functional metadata information of an erroneous code.

Self-Protection. *“System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent system-wide failures.”* [17]

The proposed architectural model arises from fault-tolerance, resilient and dependability concerns. Then, it inherently supports system resilience despite component or environmental failures, essentially through the notions of dependability metadata such as conditions regulating services delivery, and reconfiguration schemas, which naturally describe high-level security policies that have to be realised in the whole system.

Regarding malicious attacks, reconfiguration schemas may serve for expressing signatures of security attacks and response schemas to attacks, thus allowing the system to recognise and react to run-time attacks.

Additionally, self-regulating schema for failures and attacks can also be considered based on non-functional metadata. For instance, combining trust and reputation information within non-functional metadata may prove to be an efficient tool for self-protection [8]. Indeed, the use of trust allows to permanently adapt the whole system behaviour to the individual components' behaviour or responses to services requests. In summary:

- The architectural model contains built-in support for system failures through fault-tolerance related issues;
- Malicious attacks can be addressed through the use of reconfiguration schemas;
- Additional self-protection schemas, such as those based on trust and reputation can be incorporated through adequate descriptions in non-functional metadata and reconfiguration schemas.

6 Self-Organisation, Emergent Behaviour, Self-Management and how they relate together

This section intends to highlight and to contribute to the understanding of the three intimately linked, but different, notions, which are self-organisation, emergent phenomena, and self-management.

Self-organisation. We can distinguish several definitions relating to different kinds of self-organisation [9]:

- *Swarm Intelligence.* Derived from studies on insects, the theory of stigmergy states that coordination and regulation tasks are realised indirectly on the basis of information deposited into the environment, without central control. *Self-organisation is here the result of behaviour occurring from inside the system (from the social insects themselves).*
- *Decrease of Entropy.* Derived from thermodynamics studies, it has been observed that open systems exhibiting a self-organising behaviour under external pressure decrease their entropy when external pressure is applied. *Here self-organisation is the result of a pressure applied from the outside on the system.*
- *Autopoiesis/Cells.* The notion of autopoiesis (meaning self-production) is the process through which an organisation is able to produce itself. Autopoiesis applies to closed systems made of autonomous components whose interactions self-maintain the system through the generation of system's components, such as living systems (cells, or organisms). *Here self-organisation is more related to self-maintenance through self-generation.*

In the different cases above, self-organisation is realised with different mechanisms, and occurs under different conditions. Developing an artificial self-organising system implies being aware to which kind of self-organisation the system belongs. However, in all these cases, the definition essentially states that: “Self-organisation is the capacity to spontaneously produce a new organisation in case of environmental changes”.

Emergent behaviour. An emergent phenomenon is a structure (pattern or function) not explicitly represented at a lower level but which appears at a higher level. Two cases can be distinguished:

- The emergent phenomenon is an observed pattern or function which has no causal effect on the system itself (e.g. stones ordered by sea);
- The emergent phenomenon is an observed function which has a causal effect on the system. This function can be desired or not, but in both cases it has an effect.

It is interesting to note that emergence, per se, is not always needed or required or good for the system. Indeed, as stated by [4], emergent phenomenon is not necessarily a good thing for society (especially in the case of self-interested agents). The optimum order for the society can actually be bad for individuals or for everybody. For instance, prisons generate criminals that in turn feed prisons.

Self-organisation vs Emergent Phenomena. Self-organisation can be independent of emergent behaviour. Indeed, self-organisation can happen without emergent behaviour. For instance, in the case of self-organisation occurring under internal central control, the possibly observed new organisation is then fully deducible from the central entity. Similarly, emergent behaviour can happen without self-organisation, when there is no (re-)organisation, as for instance with emergent patterns on animals. These patterns have no causal effect on the whole system. There is no re-organisation of the stripes or of the cells producing the stripes.

Usually self-organisation and emergent behaviour appear together in the case of dynamic self-organising systems, for a system working with decentralised control, and with local interactions among the individual components.

Engineering Solutions for Self-Organisation and Emergent Behaviour. Current solutions for engineering self-organising systems with possibly emergent behaviour essentially consist in reproducing natural self-organising mechanisms, such as mechanisms inspired by biology or social behaviour (insects, humans, etc.). This leads to concrete solutions and middleware based on direct interactions, reinforcement, adaptive agents, cooperation [9], but without paying attention (for the moment) to software engineering issues.

Strengths of these solutions are their robustness, adaptivity, and the fact that they are made of simple individual components. *Limits* of bio-inspired solutions reside in: the control of emergent phenomena, and in the correct design of those systems.

Self-management characteristics. We identify the following characteristics for self-managing systems deriving from definitions provided in Section 5:

- Self-managing systems mostly work under *decentralised control* since one of the goals of these systems is to autonomously respond to high-level policies coming from a human administrator, to seamlessly cope with failures, malicious attacks, etc.;
- Self-managing systems permanently adapt to changes either through self-optimisation, or self-healing. This leads to a permanent *re-organisation* of the whole system;
- Individual components *interact locally* with each other, and have a local knowledge about their environment;
- Self-management itself is the *desired emergent behaviour* that we would like to observe. In addition, self-management has a direct impact on the system itself: components are chosen based on their efficiency or availability, or put apart if considered erroneous or malicious, etc.

It appears then, that self-managing systems simultaneously exhibit self-organisation under decentralised control (the first three points above) *and* emergent behaviour with a causal effect on the system (the final point).

In addition, we would like to distinguish the following three aspects that probably make self-managing systems even more complex than self-organising systems with emergent behaviour:

- Self-managing systems have to manage themselves;
- In addition to managing themselves, these systems manage any additional resource the system handles (the underlying self-managed system);
- Self-managing systems need to interact with a human administrator, in two directions. From human to system: the high-level goal stated by the human administrator has to be correctly decomposed into low-level goals that individual components have to realise. From system to human: a coherent global information has to be furnished to a human administrator despite the fact that it is produced by local decentralised information furnished by individual components.

For instance, a self-managing distributed operating system has to provide: 1. self-management tasks such as dynamic upgrading, parameter tuning, or self-protection; 2. operating systems management tasks for a distributed large-scale computer, e.g. optimising resource usage, supporting distributed data processing applications, and distributed on-line delivery of end-user services; 3. to satisfy high-level goals expressed as policies by some human administrator.

Issues for Engineering of Self-Managing Systems. We briefly list here issues for self-managing systems which include: issues related to building dependable self-organising systems with emergent behaviour and issues specific to self-managing systems.

- *Self-Organisation and Emergent Behaviour Issues.*

- Software engineering of systems which are both self-organising and having emergent behaviour;
 - To define a global goal (self-managing functionalities) and to design local behaviour (making the global expected behaviour to happen);
 - Control / Design of decentralised behaviour. Good properties have to emerge, while bad properties must be avoided. In addition, control and emergence are notions in tension with one another.
 - To identify the kind of self-organisation needed for the self-managing system to develop;
 - To clearly identify, take into account and perhaps design the environment.
- *Self-Managing Systems Issues*. These issues are mostly related to the three above mentioned aspects of self-managing systems. The system has: to correctly manage itself; to correctly manage available environmental resources; and to provide correct response to human interaction.

7 Conclusion

In this paper we have presented a model for a dependable self-organising architectural approach for addressing self-managing systems development. The basic idea relies on the separation of concern of code from functional and non-functional description of individual components, and the possibility of separately expressing different kinds of policies; and on the use of of a service-oriented middleware that seamlessly supports components execution respecting their provided descriptions and the established properties.

Although at the stage of a model, we have shown, through two initial proof of concepts experiments, the viability of the approach for self-managing systems. Future work will essentially concern the choice of formal specifications for describing policies and component descriptions.

This paper also argued that dependable self-organising architectures may prove beneficial in constructing self-managing systems either as an alternative to purely bio-inspired techniques, or more likely in combination with them, by supporting bio-inspired algorithms inside the architecture.

References

1. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering*, 6(3):213–249, 1997.
2. A. Avizienis. The N-Version Approach to Fault Tolerant Systems. *IEEE Transactions on Software Engineering*, SE-11:1491–1501, 1985.
3. A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.
4. C. Castelfranchi. The theory of social functions: challenges for computational social science and multi-agent learning. *Journal of Cognitive Systems Research*, 2(1):5–38, 2001.
5. S. Chaki, N. Sharygina, and N. Sinha. Verification of evolving software. In *Proc. Workshop on Specification and Verification of Component-based Systems, 12th. ACM Symposium on Foundations of Software Engineering 2004*, 2004.
6. N. S. Contractor and D. R. Seibold. Theoretical frameworks for the study of structuring processes in group decision support system - adaptive structuration theory and self-organizing systems theory. *Human Communication Research*, 19(4):528–563, 1993.
7. M. Deriaz and G. Di Marzo Serugendo. Semantic service oriented architecture. Technical report, Centre Universitaire d’Informatique, University of Geneva, Switzerland, 2004.
8. G. Di Marzo Serugendo and M. Deriaz. A social semantic infrastructure for decentralised systems based on specification-carrying code and trust. In D. Hales and B. Edmonds, editors, *Socially-Inspired Computing*, 2005.
9. G. Di Marzo Serugendo, M.-P. Gleizes, and A. Karageorgos. Self-Organisation and Emergence in MAS: An Overview. *Informatica*, In press, 2006.
10. C. H.̇ Duarte and T. Maibaum. A rely/guarantee discipline for open distributed systems design. *Information Processing Letters*, 74:55–63, 2000.
11. D.L. Estrin, editor. *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers*. Computer Science and Telecommunications Board, National Academy of Sciences, Washington, D.C., 2001.
12. J.Š. Fitzgerald, S. Parastatidis, A. Romanovsky, and P. Watson. Dependability-explicit computing in service-oriented architectures. In *Supp. Volume of Proc. Intl.z Conf. on Dependable Systems and Networks*, pages 34–35, 2004.
13. I. Georgiadis, J. Magge, and J. Kramer. Self-organising software architectures for distributed systems. In *WOSS’02*, 2002.

14. P. Glansdorff and I. Prigogine. *Thermodynamic study of structure, stability and fluctuations*. Wiley, 1971.
15. M. Hicks, J. Moore, and S. Nettles. Dynamic software updating. *ACM SIG-PLAN Notices*, 36(5):13–23, 2001.
16. C.B. Jones and B. Randell. Dependable Pervasive Systems. In R. Mansel and B. S. Collins, editors, *Trust and Crime in Information Societies*, pages 59–91. Edward Elgar Publishing, Cheltenham Glos, UK, 2004.
17. J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003.
18. P.A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice (Second Revised Edition)*. Springer-Verlag, 1990.
19. N. Liu, M. Parashar, and S. Hariri. A component-based programming model for autonomic applications. In J. Kephart and M. Parashar, editors, *International Conference on Autonomic Computing (ICAC'04)*, pages 10–17. IEEE Computer Society, 2004.
20. A. Moormann Zaremski and J. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.
21. A. Mukhija and M. Glinz. The CASA Approach to Autonomic Applications. In *5th IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2005)*, pages 173–182. IEEE Computer Society, 2002.
22. M. Oriol and G. Di Marzo Serugendo. A disconnected service architecture for unanticipated run-time evolution of code. *IEE Proceedings-Software, Special Issue on Unanticipated Software Evolution*, 2004.
23. M. Oriol and G. Di Marzo Serugendo. Disconnected Service Architecture for Unanticipated Run-time Evolution of Code. *IEE Proceedings-Software, Special Issue on Unanticipated Software Evolution*, 151(2):95–107, 2004.
24. B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, SE-1:220–232, 1975.
25. F.B. Schneider, editor. *Trust in Cyberspace: Report of the Committee on Information Systems Trustworthiness, Computer Science and Telecommunications Board, Commission on Physical Sciences, Mathematics and Applications, National Research Council*. National Academy Press, Washington, D.C., 1999.
26. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996. 242p. ISBN 0-13-182957-2.