A Generic Framework for the Engineering of Self-Adaptive and Self-Organising Systems

Giovanna Di Marzo Serugendo[1], John Fitzgerald[2], Alexander Romanovsky[2], Nicolas Guelfi[3]

[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, Newcastle upon Tyne, UK
John.Fitzgerald@newcastle.ac.uk,
Alexander.Romanovsky@newcastle.ac.uk
[3] Laboratory for Advanced Software Systems, University of Luxembourg, Luxembourg
Nicolas.Guelfi@uni.lu

Abstract. This paper provides a unifying view for the engineering of self-adaptive (SA) and self-organising (SO) systems. We first identify requirements for designing and building trustworthy self-adaptive and self-organising systems. Second, we propose a generic framework 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 an SA or SO system, thereby providing some assurance of its self-* capabilities. We show how this framework applies to both an SA and an SO system, and discuss several current proof-of-concept studies on the enabling technologies.

1 Introduction

Research into artificial self-adaptive (SA) and self-organising (SO) systems is flourishing, demonstrating that it is feasible to develop ad hoc self-* systems. However, if we are to build SA and SO ecosystems at a large-scale or professional level, it is important to tackle issues related to their design, development and control. Indeed the next question to answer is: how can we build *trustworthy* SA and SO systems? Trustworthiness encompasses dependability properties, plus evidence of dependability[4]. Thus, during a system's initial development, deployment and subsequent evolution, we must be able to provide assurance that emergent behaviours will respect key properties, frequently to do with safety, security or performance of the whole composed system, and that the human administrator retains control despite the self-* capabilities of the system. This paper is concerned with both parts of this question: what design-time techniques and architectures and what run-time infrastructures would be most appropriate for building reliable and controllable SA and SO systems?

We consider here the following definitions of SA and SO systems: "Self-adaptive systems work in a top-down manner. They evaluate their own global behaviour and

[4] Trustworthiness also requires acceptance by users, organisations and society at large. In this paper, however, we concentrate on the challenge imposed by the technical system.

change it when the evaluation indicates that they are not accomplishing what they were intended to do, or when better functionality or performance is possible. Self-organising systems work bottom-up. They are composed of a large number of components that interact locally according to simple rules. The global behaviour of the system emerges from these local interactions, and it is difficult to deduce properties of the global system by studying only the local properties of its parts." [1].

At first, SA and SO systems seem to be very different: SA systems tend to be more hierarchic and top-down driven; SO systems tend to be decentralised and bottom-up driven. However, there are some important points of contact between the two concepts. The need for allowing more "freedom" to self-adapting systems, by allowing a degree of decentralisation and self-organisation to the components, has already been advocated [2]. Self-organising systems with pure decentralised control should nonetheless provide assurance of their behaviour to potential customers or users prior to deployment and should allow control to be imposed by an administrator. Examples of systems already encompassing both SA and SO aspects are found in socio-technical applications involving both heterogeneous technical devices such as body or environmental sensors, PDAs, software, servers, and human users such as doctors, nurses, rescue teams, end-users, system administrators. Socio-technical systems encompass, among others, ambient intelligence and ubiquitous computing systems, emergency response or e-Health applications. Each actor (human or device) in such systems is an autonomous element. As a whole the system displays complexity, self-adaptation and self-organisation.

In this paper, we describe an attempt to unify these two views – trustworthiness of SA and SO – from an engineering perspective. We first expose requirements that need to be satisfied by such system architectures in order to make them amenable to the kinds of analysis required to provide for dependable systems design (Section 2). We then present elements of a generic framework supporting designers and developers of SA and SO systems, relating them to our identified requirements (Section 3). Sections 4 and 5 show how the framework applies to an SA and an SO system respectively. Finally, Section 6 discusses preliminary proofs of concept, and Section 7 provides some discussion. Section 8 briefly describes related work.

2 Engineering Requirements

From an engineering perspective, the systems that we consider here are distributed systems on a potentially large scale consisting of components which may be physical devices, services, or people. They have an emergent behaviour and have the capacity to respond autonomously to events within the system and in the environment. This dynamic character challenges traditional methods of engineering dependable systems, which typically rely on extensive static, design-time analysis to achieve a level of predictability. At the same time, it brings the potential to develop systems that use dynamic reconfiguration to remain resilient to threats. Thus, an engineering view of SA and SO systems must attempt to "square the circle" of achieving predictable, yet dynamic resilience in self-adaptive and self-organising systems.

A framework for trustworthy SA and SO systems must take account of a number of requirements arising from the engineering need to understand and validate system models during design, as well as the need to deliver a dependable level of service. Below we list these key requirements.

Autonomous individual components.

Robustness and self-* behaviour arise from the numerous (low-level) individual components constituting the core functionality of the system and from their (local) interactions. In SO systems, such components can be ant-like entities, agents, peers or cars. In SA systems, autonomic elements, autonomic managers, and any element of the supporting infrastructure (e.g. the service registry or sentinel monitoring mentioned in [3]) are all autonomous components.

R1: Components should be decoupled as far as possible so that it is possible to detect and respond to failure/unavailability without fundamentally harming the global system.

Interoperability.

The overall (global) behaviour of an SA or an SO system is obtained through the interactions of the individual components. In real-world scenarios involving open and dynamic systems, we must think at the individual components as heterogeneous both in nature and in design: different vendors will be providing autonomic elements; differently owned elements will want to participate in some self-organising systems (e.g. P2P, MANET, ambient intelligence scenarios). Interoperability lies at a semantic level and encompasses understanding of functional and non-functional (QoS/Constraints) capabilities, as well as coordination and interaction modes among components and between components and their environment. It should be noted that "off-the-shelf" components may have only poorly understood behaviours and hence weak specifications, entailing the use of protective wrappers [4, 5].

R2: Components should be decoupled from descriptions of their capabilities, QoS, Constraints, execution flows, interactions mode, or policies.

R3: Components should be decoupled from any underlying coordination infrastructure supporting the components interactions (e.g. decoupling the components from any shared blackboard, an event bus, etc.).

Self-awareness.

Self-* properties and behaviour arise from the capability of the system or its individual components to identify *by themselves* (internally) any new condition, failure, or problem; without specifically being instructed (from outside) by any human administrator. Self-awareness requires "sensing" capabilities and triggers "reasoning" and "acting". SA systems are currently thought of as equipped with monitoring, planning and plan execution capabilities at the level of the autonomic managers. SO systems sense their environment in different ways (artificial pheromone, configurations, neighbours, etc.), and take decisions accordingly (changing directions, role or links).

R4: Run-time capability of sensing/monitoring on-going activity at different levels (individual components, part or whole system).

R5: Run-time capability of reasoning and of acting/adapting at different levels (individual components, part or whole system).

R6: Run-time availability and usage of sensing/monitoring and acting/adapting policies at different levels (individual components, part or whole system).

Behaviour Guiding and Bounding.

The components of an SO system have their own local rules that direct their behaviour towards some optimum. SA systems have both local rules (at the level of the components) and global rules (at the different levels the system). We will refer to these rules or policies as *Guiding Behaviour*.

In addition to guiding the system towards optimal functioning, it is also important to introduce boundaries in the system behaviour limiting the scope of permitted actions but freely allowing decentralised adaptive behaviour of individual components inside the boundaries. For instance, a Grid environment may insert some limits in order to avoid a component to get all the resources; a trust-based system may have an immutable threshold below which no transaction is granted.

R7: Run-time availability and usage of individual and global goals under the form of policies (Guiding).

R8: Run-time availability and usage of environmental constraints policies (Bounding).

Development Process.

During the development process the analyst, designer and developers need in turn to define and examine different views of the system from abstract descriptions to concrete code or policies.

R9: Design-time description of expected system's / components' properties.

R10: Design-time description of self-* behaviour patterns.

R11: Design-time description of the different policies described above (R6, R7, R8).

R12: Run-time enforcement of policies (described by R6, R7, R8).

3 A Generic Framework

In this section we propose component technologies that may be capable of meeting the requirements imposed above. These have formed the basis of our proof of concept studies (Section 6).

Service-Oriented Architecture

Service-oriented architectures are becoming widely accepted as architectural solutions for systems involving autonomous components, dynamicity and heterogeneity [6]. A wrapper around the autonomous components let them become services while keeping their autonomous aspect. Middleware supporting services interactions can come into different flavours (coordination spaces favouring indirect communication (SO) but also discovery of services publishing and requesting services (SA)).

Addresses **R1**: handling of heterogeneous and autonomous components in a homogeneous, modular and in a loose coupling way.

Self-Describing Components/Services

As pointed out in [7], interoperability is fundamental when different service providers are involved in the same system. Decoupling components (software program) 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 or changing policies: such as replacing a failing autonomic manager with one that has equivalent characteristics.

Addresses **R2**: interoperability aspects driven by heterogeneous design.

Acquired, Updated and Monitored Metadata

Sensing and acting is a fundamental activity of both SA and 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 and the supporting infrastructure. Metadata is information about the running system (its components, infrastructure and environment). It is distinct from the data used by components in the course of their normal operation, and distinct from the code that implements component services. Metadata may convey functional information (e.g. pre/post-conditions, known component failure modes) and non-functional information (e.g. availability of resources, reliability/adaptability measures).

Addresses **R2**, **R4**, **R5**: published metadata supports interoperability; updated metadata values support self-awareness (sensing, reasoning and acting on the basis of metadata values).

SA and SO Architectural/Design Patterns

Architectural patterns describe high-level coordination techniques such as the notion of observer/controller [8], the notion of autonomic manager coupled with any autonomic element, or coordination through stigmergy or field-based structures [9].

Addresses **R3**, **R10**: description of (high-level) coordination/adaptation architecture.

SA and SO Adaptation Mechanisms

SA and SO Mechanisms are lower-level patterns defined at design-time whose purpose is to describe how SA/SO adaptation is triggered on the basis of metadata, e.g. switching coordination pattern, replacing a component with a functionally equivalent one if the performance of the component is too low. These patterns are implemented in specific applications through executable policies (see below).

Addresses **R11**: design-time specification of acting/adapting policies (in response to monitored data).

Executable Policies

As discussed in the previous section, policies come in many varieties, lie at different levels (component, system, interactions, environment), and have different scopes. Policies may include (re-)configuration aspects and may also include security-related policies, such access constraints, or service delivery conditions. Policies are based on monitored metadata; their enforcement at run-time triggers adaptation and implementation of SA/SO mechanisms. Essential policies are:

- Monitoring Policies (e.g. frequency, type of metadata monitoring);
- Guiding Policies (e.g. goal-driven or utility-driven behaviour);
- Bounding Policies (e.g. system/environmental constraints);
- SA/SO Mechanisms Policies (e.g. adaptation decision based on monitored metadata)

Addresses **R6**, **R7**, **R8**: description and run-time implementation of SA/SO mechanisms and policies described above.

Formal Languages and Specifications

In order to support predictable dynamic reconfiguration, metadata, component descriptions, and policies need to be available at run-time. The concept of "lingua franca" frequently advocated in the literature for solving interoperability issues could be used here for describing these different elements. It is likely that several different languages for the various elements above will be required. Each could be an extension of an existing language, or could be brand new, but each should be as "formal" as possible, in order to allow run-automated reasoning at the semantic level, for example in determining substitutability of services, acceptable degraded performance characteristics etc. The essential forms of specification are:

- Description of patterns;
- Self-description of components;
- Specification of metadata;
- Specification of policies.

Each of these forms of specification may be used in either design-time or run-time decision-making processes.

Design-time Activities

Figure 1 represents the design-time activities of the designer: during the analysis phase, properties of the overall system are identified, driven by these properties, the designer then selects the architectural patterns and adaptation mechanisms that the system will adhere to; she then instantiates the chosen patterns for the specific application, architecture and policies, designs the individual components, selects and describes the necessary metadata.

Addresses **R9**, **R10**, **R11**: identification of system's / components' properties, and corresponding patterns, and policies for enforcing them.

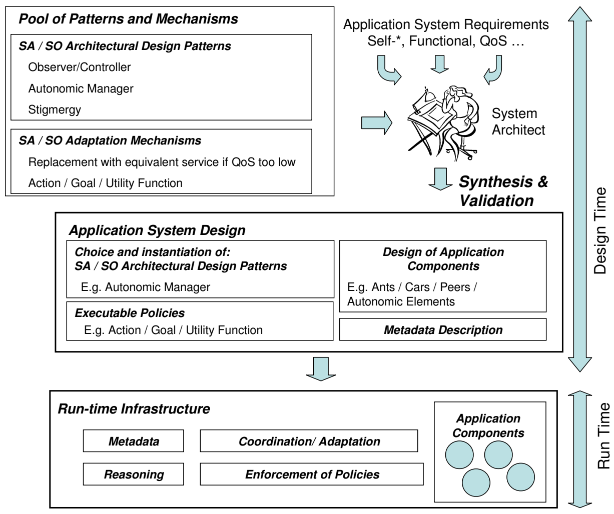
Pool of Patterns and Mechanisms
SA / SO Architectural Design Patterns
Observer/Controller
Autonomic Manager
Stigmergy
SA / SO Adaptation Mechanisms
Replacement with equivalent service if QoS too low
Action / Goal / Utility Function
Application System Requirements
Self-*, Functional, QoS …
System Architect
Synthesis & Validation
Application System Design
Choice and instantiation of: SA / SO Architectural Design Patterns
E.g. Autonomic Manager
Executable Policies
E.g. Action / Goal / Utility Function
Design of Application Components
E.g. Ants / Cars / Peers / Autonomic Elements
Metadata Description
Design Time
Run-time Infrastructure
Metadata
Coordination/ Adaptation
Reasoning
Enforcement of Policies
Application Components
Run Time

Fig. 1. Design-time vs Run-time

Run-time Infrastructure

The run-time environment itself supports a service-oriented architecture. 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. Figure 2 shows the run-time infrastructure:

– **Metadata** is stored, published and updated at run-time both by the run-time infrastructure (monitoring activities) or by the components themselves (sensing/acting). Different types of metadata are available: component descriptions (possibly including interface information), environment-related metadata (possibly supporting coordination), metadata related to either individual components (e.g. availability level, efficiency) or to groups of components.
– **Policies** are also 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 or component level. Components can react directly to a low-level policy taking account of current values of metadata.
– **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 provides tasks related to the processing of metadata, such as comparison/matching, determination of equivalence and metadata composition. They also encompass automated reasoning over policies and metadata. (Addresses **R4**, **R5**, **R12**).

– **Coordination/Adaptation.** This service implements the SA/SO architectural pattern chosen at design-time. It manages the list of components, seamlessly activates or connects the ones that will be used according to specified coordination/adaptation policies. It encompasses automated reasoning on adaptation policies. (Addresses **R3**, **R5**, **R12**).

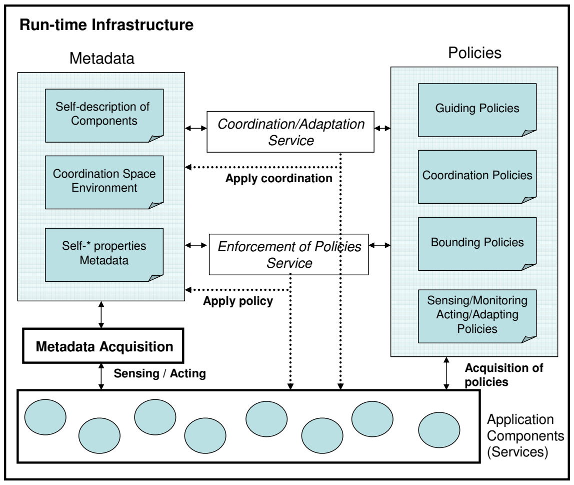
Run-time Infrastructure
Metadata
Self-description of Components
Coordination Space Environment
Self-* properties Metadata
Coordination/Adaptation Service
Apply coordination
Enforcement of Policies Service
Apply policy
Policies
Guiding Policies
Coordination Policies
Bounding Policies
Sensing/Monitoring Acting/Adapting Policies
Metadata Acquisition
Sensing / Acting
Acquisition of policies
Application Components (Services)

Fig. 2. Run-time Generic Infrastructure

It is worth noting that the run-time environment 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.

Generic services necessary to build such a run-time infrastructure encompass: a registry/broker that handles the service descriptions and services requests; an acquisition and monitoring service for the self-* related metadata; a registry that handles the policies; and a reasoning tool that enforces the policies on the basis of metadata.

4 Application to a Self-Adaptive System

We consider the application of our proposed framework to a simplified version of a well-known example of an adaptive system [10]. A finite set of resources is dynamically allocated between several (say two) applications. Each application provides a service, the demand for which varies over time. The performance of the applications depends on the demand and on the resources allocated to each, and is subject to Service

Level Agreements (SLAs) which are generally defined in terms of metrics expressing the quality of service provided to the consumer, e.g. maximum application-response time, or minimum application throughput [11]. The performance of the overall system depends on the performance of the individual applications with respect to their SLAs. The goal of the system as a whole is to *optimise overall system performance* given the set of SLAs governing the applications.

More precisely, according to Kephart and Walsh [10], two *Application Managers* (AM1 and AM2) each handle different resources (routers and servers). Each Application Manager dynamically allocates its resources according to policies obtained from a *Policy Repository*. Whenever an Application Manager cannot implement its policy (e.g. through lack of resources), it asks a *Resource Arbiter* (RA) for additional resources. The Resource Arbiter may remove resources from one Application Manager to give them to the other one. SLAs are contracts between service providers and consumers; the policies define how the application has to adapt itself to changing demand and resources availability.

We sketch out how this example can be instantiated within the framework of Section 3. We will restrict resources to servers only. The components here are the two Application Managers (AM1, AM2), the Resource Arbiter (RA) and two Servers (S1, S2). The metadata here are S1 and S2 transaction response times and S1 and S2 CPU availability. Metadata are permanently monitored and updated, the requests are directly submitted by the corresponding AM to the RA according to the underlying SOA. The "Action" policies defined in [10] are examples of SA Adaptation policies in our framework. Both AM1 and AM2 have the same policies, the RA always give priority to AM1.

– AM-Policy1: "increase CPU by 5% if response for transaction is above 100 ms"
– AM-Policy2: "if transaction time above 100 ms and CPU usage is more than 98% send Request to Resource Arbiter for more CPU"
– RA-Policy: "if Request for more CPU, Grant and give priority to AM1".

This is an example of a top-down system with two autonomous components taking local decisions on the basis of their individual metadata, and a third one taking decisions regarding the other two. All components perform some reaction triggered by an event (lack of CPU). As discussed in [10] this design solution is not necessarily scalable, and other policies, such as goal or utility-function policies, may be used instead. These can be accommodated just as well within the generic framework described in this paper (see the discussion of Guiding Policies above). Bounding policies could here be added when the RA cannot find a good solution to allocate the resources, e.g. to stop AM1 taking all the resources.

Let us now consider this system from an engineering perspective. The performance of the individual applications is to provide response time below 100 ms. A system level requirement may be to provide two services that both have a response time below 100 ms. A design-time validation of this latter property on the model and policies described above reveals that there are cases when the system is likely to violate the requirement: in particular, when the demand on AM1 is too high, then AM2 may fail to deliver its service to the level required by the specified SLA. The problem stems from the fact that resources are finite, and the system may not adapt indefinitely. This could then lead to a revision of the SLAs.

5 Application to a Self-Organising System

We now consider the application of our framework to a self-organising system. We will consider a simple stigmergy-based system, where components communicate indirectly with each other by modifying their local environment, by inserting virtual tags at their local positions. We take the example of traffic light controllers inserting traffic flow information observed at the 8 branches of a road intersection (4 paths and 2 directions for each path). Cars arriving at road intersections sense the traffic flow and may decide to change their route.

The components here are: the mobile entities (cars) and the traffic light controllers. Cars are programmed to go from one source location to a destination location and are equipped with an initial planned route. The metadata here are given by the traffic flow (8 values) at each road intersection. These values are permanently updated by the traffic light controllers sitting at each road intersection. Policies attached to the components are:

– Car Policy: “if traffic flow down the path is above threshold, change direction and recalculate route”
– TrafficLight Policy: “modify traffic lights duration according to traffic flow observed”.

This is an example of a bottom-up system with several independent components communicating indirectly and taking decisions on the basis of locally available metadata. Some external pressure or control could be inserted in this system by directly modifying the metadata at different locations, such as inserting a diversion by blocking a path. It is also interesting to note that the use of guiding policies is not necessary in a pure bottom-up system. Indeed, the cars or traffic lights policies could as well be coded into the components.

As for the previous example, let’s now consider this system from an engineering point of view. The performance of the overall system is to maximise traffic throughput. The performance of an individual car is to minimise the travel time given its origin and destination point, while those of a traffic light controller is to maximise throughput. These are the expected self-* properties of the overall system and of the individual components identified at design-time.

The interesting point to note here is how the global property (maximising traffic throughput) is broken down and implemented into local rules (cars and traffic-light policies), and how this activity initiated at design-time is subsequently refined during the development and implementation in order to derive the actual policies used at run-time. The actual proof that the global properties are correctly broken down into local rules in a decentralised schema is still a subject of research, and we do not consider this here. research.

6 Proofs of Concept and Initial Experiments

We are conducting several proofs of concept studies on the enabling technologies needed to realise the framework presented above.

Study 1: Acquisition and Use of Metadata. A Mediator system [12], aiming at improving dependability of Web Services (WSs), is being developed in Newcastle University using the framework presented above. This introduces an intermediate overlay network of the specialised SubMediator WSs. Each of these SubMediators records in a local database the dependability metadata derived from continuous observations of the target application WSs in the context of an e-Science application. It acts as a client to these WSs and monitors them by tracking availability, functionality, performance, faults and exceptions.

These metadata allow each SubMediator to make dynamic decisions improving the dependability of the execution of the client requests by choosing the target WSs which fits best to the required level of quality defined by the client. Some examples of such requirements are the dynamic choice of the most reliable WS or of the WS with a quickest reply. If there are no individual WSs satisfying these requirements the SubMediator dynamically chooses a suitable fault tolerance mechanism which can satisfy them, from a repertoire of mechanisms that the Mediator system supports. This decision is made using the metadata from the local database. The fault tolerance mechanisms and the required level of dependability are defined by the clients using a policy language. Supported fault tolerance mechanisms include: retry of the request, multi-routing over the Internet, various ways of using diverse target WSs (e.g. sequential requests in the manner of recovery blocks, or concurrent requests with or without majority voting).

The distributed architecture of this system makes it possible to collect, publish and make decisions using runtime dependability metadata specifically representing the end-user's perspective of the network and component WSs behaviour, therefore paving the way to achieve dependability-explicit operation of WSs.

The first version of the system representing a monitoring tool for collecting metadata about a selected set of target WSs [13] is available for downloads[5].

Study 2: Run-time infrastructure. A restricted version of the run-time environment in Section 3 has been implemented, supporting functional self-description of services, and a limited form of non-functional QoS description. The underlying infrastructure is a service-oriented middleware allowing registration of formal service descriptions and service requests, description matching and seamless binding of components (self-assembly). Interoperability is supported without a specific API and is solely based on service descriptions.

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 pre-defined keywords [14]. 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. 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 [15]. This prototype supports specifications written either in Prolog, or as regular expressions.

[5] http://www.students.ncl.ac.uk/yuhui.chen/#Download

Study 3: Dynamically Resilient Systems. In our most recent study, a first instantiation of our framework has been proposed in [16] supporting self-reconfiguration and dynamically resilient systems. Resilience mechanisms at design-time are translated into resilience policies enforced at run-time on the basis of monitored resilience metadata.

Conclusions. Study 1 has demonstrated the possibility of building a dynamically self-configurable architecture 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 (without any involvement of the e-Scientist). Study 1 has also shown the interest of the approach of separately describing execution flows and policies to dynamically adapt the system to high-level user needs, in this case the needs of the scientist using the system. Study 2 has shown that it is possible to dynamically add in the system and seamlessly use additional features; to dynamically replace updated entities without the calling entities noticing the replacement (even during a call [14]).

7 Discussion

Predictability. The "Enforcement of Policies" services described above are intended to support matching and replacement using automated reasoning on specifications, but are not meant to work as an artificial intelligence tool. Therefore, *predictability* is primarily obtained by the enforcement of explicitly defined policies. However, in a large-scale environment with many components from various suppliers, it may be difficult to ensure conformance. There may be a need for a kind of "meta-policy" spanning the whole system or application and to which individual policies would need to adhere. An alternative would be to consider hierarchical policy schemas. In addition to the risk of conflicting policies, it may also happen that a chosen emergent configuration is suboptimal. The Bounding policies mentioned above are intended to prevent the system going beyond its limit; they should have precedence over other policies whenever the boundaries of systems behaviour are reached. However, this still remains an issue of further research and study.

On metadata. In the framework outlined in Figure 2, the use of shared metadata provides direct support for self-organising systems (which could work without policies), while the use of policies supports more naturally self-adaptive systems. In the case of SO systems, having both shared metadata and policies, in particular the use of Coordination and Bounding policies, allow the designer to foresee and enforce at run-time some form of overall control on the system. For SA systems, metadata shared among components provides support for inserting self-organising and decentralised behaviour into these systems that generally show central and hierarchic features.

The use of metadata can go far beyond what has been described so far, it could also serve to monitor how well a certain decision has an impact on a self-* property (e.g. self-optimisation), such as quantifying the degree of self-adaptation of a system.

Control and feedback loop. Metadata is either directly modified by components or indirectly updated through monitoring. Metadata, together with the policies, cause the reasoning tool to determine whether or not an action must be taken. The Enforcement of Policies services act on both components and metadata, impacting components both directly and indirectly.

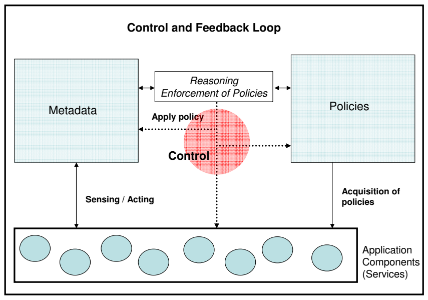

Fig. 3. Control and Feedback Loop

Control is inserted in three different ways (Figure 3). First, the Reasoning and Enforcement of Policies services directly act on components by dynamically reconfiguring them, allocating more components, removing faulty ones, etc. Second, an indirect action is performed by modifying the metadata used by the components to sense their environment. This is a technique used for (externally) controlling self-organising system working with stigmergy. Third, an additional way of controlling the system consists of modifying the policies used by the components for driving their behaviour on-the-fly. As described in Section 2, policies are decoupled from the components even if they are locally attached to them: changing the policies will immediately affect the corresponding component. Even though the control shown on Figure 3 is internal to the system, external control is applied in the same way, by acting directly on the components, metadata or policies.

Dynamic policies. Policies are considered to be as much as possible decoupled from the components themselves. This has the advantage, as shown in Figure 2, to provide the possibility to the components to dynamically acquire policies at run-time. This may be useful when devices change context (e.g. a PDA moving around), or when global policies change (e.g. rights are denied to some user).

8 Related Work

The "Observer-Controller" is a generic paradigm architecture [8] attaching to individual components, or groups of components, an "observer" component responsible for monitoring events and states, and a "controller" component responsible to take actions whenever the observer part results let it consider appropriate. The implementation of the "observer-controller" structure is dependent on the specific application. The approach proposed in this paper can be viewed as an instantiation of this paradigm, since the enforcement of policies at run-time act as a controller, while the acquisition and monitoring of metadata act as an observer.

In the field of autonomic computing, a uniform representation and composition of autonomic elements encompassing the use of a service-oriented architecture supporting the interactions of these autonomic elements, preliminary design patterns and policies is proposed in [3]. The notions of registry and brokers [3] are similar to those described in our framework as the services handling component descriptions (matching requests, retrieving services, creating appropriate workflows); the monitoring aspect of the sentinels [3] relates to the monitoring of metadata.

Accord [17] is a programming framework for autonomic applications. It supports the notion of rules controlling both the component and interaction behaviour, and allows dynamic addition, deletion or replacement of components as well dynamic changing of interactions. Our Study 2 (Section 6), from which the run-time aspect of our framework is derived, does not use the notion of rules, but allows dynamic replacement, adjunction, or removal of services on the basis of their specification. Changes are enforced by dynamically reconfiguring services and by modifying metadata.

Self-Managed Cells (SMC) [18] consist of a set of heterogeneous hardware and software elements, a set of 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 some metadata is collected about their behaviour.

Design-time concerns have given rise in recent years to diverse proposals defining design patterns for coordination of SO systems [9], design patterns for self-managing systems [3], and bio-inspired design patterns for distributed systems [19]. Our paper does not propose any design pattern, however our proposal encapsulates the use of design patterns.

A proposal similar to the one provided in this paper is discussed in [20]. It is intended specifically for autonomic systems, and shares the same 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. To this extent, an interface for services is proposed that allows monitoring of and interaction with the services.

9 Conclusion

We have discussed a generic framework supporting the development of trustworthy self-adaptive and self-organising systems, derived from a consideration of engineering requirements. The framework encompasses support for decision-making at design-time and at run-time. We have briefly described the key elements of architectures required to implement this framework and initial proof-of-concept studies on the component technologies. Leveraging the studies, we plan to build a "seed" run-time infrastructure based on our framework. We will emphasise three areas: resilience, self-reconfiguration, and control of SA or SO systems (using e-Science applications); establishment of self-*

properties and their verification; and adaptation of applications involving digital rights management.

Acknowledgements

We are grateful to Yuhui Chen for his work on the Mediator system. This work is partially supported by the EC Network of Excellence on Resilience for Survivability in IST (ReSIST: www.resist-noe.org).

References

1. Babaoglu, O., Shrobe, H., eds.: First IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007) 9-11 July, Boston, MA, USA, IEEE (2007)
2. Kephart, J.: Research challenges of autonomic computing. In: 27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA, ACM (2005) 15–22
3. White, S., Hanson, J., Whalley, I., Chess, D., Kephart, J.: An architectural approach to autonomic computing. In Kephart, J., Parashar, M., eds.: International Conference on Autonomic Computing (ICAC'04), IEEE Computer Society (2004) 2–9
4. Anderson, T., Randell, B., Romanovsky, A.: Wrapping the future. In: IFIP 18th World Computer Congress 2004, Toulouse, France. (2004)
5. van der Meulen, M., Riddle, S., Strigini, L., Jefferson, N.: Research challenges of autonomic computing. In: COTS-Based Software Systems: 4th International Conference, ICCBSS 2005 2005, Bilbao, Spain, Springer-Verlag (2005)
6. Singh, M., Huhns, M.N.: Service-Oriented Computing - Semantics, Processes, Agents. John Wiley & Sons, Ltd, UK (2005)
7. Rana, O.F., Kephart, J.O.: Building effective multivendor autonomic computing systems. IEEE Distributed Systems Online **7** (2006) art. no. 0609-o9003.
8. Müller-Schloer, C.: 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, ACM (2004) 2–5
9. De Wolf, T., Holvoet, T.: Design Patterns for Decentralised Coordination in Self-Organising Emergent Systems. In: Engineering Self-Organising Systems. Volume 4335 of LNAI., Springer-Verlag (2007) 28–49
10. Kephart, J.O., Walsh, W.E.: An artificial intelligence perspective on autonomic computing policies. In: IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004). (2004) 3–12
11. Leff, A., Rayfield, J.T., Dias, D.M.: Service-level agreements and commercial grids. IEEE Internet Computing **7** (2003) 44–50
12. Chen, Y., Romanovsky, A.: A mediator system for improving dependability of web services. In: The International Conference on Dependable Systems and Networks (DSN-2006). (2006)
13. Li, P., Chen, Y., Romanovsky, A.: Measuring the dependability of web services for use in e-science experiments. In: International Service Availability Symposium (ISAS 2006). (2006) 193–205
14. Oriol, M., Di Marzo Serugendo, G.: A disconnected service architecture for unanticipated run-time evolution of code. IEE Proceedings-Software, Special Issue on Unanticipated Software Evolution (2004)

15. Di Marzo Serugendo, G., Deriaz, M.: Specification-carrying code for self-managed systems. In Martin-Flatin, J.P., Sventek, J., Geihs, K., eds.: IEEE International Workshop on Self-Managed Systems and Services. (2005)
16. Di Marzo Serugendo, G., Fitzgerald, J., Romanovsky, A., Guelfi, N.: A Metadata-Based Architectural Model for Dynamically Resilient Systems. In: ACM Symposium on Applied Computing (SAC'07), to appear. (2007)
17. Liu, H., Parashar, M., Hariri, S.: A component-based programming model for autonomic applications. In Kephart, J., Parashar, M., eds.: International Conference on Autonomic Computing (ICAC'04), IEEE Computer Society (2004) 10–17
18. Dulay, N., Lupu, E., Sloman, M., Sventek, J., Badr, N., Heeps, S.: 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., Springer (2005) 1–6
19. Babaoglu et al., O.: Design patterns from biology for distributed computing. ACM Transactions on Autonomous and Adaptive Systems **1** (2006)
20. Liu, L., Schmeck, H.: A roadmap towards autonomic service-oriented architectures. In: International Service Availability Symposium (ISAS 2006). (2006) 193–205