MetaSelf - A Framework for Designing and Controlling Self-Adaptive and Self-Organising Systems

Giovanna Di Marzo Serugendo[1], John Fitzgerald[2], Alexander Romanovsky[2], and 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 proposes a unifying framework for the engineering of dependable self-adaptive (SA) and self-organising (SO) systems. We first identify requirements for designing and building such SA and SO 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 two different systems: (1) a dynamically resilient Web service system; (2) design of an industrial assembly system with both SA and SO capabilities.

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 on a large scale or professional level, it is important to address their engineering design, development and control. One of the most significant aspects of this is the challenge of developing *trustworthy* SA and SO systems. A trustworthy system must be dependable, and there must also be evidence of dependability so that reliance can justifiably be placed on the developed system. Thus, during development, deployment and subsequent evolution, we must provide assurance that the emergent behaviours will respect key properties, for example relating to safety, security or performance, and that the human administrator retains control despite the system's self-* capabilities. Such assurance can come through design verification of designs as a means of avoiding defects, the design of fault tolerance into the system and the testing of system behaviours prior to deployment. The contribution of this paper is to propose a framework for the design of trustworthy SA and SO systems in which verification and fault tolerance techniques can fit.

We use the following working definitions of SA and SO systems: "Self-adaptive systems work in a top-down manner. They evaluate their own *global* behaviour and 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." [9].

At first sight, SA and SO systems appear very different: SA systems tend to be hierarchic and driven top-down; SO systems tend to be decentralised and driven bottom-up. However, it would appear that many applications demand systems that include both SA and SO characteristics. The need to allow more "freedom" to self-adapting systems, by allowing some decentralisation and self-organisation to the components, has already been advocated [12]. Conversely, self-organising systems with purely decentralised control should provide assurance of their behaviour to potential users prior to deployment and should allow control to be imposed with confidence 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. Socio-technical systems encompass, among others, ambient 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 at a unified view of the engineering of trustworthy SA and SO systems. We first expose requirements that should be satisfied by such system architectures in order to make them amenable to the kinds of rigorous analysis and fault tolerance design required for dependable systems (Section 2). We then propose elements of a generic engineering framework supporting designers and developers of SA and SO systems, relating them to our identified requirements (Section 3). Sections 4 and 5 show how our framework applies to two applications from different domains. We discuss these in Section 6 and describe related work in Section 7.

2 Engineering Requirements for Trustworthy SA/SO Systems

The systems that we consider here are distributed systems on a potentially large scale consisting of components that may be physical devices, services, or people communicating over networks that may be fixed or ad-hoc. They exhibit emergent behaviour and have the capacity to respond autonomously to events within the system and in the environment. In considering trustworthy SA/SO systems, we focus on threats: events that could compromise the delivery of the specified system level behaviour. These threats include component or network failures, or deviation of the environment from anticipated behaviour. Building a dependable SA/SO system entails gaining confidence that specified properties will be respected, even under certain assumptions about the range of threats that should be tolerated. The dynamic character of SA/SO systems brings the potential to use dynamic reconfiguration as a response to threats. However, it also challenges traditional methods of engineering dependable systems, which typically rely on extensive static analysis during development to achieve a level predictability. Thus, an engineering view of trustworthy SA and SO systems must achieve a balance, gaining predictable, yet dynamic, resilience. A framework for engineering trustworthy SA and SO systems must take account of several requirements arising from the engineering need to validate system models during design, as well as the need to deliver a dependable level of service. Below we list these key requirements; each requirement is numbered **Rn** for reference.

Autonomous individual components. Robustness and self-* behaviour arise from the numerous (low-level) individual components providing the core functionality of the system, and from their (local) interactions. In SO systems, such components can, for example, be ant-like entities, agents, peers or motor vehicles. In SA systems, autonomic elements, autonomic managers, and any element of the supporting infrastructure (e.g. the service registry or sentinel monitoring mentioned in [20]) are all autonomous components. In trustworthy SA/SO systems, the localisation of threat detection and response is important, both in order to limit harm and in order to provide modularity in design verification.

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

Interoperability. The overall (global) behaviour of an SA or an SO system arises from the interactions of the individual components. In real-world scenarios involving open and dynamic systems, the individual components are heterogeneous both in nature and in design: different vendors will be providing autonomic elements, and differences in the ownership of participant elements (e.g. in P2P, MANET and ambient intelligence scenarios) may make it difficult to engineer centralised control. Interoperability lies at a semantic level and encompasses understanding of functional and non-functional (Quality of Service/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 [1, 19].

R2: Components should be decoupled from descriptions of their capabilities, QoS, Constraints, execution flows, interaction modes, 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, 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 (e.g. artificial pheromones, configurations, neighbours), and take decisions accordingly to change 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 a 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 of the system). We will refer to these rules or policies as *guiding behaviour*.

In addition to guiding the system towards optimal functioning, it is 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 grabbing all the resources; a trust-based system may have an immutable threshold below which no transaction request 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 and component properties.

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

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

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

3 A Framework for Engineering SA/SO Systems

In this section we propose a framework for engineering dependable SA/SO systems that aims to take account of the requirements listed above. We first identify technologies that together may be capable of meeting these requirements. We then describe a framework in which these technologies contribute to design-time analysis of system dependability characteristics coupled with bounded run-time freedom to adapt and reconfigure in response to threats.

3.1 Relevant Technologies

Service-Oriented Architecture. Service-oriented architectures are becoming widely accepted as architectural solutions for systems involving autonomous components, dynamicity and heterogeneity [18]. 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)). Service-oriented Architectures address **R1** by supporting loose coupling and hence the handling of heterogeneous and autonomous components in a homogeneous, modular way.

Self-Describing Components/Services. As pointed out in [16], interoperability is fundamental when different service providers are involved in the same system. Decoupling components (software programs) from descriptions of their capability, QoS, requirements and constraints is thus a solution for solving interoperability and deriving runtime solutions in case of unexpected condition or changing policies: such as replacing a failing autonomic manager with one that has equivalent characteristics. The approach addresses the need for for interoperability driven by heterogeneous design that underpins requirement **R2**.

Acquired, Updated and Monitored Metadata. Sensing and acting is a fundamental activity in 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 are data *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). Metadata are often used during a design process. For modern SA/SO systems built on architectures supporting self-describing components or services, it can be acquired, maintained and exploited within the running system.

The use of metadata addresses requirements **R2**, **R4**, **R5**. Published metadata supports interoperability; updated metadata values support self-awareness (sensing, reasoning and acting on the basis of metadata values). In particular, metadata (and observed changes in metadata) may serve to trigger the identification and response to a threat. Explicit handling of such dependability-related metadata is important to the verification of resilience for trustworthy SA/SO systems.

SA and SO Architectural/Design Patterns. Architectural patterns describe high-level coordination techniques such as the Model Reference Control Architecture (MRAC) from control theory [2], the notion of observer/controller [15], the notion of autonomic manager coupled with any autonomic element [7] for self-adaptation; or coordination through stigmergy or field-based structures [8], trust or gossip for self-organisation. The use of such patterns addresses requirements **R3**, **R10** for the 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. For example, a switching coordination pattern might describe the replacement of a component with a functionally equivalent one if the performance of the first deteriorates, a pattern describing an adaptive fault tolerance structure in which replicas are dynamically configured, or what action to take next on the basis of locally sensed information. These patterns may be implemented in specific applications through executable policies (see below). They help to address requirement

R11 by supporting the design-time specification of acting/adapting policies in response to monitored metadata.

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)

The use of executable policies addresses requirements **R6**, **R7** and **R8**.

In our framework, we draw a distinction between policies and SA/SO mechanisms, although the two are closely related. Policies, which are application-specific, implement specific SA/SO mechanisms in a coordinated fashion. The mechanisms are design patterns specifically intended to promote resilience to threats.

Formal Languages and Specifications. Formal techniques have a long history of application in the development of demonstrably dependable computing systems, particularly in critical applications. Developments in formal modelling and analysis technology are making it cheaper, through automated or semi-automated proof and model checking, to validate designs and even implementations to a high degree of assurance.

In order to support predictable dynamic reconfiguration, metadata, component descriptions, and policies need to be available at run-time. The concept of "lingua franca" 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 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.

3.2 MetaSelf - An Engineering Framework

We now present the design-time and run-time aspects of MetaSelf, a framework for the engineering of dependable SA/SO systems.

Figure 1 represents key design-time activities. During the *Analysis* phase, desired properties of the overall system are identified. In particular functional aspects, self-* requirements and properties (e.g. achieving self-protection or self-healing), and QoS.

During the *Design* phase, driven by the identified properties, the designer first selects the SA/SO architectural patterns (e.g. autonomic manager) and identifies the adaptation mechanisms that the system will adhere to (e.g. stigmergy or field-based). Second, it refines the chosen patterns for the specific application, architecture and policies, identifies and designs the individual components (agents, services, etc), selects and describes the corresponding metadata. These activities relate to requirements **R9**, **R10** and **R11** through the explicit identification of system-level and component-level properties, and corresponding patterns, and policies for enforcing them.

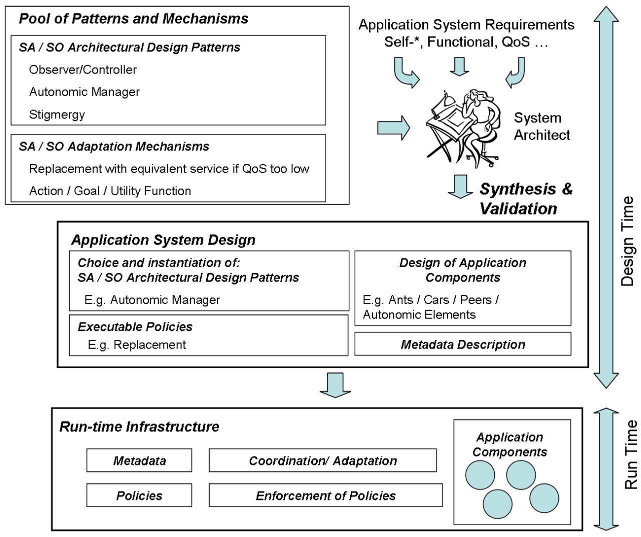
Pool of Patterns and Mechanisms
SA / SO Architectural Design Patterns
Observer/Controller
Autonomic Manager
Stigmergy
SA / SO Adaptation Mechanisms
Replacement with equivalent service if QoS too low
Action / Goal / Utility Function
Application System Requirements
Self-*, Functional, QoS …
System Architect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

Fig. 1. Design-time vs Run-time

During the *Implementation* phase, the run-time infrastructure is developed, together with the individual components, the executable policies and metadata sensing and monitoring capabilities.

The run-time environment is itself based on 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 (for monitoring activities) or by the components themselves (for 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**).

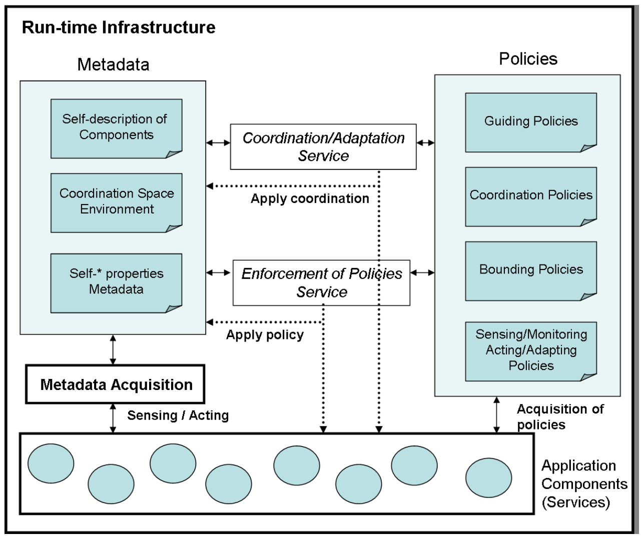
Run-time Infrastructure
Metadata
Policies
Self-description of Components
Coordination Space Environment
Self-* properties Metadata
Coordination/Adaptation Service
Apply coordination
Enforcement of Policies Service
Apply policy
Guiding Policies
Coordination Policies
Bounding Policies
Sensing/Monitoring 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.

Control and feedback loop. Metadata are 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.

Control is provided in three ways (Fig. 3). First, the "Reasoning and Enforcement of Policies" services act directly on components by reconfiguring them dynamically, 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 Fig. 3 is internal to the system, external control is applied in the same way, by acting directly on the components, metadata or policies.

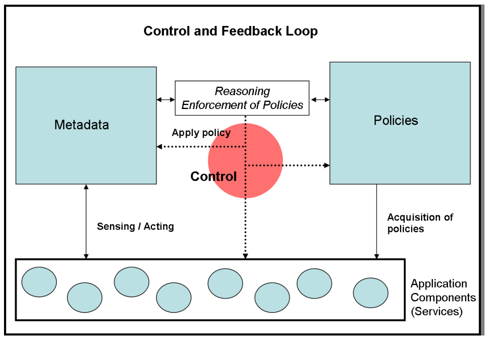

Fig. 3. Control and Feedback Loop

Dynamic policies. Policies are considered to be as much as possible decoupled from the components themselves. This has the advantage, as shown in Fig. 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).

4 Application 1: Dependable Web Services

In this section we briefly show how the generic framework proposed has been applied in developing a WS-Mediator architecture - a novel approach to improving dependability of web service composition [6, 5]. WS-Mediator relies on an off-the-shelf mediator solution implementing run-time dependability monitoring and assessment, resilience-explicit computing and fault tolerance mechanisms used together to achieve dependable dynamic web service integration. A Java WS-Mediator framework based on has been developed and applied in a number of experiments conducted in the context of the bioinformatics and virtual organisation domains.

The WS-Mediator architecture is developed for the web service domain, in which components, i.e. web services, meet our requirements for decoupling from their descriptions and from the underlying coordination infrastructure. Self-awareness is architected as an overlay network of dedicated components monitoring the application services and collecting dynamic information about their behaviour as three metadata (m, r, f) described below. The WS-Mediator system consists of a set of interconnected functionally-identical Sub-Mediators distributed over the Internet (Fig. 4) to monitor the dependability of web services, and to provide accurate dependability metadata, presenting web service dependability characteristics from the client's perspective. Each service is periodically invoked and if a valid result is returned, an availability score (m) increases. The round-trip response time of the invocation is recorded for calculating the average response time (r) of the service. If the service returns an invalid response, the value of m decreases, and the error message is logged in the database for statistic (f) recording types of failure. If the service fails to respond, or if an exception arises during the invocation, m also decreases, and the type of the exception is also logged for f.

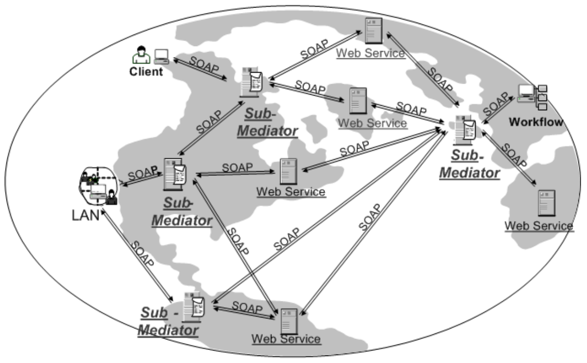

Fig. 4. The overlay architecture of the WS-Mediator system

The collected metadata are dynamically analysed and used to choose and enforce one of the fault tolerance policies. These policies are statically defined by the system architect to represent the most efficient strategies of tolerating a wide range of erroneous conditions in the application services and of the communication media. The policies make use of the various types of redundancy available in such global setting, including path diversity and application service diversity, and allow synchronous and asynchronous invocation of the application services. The most typical fault tolerance modes use backward recovery (retry and try of a similar service [17], masking diversity (invoking several similar services concurrently [3]) and path diversity (routing the service invocation through different routes). A basic reasoning engine is implemented as part of each WS-SubMediator to ensure the best possible choice for the current situation in the network and the application services as observed from the location of this WS-SubMediator.

The WS-Mediator architecture has been evaluated in two major sets of experiments. First we verified the validity of dependability monitoring using two web services provided to us by the GOLD project[4]. Second, in order to demonstrate the applicability and effectiveness of the WS-Mediator approach, we experimented with three web services implementing an algorithm which is commonly used in *in silico* experiments in bioin-

[4] http://www.neresc.ac.uk/projects/GOLD/projectdescription.html

formatics to search for gene and protein sequences that are similar to a given input query sequence (so-called BLAST services[5][6][7]). Our experiments with several real-world web services using the Java WS-Mediator framework have demonstrated the applicability and efficacy of the approach. The WS-Mediator architecture allows system integrators to improve dependability of the composed systems in an interoperable fashion without involving or restricting the service providers.

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

5 Application 2: Industrial Assembly Systems

Our generic framework has been applied to an industrial robotic example in order to enhance assembly systems with self-organisation and self-adaptation. The aim of this section is show how this framework can be applied to such a domain and to highlight the steps necessary to produce the design of a self-organising and self-adapting system. A full description of this example can be found in [11].

An *assembly system* is an industrial installation that receives parts and joins them in a coherent way to form a final product. It consists of a set of equipment items (modules) such as conveyors, pallets, simple robotic axes for translation and rotation as well as more sophisticated industrial robots, grippers, sensors of various types, etc. An *Evolvable Assembly System (EAS)* is an assembly system which can co-evolve together with the product and the assembly process. EASs are not yet self-organising or self-adapting - configuration of the agents, their positioning into an appropriate assembly system layout, and their monitoring and possible re-configuration in case of failure are still done manually.

Our goal is to design *self-organising evolvable assembly systems*, i.e. given a specified product order provided in input, the system's modules spontaneously select each other (preferred partners) and their position in the assembly system layout. They also program themselves (micro-instructions for robots' movements). The result of this self-organising process is a new or reconfigured assembly system that will assemble the ordered product. In the self-organising jargon, the appropriate assembly system *emerges* from the self-organisation process going on among the different modules of the assembly system. Any new product order (seen as a global goal) triggers the self-organising process, which eventually leads to a new appropriate system - there is no central entity, modules progressively aggregate to each other in order to fulfil the product order. This automated process does not stop at the layout formation. During production time, whenever a failure or weakness occurs in one or more of the current elements of the system, the process may lead to two different outcomes: the current modules adapt their

[5] http://www.ebi.ac.uk/
[6] http://hc.ims.u-tokyo.ac.jp/JSBi/journal/GIW00/GIW00P072/index.html
[7] http://pathport.vbi.vt.edu/main/home.php

behaviour (change speed, force, task distribution, etc) in order to cope with the current failure, eventually degrading performance but maintaining functionality; or may decide to trigger a re-configuration leading to a repaired system. The actual decision will depend on the situation at hand and on specific production constraints.

The concept is illustrated by considering the assembly of an adhesive tape dispenser (Fig. 5) consisting of two body parts (Parts 1 and 3) locked by a screw (Part 4) and the roll of tape (Part 2).

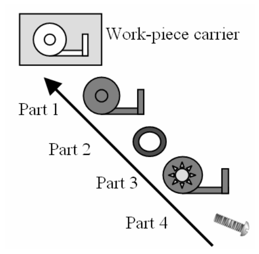

Fig. 5. Assembly of an Adhesive Tape Dispenser

The generic framework described above, when applied to Evolvable Assembly Systems, requires the following steps (at design-time): determination of components, self-* requirements, a self-organisation and/or self-adaptation mechanism and the choice of a coordination mechanism. On the basis of these choices, corresponding metadata and policies are derived.

5.1 Components

The components (and services) of the system are provided by the several agents. The *order agent* carries the generic assembly plan for tape dispensers. This is a series of instructions specific to the products being assembled, but independent from the specific industrial modules that will carry the assembly and from their positioning in the assembly layout. The *product agents* carry the layout-specific assembly instructions (micro-instructions) which the visited resource modules will be asked to execute. This is a translation of the generic assembly plan specific to the modules that participate to the layout and to their position. The *manufacturing resource agents* include the conveyor units, robots, grippers, feeders, etc. and the *part agents* are the Parts 1 to 4 and the work-piece carriers.

5.2 Self-* Requirements

Layout Formation. Any new assembly plan triggers a self-organisation process leading to a new configuration (layout). This encompasses self-selection of modules and their partners, and establishment of the layout-specific assembly instructions.

Task coordination at production time. This encompasses *task sequencing* in which modules coordinate their work according to the current status of the product being built; and *collision avoidance* whereby modules with overlapping workspace must maintain a minimum distance to each other while moving.

Self-Adaptation. *Self-healing*: During production, whenever a module failure is detected, by the module itself or one of its partners, either the modules can solve the problem by themselves (software restart, open and close a gripper, etc.) or they alert the user. ”Module failure” can mean many types of perturbations, for instance a blocked gripper, a software problem, or a lost part. *Self-optimization*: During production, the speed of processing the product parts is adjusted to the queuing level.

5.3 Mechanisms

Self-organisation mechanism. The mechanism chosen for letting the different modules and product parts re-organise whenever a new assembly plan arrives, or when a change in one of the modules is requested, is inspired by the tiles self-assembly model of crystal growth. In this model, tiles progressively attach to each other following matching rules and current configuration of the structure to build [21].

Self-adaptation mechanism. Self-adaptation, e.g. resilience to failures, to the effects of fatigue and collision avoidance, is performed by modules monitoring their own behaviour or their neighbor's behaviour.

Coordination mechanism. During production, the coordination of tasks for each individual product item is done through indirect communication by storing the current advancement of the assembly in a shared place - a RFID attached to the product item.

5.4 Policies and Metadata

We have defined policies and metadata addressing the self-* requirements above, including layout formation, coordination, and self-adaptation to weaknesses and failures, taking into account the chosen mechanisms. We have also defined a full list of policies and metadata in [11]. We present here a sample of them, specifically self-healing policies and metadata in case of module fatigue (see also the resilience scenario below).

Modules monitor their own precision, and possibly also their neighbour's precision, in order to detect the effects of fatigue or other kinds of disturbance and to take corresponding countermeasures. Examples of industrial modules (components) policies are: If the target position after a movement has not been reached correctly, take corrective measure (advance more / less, ask for maintenance); In case of malfunctioning, request a replacement from a coalition partner; If no replacement found, ask for a re-configuration of the layout; If queuing level is too high and speed is at maximum, ask for a re-configuration of the layout.

An example of a system-level policy (bounding policy) is: If one (or more) failure occur more than a certain number of times in a certain period of time, then alert user and trigger new configuration.

Self-* Properties Metadata. For each component: Maximal / optimal speed of operation; Own precision - movements on every axis; Precision of partners (neighbors); Queuing level of product agents (in input to that resource agent); Quality of assembled product.

5.5 Scenarios

For this example, we considered the following scenarios: arrival of a *new product order*; *resilience* during production (failure of a robot); and small *change in the product design*.

Establishment of a Circular layout (new product order). The layout formation is progressively reached through a series of service requests including needed 3D movements (x/y/z dimensions micro-instructions). The first request is the one from the order agent requesting fulfilment of the generic assembly plan. Requests are progressively fulfilled by resource agents. Self-description metadata are matched against the requests taking into account skills, constraints and any system/bounding policy, and currently achieved configuration. This corresponds to the matching rules of the tile self-assembly model.

Resilience. A robot experiences problems in reaching its target positions and asks for maintenance. As a temporary solution, another robot is asked to take over. After taking over the latter robot experiences a high level of queuing. This leads that robot to ask a re-configuration (P4).

From screw to snap-fit (small design change). The screw is not needed any more because a snap-fit is now integrated with Part 3. A minimum change policy in case of reconfiguration drives the selection of an additional gripper able to apply the appropriate force on Part3.

5.6 Implementation

Implementation of this example is under way - matching of skills using a directory facilitator together with dynamic composition of complex skills from simpler ones is already performed. Layout formation and self-adaptation are being developed.

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

6 Discussion

Predictability. The “Enforcement of Policies” services described in Section 3 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 system's behaviour are reached. However, this still remains an issue of further research and study.

On metadata. In the framework outlined in Fig. 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.

Centralised/Decentralised Run-time infrastructure. As mentioned previously, the run-time infrastructure is considered itself as a set of services possibly running on different machines and with local autonomy, neither the supporting infrastructure nor system need to be centralised.

Towards a Development Method. Leveraging from Application 2 (Section 5), a full development method is envisaged for designing self-organising and self-adaptive aspects of a given system. First, the components (services, agents, etc) of the system have to be identified. Second, the self-* requirements are established where and when self-organisation and self-adaption is needed/desired. Third, the appropriate mechanisms corresponding to the self-* requirements are determined. They can be inspired by nature, by a specific model, or follow a known self-organising/self-adaptive pattern. Finally, sets of policies and metadata for the different self-* requirements are identified (system-level and component-level).

7 Related Work

The "Observer-Controller" is a generic paradigm architecture [15] 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 acts 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 [20]. The notions of registry and brokers [20] 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 [20] relates to the monitoring of metadata.

Accord [13] 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.

Self-Managed Cells (SMC) [10] 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 that some metadata are collected about their behaviour.

Design-time concerns have given rise in recent years to diverse proposals defining design patterns for coordination of SO systems [8], design patterns for self-managing systems [20], and bio-inspired design patterns for distributed systems [4]. Our paper does not propose any specific design pattern, however our proposed framework does encompass the use of design patterns.

A proposal similar to the one provided in this paper is discussed in [14]. 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.

8 Conclusion

We have discussed Metaself, 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 its application on two examples. Leveraging the studies, we plan to build a “seed” run-time infrastructure based on our framework. We will emphasise the following areas: resilience, self-reconfiguration, and control of SA or SO systems (using e-Science applications); establishment of self-* properties and their verification.

Acknowledgements

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

References

1. T. Anderson, B. Randell, and A. Romanovsky. Wrapping the future. In *IFIP 18th World Computer Congress 2004, Toulouse, France*, 2004.

2. K.J. Astrom and B. Wittenmark. *Adaptive Control*. Addison-Wesley, second edition edition, 1995.
3. A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Press*, 11:1491–1501, 1985.
4. O. Babaoglu et al. Design patterns from biology for distributed computing. *ACM Transactions on Autonomous and Adaptive Systems*, 1(1), 2006.
5. Y. Chen. *WS-Mediator for Improving Dependability of Service Composition*. PhD thesis, Newcastle University (UK), 2008.
6. Y. Chen and A. Romanovsky. Improving the dependability of web services integration. *IT Professional*, 10(3):29–35, 2008.
7. IBM Corporation. An architectural blueprint for autonomic computing. White Paper 4th Ed., June 2006.
8. T. De Wolf and T. Holvoet. Design Patterns for Decentralised Coordination in Self-Organising Emergent Systems. In *Engineering Self-Organising Systems*, volume 4335 of *LNAI*, pages 28–49. Springer-Verlag, 2007.
9. G. Di Marzo Serugendo, J.-P. Martin-Flatin, M. Jelasity, and F. Zambonelli, editors. *First IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007) 9-11 July, Boston, MA, USA*. IEEE, 2007.
10. N. Dulay, E. Lupu, M. Sloman, J. Sventek, N. Badr, and S. Heeps. Self-managed cells for ubiquitous systems. In *Proceedings of the Third International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security, MMM-ACNS 2005, St. Petersburg, Russia, September 25-27, 2005, Proceedings*, volume 3685 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 2005.
11. R. Frei, G. Di Marzo Serugendo, and J. Barata. Designing self-organization for evolvable assembly system. In *IEEE International Conference on Self-Adaptive and Self-Organising Systems (SASO'08)*. IEEE Computer Society, 2008.
12. J. Kephart. Research challenges of autonomic computing. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 15–22. ACM, 2005.
13. H. 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.
14. L. Liu and H. Schmeck. A roadmap towards autonomic service-oriented architectures. In *International Service Availability Symposium (ISAS 2006)*, pages 193–205, 2006.
15. C. Müller-Schloer. Organic computing: on the feasibility of controlled emergence. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2004*, pages 2–5. ACM, 2004.
16. O. F. Rana and J. O. Kephart. Building effective multivendor autonomic computing systems. *IEEE Distributed Systems Online*, 7(9), 2006. art. no. 0609-o9003.
17. B. Randell and J. Xu. The evolution of the recovery block concept. In *Software Fault Tolerance*, pages 1–22. J. Wiley. New York, 1994.
18. M.P. Singh and M. N. Huhns. *Service-Oriented Computing - Semantics, Processes, Agents*. John Wiley & Sons, Ltd, UK, 2005.
19. M. van der Meulen, S. Riddle, L. Strigini, and N. Jefferson. Protective wrapping of off-the-shelf components. In *COTS-Based Software Systems: 4th International Conference, ICCBSS 2005, Bilbao, Spain*, volume 3412 of *LNCS*, pages 168–177. Springer-Verlag, 2005.
20. S. White, J. Hanson, I. Whalley, D. Chess, and J. Kephart. An architectural approach to autonomic computing. In J. Kephart and M. Parashar, editors, *International Conference on Autonomic Computing (ICAC'04)*, pages 2–9. IEEE Computer Society, 2004.
21. E. Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, California Institute of Technology, 1998.