A Metadata-Based Architectural Model for Dynamically Resilient Systems

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

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

Nicolas Guelfi
Laboratory for Advanced Software Systems
University of Luxembourg
Luxembourg
Nicolas.Guelfi@uni.lu

ABSTRACT

Designing open and distributed systems that can *dynamically* adapt in a *predictable* way to unexpected events is a challenging issue still not solved. Achieving this objective is a very complex task since it implies reasoning at run-time, explicitly and in a combined way, on a system's functional and non-functional characteristics. This paper proposes a service-oriented architectural model allowing the dynamic enforcement of formally expressed metadata-based resilience policies. It also describes preliminary dynamic resilience experiments acting as proof of concept.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*data abstraction, patterns*

General Terms

Verification, Experimentation

Keywords

Metadata, dynamic reconfiguration

1. INTRODUCTION

Large network-enabled computing systems are becoming integral to society, security and economy [18] and are likely to become more pervasive as technology is embedded in a wider range of products in the environment [9]. The openness and flexibility of such systems pose challenges to the maintenance of predictable levels of dependability, but also provide opportunities for dynamic resilience in response to threats.

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

Current techniques and tools used to develop dependable systems largely rely on design-time analysis. For example, the decision to use a particular fault tolerance technique such as n-version programming, and its particular configuration (the choice of the particular version components and their number n) is governed by the characteristics of the components that are available to the developer, their development costs, reliabilities and other factors [2]. The analysis underpinning this design decision is done at design time because the set of possible components and configurations is relatively fixed. When a component evolves, or changes, or requirements change, part of the design process must be repeated to verify characteristics of the adapted solution. However, next generation systems open the possibility of making dependability-maintaining adaptations at run-time. Consider an example scenario:

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

One classical approach [3, 15] to the provision of dependability here would involve a design-time decision to use a fault tolerance strategy such as deploying a fixed number of diverse but compatible GPS sources instead of relying on a single one. The selection of the component services would be based on information about the available services such as known availability levels and failure rates, and formal descriptions of the functionality provided by the components.

Such classical approaches, when applied in a more open and dynamic environment, restrict dynamic flexibility. In the scenario, for example, the architectural decision to employ several diverse GPS services has to be made using the information available at design time. However, at run-time, a different set of compatible services may be available (including services not known at design-time) allowing the service to be maintained either at lower cost or with reduced degradation.

In an alternative, more resilient, approach the components that rely on the GPS service could reconfigure dynamically. There are many options, and they could pursue a policy of trying various alternatives: they could switch to an alternate GPS with higher availability, or use another service in parallel with the low availability one, or ultimately signal a failure. The main point is that the system could evolve dynamically to offer continued, if necessary (predictably) degraded, service using the resources available at the time the negative event is detected. Components are not owned by a single central authority, but may be supplied and withdrawn, at any time, by independent providers.

We are concerned with predictable dynamic resilience of open systems built from components that interact via a network-based infrastructure. By *dynamic resilience* we denote a system's capacity to respond dynamically by adaptation in order to maintain an acceptable level of service in the presence of impairments. By *predictable dynamic resilience* we mean the capacity to deliver dynamic resilience within bounds that can be predicted at design time.

In this paper we present an architectural approach to the achievement of predictable levels of dynamic resilience in distributed systems of the kind described above. Section 2 gives an overview of an architectural model that exploits component metadata to support decision-making and reconfiguration based on the dynamic enforcement of explicitly expressed resilience policies. We indicate the need for formal specifications for achieving predictability in such a model in Section 3. Section 4 presents two proof of concept studies in which we have performed preliminary implementations of core parts of our architecture. A discussion is provided in Section 5. Related work is discussed in Section 6.

2. ARCHITECTURAL MODEL

As suggested by the example scenario above, certain features are essential to providing dynamic resilience in the open and flexible systems that are envisaged in the future. The system architect requires architectural patterns that use run-time information to maintain resilience through adaptation, e.g. by dynamically composing a satisfactory service from lower-specification components. We will refer to such patterns as *dynamic resilience mechanisms* (DRMs). A key feature is the availability at run-time of *resilience metadata* – information about system components, sufficient to govern decision-making about dynamic reconfiguration. Such metadata can be used to guide reconfiguration in accordance with *resilience policies* (e.g. to increase the number of alternate services if availability starts to decline). Finally, we require a *run-time environment* for the acquisition, maintenance and publication of metadata, including reasoning services for performing reconfigurations in accordance with policies. These four main features are summarised in Figure 1. Each component is considered in more detail below.

2.1 Architecture Components

Dynamic Resilience Mechanisms

Dynamic resilience mechanisms (DRMs) are generic, not application-specific, but are realised in application-specific designs as resilience policies. In our scenario, the pattern allowing dynamic selection and parallel composition of services to maintain availability is an example of such a mechanism. The DRM in Figure 1 describes a very simple pattern allowing a dynamic adaptation in response to component availability falling below a specified threshold. It describes the architecture of the Assembly A (a sub-system made in this case of a single component S generating a data stream), names the threshold T to be maintained by that Assembly and describes the adaptation that takes place, in this case a simple replacement of S by a functionally substitutable S'. A precondition on the adaptation is that S' should supply at least the same functionality as S and provide a level of availability higher or equal to the threshold T. In order to perform design-time reasoning (and hence predict levels of resilience), DRMs require theories describing the effects on resilience characteristics of the system. In our example, the theory is just an assertion about the availability of this simple assembly after the adaptation has taken place. In practice, such theories would consist of several assertions. Note that the DRM is a specification for a reconfiguration rather than a detailed description of how it should be implemented – this latter role is performed by the resilience policies executing in the run-time system. The theory allows the prediction of the run-time resilience characteristics of a system containing policies that correctly implement the DRM[1].

Resilience Policies

Resilience policies are descriptions of changes to the running system based on metadata, which may include structural alterations. Resilience is achieved by the enforcement of a resilience policy, defined according to a dynamic resilience mechanism. For example, a policy may specify to provide a replacement component for one whose failure rate exceeds a threshold, or if none exists to find two components of lower reliability that can be composed. Such policies include (re-) configuration aspects, as described above, they may as well include security related policies, such access constraints, or service delivery conditions. Resilience policies, when enacted, use reasoning and adaptation services provided by the underlying run-time environment.

Metadata

Resilience policies, executed at runtime in an open and dynamic system, require appropriate *metadata* - data describing functional and non-functional properties of components, as distinct from the data used by components in the course of their normal operation, and distinct from the code that implements component services. We will use the term *resilience metadata* to refer to metadata that is relevant to system dependability and resilience.

Examples of *non-functional resilience* metadata include: availability or reliability measures (e.g. mean time to fail,

[1]This is only an example of a possible DRM description, more work is needed to actually define a complete language for expressing DRMs.

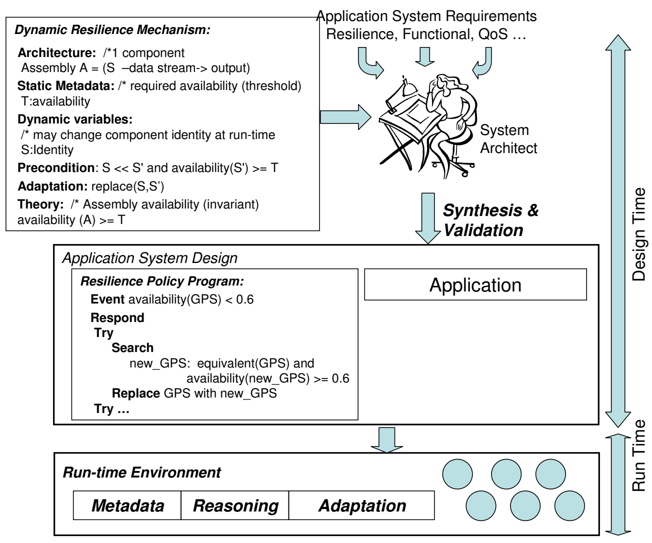
Dynamic Resilience Mechanism:
Architecture: /*1 component
Assembly A = (S –data stream-> output)
Static Metadata: /* required availability (threshold)
T:availability
Dynamic variables:
/* may change component identity at run-time
S:Identity
Precondition: S << S' and availability(S') >= T
Adaptation: replace(S,S')
Theory: /* Assembly availability (invariant)
availability (A) >= T
Application System Requirements
Resilience, Functional, QoS …
System Architect
Synthesis & Validation
Application System Design
Resilience Policy Program:
Event availability(GPS) < 0.6
Respond
Try
Search
new_GPS: equivalent(GPS) and
availability(new_GPS) >= 0.6
Replace GPS with new_GPS
Try …
Application
Run-time Environment
Metadata
Reasoning
Adaptation
Design Time
Run Time

Figure 1: Enabling technologies for dynamic resilience

mean time to repair); description of needed resources (CPU, network bandwidth, memory) for the component to work properly; or trust-based information on other components.

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

Reasoning and Adaptation Services

Figure 2 expands the run-time environment part of Figure 1. The run-time environment itself follows a service-oriented architecture. The *Metadata Registry* stores published trustworthy metadata and maintains the link with the corresponding component. The run-time environment receives requests for components based on metadata information; retrieves and determines the list of components corresponding to a specific metadata request on the basis of the information available in the Metadata Registry. On the basis of those requests, the *Run-time Reasoning Service* provides different tasks related to the processing of metadata stored in the metadata registry, such as comparison/matching of metadata, determination of equivalent metadata information, composition of metadata. This service encompasses automated reasoning over the policies, and any ontology of specific keywords used to express the metadata. The *Architecture Adaptation Service* manages the list of components, seamlessly activates or connects the ones that will be used according to a specified resilience policy program. This service encompasses automated reasoning on formal resilience policy programs.

2.2 Example

Figure 2 shows the GPS example discussed above: three GPS services have been registered in the run-time environment (GPS_1, GPS_2 and GPS_3); their corresponding metadata are available in the metadata registry. Metadata labelled "functionality" are functional resilience metadata that characterise the type of component. Metadata labelled "availability" are non-functional resilience metadata reporting the actual availability of the component. In this example, metadata labelled "availability" are permanently updated (by the components themselves or by some underlying system[2]) in order to reflect the actual availability of the different GPS services. The functionality labelled "GPS" is first provided by the GPS_1 service, but at some point, its availability falls below the required 0.6 threshold (the availability of the GPS_1 service shows a value of 0.5). The Dynamic Resilient Mechanism displayed on Figure 1, established at design time, enforces a change of GPS service if the availability of the service is below a certain threshold. The corresponding resilience policy program, used at run-time, (shown on Figure 2), expresses this resilience constraint. It is based on metadata labelled "availability". The run-time environment then looks for an equivalent service in the metadata registry: it looks for any other service that displays the same functionality (functionality=GPS), and which has an availability greater than 0.6. Service GPS_3 is the only one that satisfies these criteria (availability=0.8), therefore, the run-time environment replaces on-the-fly component GPS_1 by GPS_3.

In this example, metadata are essentially a series of keywords ("GPS", "availability", and "functionality") and numeric values (0.5, 0.6, etc). Reasoning occurs on the logical formulae of the resilience policy as well as on the actual values associated to metadata. Dynamic adaptation consists in replacing on-the-fly the faulty service.

[2]The exact way how metadata is obtained is out of the scope of this paper. We assume here that the run-time environment is fed with the necessary metadata.

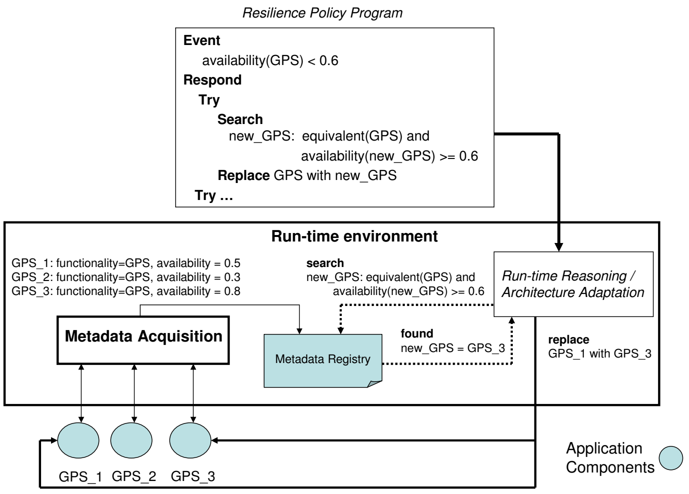
Resilience Policy Program
Event
availability(GPS) < 0.6
Respond
Try
Search
new_GPS: equivalent(GPS) and
availability(new_GPS) >= 0.6
Replace GPS with new_GPS
Try ...
Run-time environment
GPS_1: functionality=GPS, availability = 0.5
GPS_2: functionality=GPS, availability = 0.3
GPS_3: functionality=GPS, availability = 0.8
search
new_GPS: equivalent(GPS) and
availability(new_GPS) >= 0.6
Run-time Reasoning / Architecture Adaptation
Metadata Acquisition
Metadata Registry
found
new_GPS = GPS_3
replace
GPS_1 with GPS_3
GPS_1
GPS_2
GPS_3
Application Components

Figure 2: A run-time environment supporting dynamic resilience

3. ON THE NEED FOR FORMAL SPECIFICATIONS

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

Describing Metadata

It may be helpful to distinguish between value-based metadata, such as availability distributions, probabilities of failure etc., and functional description metadata such as preconditions, post-conditions and descriptions of known failure modes. Value-based metadata typically quantify some non-functional property and may be expressed as structured values drawn from a possibly ordered collection. Resilience mechanisms employing such metadata will allow explicit reasoning about reliability, performance, quality of service, availability of each component followed by system run-time evolution. This includes retrying after a dynamically-chosen period of time, signalling a failure when the component does not deliver the service of the required quality, switching from the less reliable component to a more reliable one, and, more generally, applying dynamic mechanisms based on the recovery block scheme [15] or employing a dynamic set of components as versions in n-version programming [2].

Component specifications may likely change as components evolve, but resilience may be maintained if it is possible to reason dynamically about the functional properties of components, including abnormal behaviours. Dynamic resilience mechanisms require component specifications as metadata, including both specifications of services offered and services required. These can be given by logical expressions of pre-/post-conditions (in some cases rely/guarantee conditions [8]), potential failure behaviours and responses when the components are used outside their pre-conditions. These are a very different form of metadata from the value-based examples. In particular, the on-line reasoning required to deal with such functional description metadata requires more complex logic than may be the case for value-based metadata.

Describing Composition

Within an architecture, system components are composed by connectors. In a Web services environment, for example, the individual service invocations that make up an application are composed by means of combinators defined in a workflow language (e.g. parallel or serial composition). If a reconfiguration is to take place, it is necessary to know what the effects of the reconfiguration may be on the dependability properties of interest. In our GPS example, it is necessary to be able to predict the effects of parallel composition on service availability, in order to determine how many GPS services should be used, and which ones. This implies that the composition mechanism should have a semantics, at least in terms of the metadata of interest. This semantics is built in to the metadata reasoning tool, and so should be formal. For some existing architectural description languages [20, 1], such theories already exist, although they tend to deal primarily with the composition of functional rather than QoS properties.

Describing Policies

For many applications, we require that systems should be predictably dependable in the sense that degraded modes of operation and the circumstances in which they occur are understood in advance of deployment. This requirement for predictable dependability tends to lead towards the use of formal descriptions of system requirements, structural and security policies, as well as specifications of components. Resilience policies bring together metadata and composition theories to describe the trigger conditions under which reconfigurations occur, and the reconfigurations themselves.

Policies should be formal in the sense that their behaviour may be analysed in advance of deployment. In particular, the effects of interactions between multiple policies should be susceptible to some level of prediction.

Potential and limits

The use of dynamically updated metadata at run-time becomes increasingly useful for a wide range of open systems particularly for ambient intelligence systems (e.g. rescue scenario, tag-based systems for entertainment or road traffic monitoring). Services or components participating in such systems may be: data service providers (GPS example, sensor networks, videos or music files providers, etc.); or managing service providers (network routing in mobile ad hoc network - MANET, file system provider in a P2P system, task and resource allocation in Grid computing system, or dynamic access control).

The use of resilience policies in combination with metadata provides an even powerful tool for supporting these applications when facing the dynamic changes of the environment in which they evolve, e.g. by choosing the best adapted service at any time, receiving data despite failures from an original sender, maintaining networks functionalities or security constraints, or realising and maintaining Grid computing resource allocation.

On the one hand, formal specifications allow ensuring predictability when designing the applications. However, on the other hand, they may limit the efficiency of the approach since the complexity and overhead introduced by underlying reasoning tools are directly related to the expressiveness of the formal specification language: the more expressive the language, the more difficult and inefficient at run-time the reasoning tool will be.

4. PROOF OF CONCEPT STUDIES

We describe two studies realised in the above context. Study 1 focuses on the gathering of metadata and its use to reconfigure workflows in e-Science. Study 2 concentrates on the issues of dynamic reconfiguration using the run-time environment services described in Section 2.

4.1 Study 1: Acquisition and Use of Metadata

In this preliminary work we have developed a tool that records in a database the dependability metadata derived from continuous observations of Web services. The tool measures the dependability of Web services by acting as a client to the Web service under investigation. It monitors a given Web service by tracking the following characteristics:

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

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

We have been analyzing existing workflow languages used for composing e-science Web service applications and shown that it is possible to perform a simple reconfiguration of a workflow (specifically a SCUFL workflow developed in the Taverna environment[3]) by selecting a service or composition of services on the basis of availability metadata, to achieve the desired probability of successful completion of the task specified in the workflow. The metadata were collected and updated by the monitoring tool[4].

4.2 Study 2: Reconfigurable Service-oriented Architecture

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

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

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

Implementations. Two different implementations of the above architecture have been realised. The first implementation has been realised for specifications expressing: signatures of available operators whose parameters are Java primitive types; and required quality of service. Both operators name and quality of service are described using predefined keywords [13]. 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 [7]. This prototype supports specifications written either in Prolog, or as regular expressions. However it cannot check together specifications written in two different languages. In the case of Prolog, the middleware calls SWI Prolog tool to decide about the conformance of two specifications, in the case of regular expressions we have implemented a tool that checks two regular expressions. In our current implementation using Prolog, specifications are registered as Prolog facts and rules, while specification requests are Prolog queries. In the case of regular expressions, the registered specification must match as a regular expression the specification request (but not necessarily the opposite).

4.3 Dynamic Resilience Experiments

The above described implementations, even though serving as a proof of concept, have been turned to be useful to carry on the following preliminary experiments on some dynamically resilient functionalities.

[3]http://taverna.sourceforge.net

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

Dynamic Re-Configuration

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

For the particular case of automatic and seamless integration of new components, initial experiments with Study 2 have proven useful for dynamic run-time evolution of code. Indeed, without stopping any specific entities or the whole system, it has been possible: to add in the system and seamlessly use additional features; to seamlessly replace updated entities without the calling entities noticing the replacement (even during a call [13]). Since a specification request is the only element necessary for activating a service, inserting a new functionality then simply consists in registering its corresponding service to the middleware. Replacing an updated entity consists in having both the old and the new entities present at the same time in the system, and if necessary to transfer the state of the old entity to the new one before stopping the updated entity.

Dynamic Optimisation

Study 1 demonstrated that it was possible to choose the most suitable Web service or to employ/combine several Web services if the availability of individual Web services was not good enough. In addition, in Study 1 we extend the workflow language with an additional functionality to support a search for the most dependable service out of the set of the service which are now monitored by the tool. This will allow us to add some degree of self-optimisation to the existing bioinformatics systems.

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

5. DISCUSSION

The successful application of our architecture is predicated on certain assumptions about the supporting technologies and their future development.

Accuracy of collected metadata

Our approach relies to some extent on the accuracy of dynamically acquired metadata. We believe that developments in this area are capable of providing sufficiently accurate metadata for useful applications. In addition, the reasoning tool may take accuracy into account, reasoning within certain tolerances.

In Study 1, the accuracy of metadata used depends on some factors outside of the composed application's control as the probe requests are sent over the Internet. Nevertheless, we believe that the claim about their accuracy is correct because the mechanism used to collect and process them allows their accuracy to be improved by dynamic or static adjustment of the probing frequency. In addition, the metadata accuracy requirement is not very stringent for this application, as the main decision about reconfiguration is made using metadata collected over long period of time (e.g. availability and performance).

Predictability

The reasoning tool described above is intended to work as a matching and replacement tool using automated reasoning on specifications, but is 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 conflicting policies, it may also happen that some non-expected (emergent) configuration, possibly non ideal, is actually the chosen one. This is an issue of further research and study.

Component Reconfiguration and Replacement

In general, reconfiguration or replacement of a faulty or degraded component may require the restoration of state. Such restoration may not be tractable at certain times. Our approach assumes that we can time such changes so as to minimize the restoration costs. It is worth noting that we are not proposing dynamic adaptation as a form of fault-tolerance, but as a technique for helping long-living applications to meet their quality requirements. Thus a failure event does not necessarily trigger an instant reconfiguration. The current configuration deals with failures by means of whatever fault tolerance strategies are built into it. Dynamic reconfiguration permits to switch to a solution with longer-term benefit, but which may need more time to be completed.

We realise that in some situations (e.g. when the components have state) switching between various fault tolerance schemes should be supported by a number of dedicated services, one of which is state recovery/restore and another one is state mapping between diverse components. There is some useful background work on recovery of the failed versions in n-version programming which can be applied here (e.g. [16]).

Study 2 has addressed the state change problem, and a state capture facility has been implemented allowing transfer of state to a replacing component. Study 2 also identified the different cases where a replacement (and consequently the state change) could be carried out.

6. RELATED WORK

Predictable Resilience by Policies

Architecture adaptations are mainly seen as component reconfigurations, generally governed by policies described either at the general level (as guidelines, directives, or constraints on actions or states) or at the detailed level of technical reconfiguration actions [21]. Work on the analysis of policies is more limited, the main approaches being based on quality of service (QoS) awareness [5] and arithmetic on QoS attributes. Architectural reconfiguration using evolution has been described using architecture description languages (ADLs) [11, 12]. Several approaches have been proposed for compositional reasoning based on formal methods [4, 17] but not linked with component models and resilience metadata. Our proposed model aims to integrate design of resilience policies (using component metadata and dynamic resilience mechanisms) and application design, and to provide a general approach to the architectural design of the non functional requirements of distributed systems using explicit metadata.

Trustworthy Resilience Metadata

Research on Web Services and Grid computing systems heavily studies metadata. Among others, we can cite Astrolabe [22] and the Grid Monitoring and Discovery System (GMDS) [19]. Astrolabe uses a relational database to monitor the dynamic state of a set of distributed resources on which it is installed. GMDS uses interfaces within the Web Services Resource Framework to allow resources to publish information and provide querying; the Grid Index Service collates such information and makes it available in one place. Such systems assume homogeneity: they are required to be running on the participating computer nodes. This can not be guaranteed when application components are provided dynamically by various suppliers in open, boundaryless, distributed systems. Our proposed model aims to permit collation and run-time analysis of metadata from such disparate sources.

Reasoning and Adaptation Services

Formal reasoning tools (theorem provers and model checkers) have been developed for off-line reasoning in design; they require human guidance in certain situations, and are not suitable for run-time usage. Attempts to use lightweight run-time reasoning include: reasoning over ontologies using description logics such as [23], or dynamic model checking supporting interaction among autonomous agents [14]. In this latter work the goal is to check automatically and at run-time if a given property (such as a security constraint) can be verified given the local constraints of individual agents (e.g. access control, payments constraints, etc.) are compatible with the multi-agent interaction model (i.e. protocol) the agents want to participate in.

In the context of sensor networks, the Semantic Streams framework [24] allows users to introduce (high-level/semantic) queries over real-time data provided by existing sensors located at diverse position throughout the world. At run-time, the system responds to queries by automatically combining the different (most appropriate) data provided by the sensors. The Semantic Streams framework encompasses a markup and query language working over SICStus Prolog for encoding sensor data description. Automated reasoning is then applied on these descriptions in order to answer queries. The approach is very similar to Study 2 presented in this paper, while Study 2 focuses on the description of services and on semantic queries requesting available services, Semantic Streams focuses on queries over real-time sensor data and allows composition of different data in order to answer possibly complex queries. The architectural model proposed in this paper goes a step further since: it allows any service (data provider services, managing services, etc.) to join the system, reasoning is applied on requested service functionalities as well as on service resilience (through the resilience policy programs), dynamic replacement or reconfiguration of the system is supported by the underlying run-time environment.

7. CONCLUSION

There has been so far relatively little examination of resilience mechanisms that exploit component specifications as metadata in the dynamic system. However, the potential for publication of component specifications as metadata exists in several modern system architectures (such as service-oriented architectures [10]). Furthermore, advances in automated reasoning such as model-checking techniques [6], although generally only applied to design-time evolution, raise the potential for dynamic reconfiguration based on this metadata, at least for constrained cases, and make this an area well worth studying as basic research.

In this paper we present a novel approach to the development of dynamically resilient systems using a specific architectural model. Its basic idea is to separate functional and non-functional descriptions of the individual components, to explicitly and formally express various resilience policies on the basis of metadata, and to enforce the policies by using a dedicated service-oriented middleware.

We have shown the viability of the approach through two preliminary proof of concept experiments. Building on these results, future work will essentially focus on: 1. defining an appropriate formal specification language for expressing resilience policies, metadata and component descriptions; 2. developing the corresponding run-time reasoning tools.

8. ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewers for their comments and to Yuhui Chen for his work on Study 1. Alexander Romanovsky is supported by IST FP6 Project on Rigorous Open Development Environment for Complex Systems (RODIN).

9. REFERENCES

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering*, 6(3):213–249, 1997.

[2] A. Avizienis. The N-Version Approach to Fault Tolerant Systems. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[4] M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional Verification of Middleware-Based

Software Architecture Descriptions. In *International Conference on Software Engineering (ICSE'04)*, pages 10–24, 2004.

[5] H. Cervantes and R. S. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *International Conference on Software Engineering (ICSE'04)*, pages 614–623, 2004.

[6] S. Chaki, N. Sharygina, and N. Sinha. Verification of evolving software. In *Proc. Workshop on Specification and Verification of Component-based Systems, 12th. ACM Symposium on Foundations of Software Engineering 2004*, 2004.

[7] G. Di Marzo Serugendo and M. Deriaz. Specification-carrying code for self-managed systems. In J.-P. Martin-Flatin, J. Sventek, and K. Geihs, editors, *IEEE International Workshop on Self-Managed Systems and Services*, 2005.

[8] C. H. C. Duarte and T. Maibaum. A rely/guarantee discipline for open distributed systems design. *Information Processing Letters*, 74:55–63, 2000.

[9] D. Estrin, editor. *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers.* Computer Science and Telecommunications Board, National Academy of Sciences, Washington, D.C., 2001.

[10] J. S. Fitzgerald, S. Parastatidis, A. Romanovsky, and P. Watson. Dependability-explicit computing in service-oriented architectures. In *Supplementary Volume of Proceedings of International Conference on Dependable Systems and Networks*, pages 34–35, 2004.

[11] H. Gomaa and M. Hussein. Software Reconfiguration Patterns for Dynamic Evolution of Software Architecture. In *4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pages 79–88, 2004.

[12] R. Morrison and al. Software Architectures in the ArchWare ADL. In *4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pages 69–78, 2004.

[13] M. Oriol and G. Di Marzo Serugendo. A disconnected service architecture for unanticipated run-time evolution of code. *IEE Proceedings-Software, Special Issue on Unanticipated Software Evolution*, 2004.

[14] N. Osman, D. Robertson, and C. Walton. Run-Time Model Checking of Interaction and Deontic Models for Multi-Agent Systems. In *Autonomous Agents and Multi-Agents Systems*, 2006.

[15] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, 1(2):221–232, 1975.

[16] A. Romanovsky. Class diversity support in object-oriented languages. *Journal of Systems and Software*, 48(1):43–57, 1999.

[17] H. Schmidt. Trusted Components - Towards Automated Assembly with Predictable Properties. In *ICSE Workshop on Component-Based Software Engineering*, 2001.

[18] F. Schneider, editor. *Trust in Cyberspace: Report of the Committee on Information Systems Trustworthiness, Computer Science and Telecommunications Board, Commission on Physical Sciences, Mathematics and Applications, National Research Council.* National Academy Press, Washington, D.C., 1999.

[19] J. Schopf, M. D'Arcy, N. Miller, L. Pearlman, I. Foster, and C. Kesselman. Monitoring and Discovery in a Web Service Framework: Functionality and Performance of the Globus Toolkits MDS4. Technical Report ANL/MCS-P1248-0405, Argonne National Laboratory, 2005.

[20] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996. 242p. ISBN 0-13-182957-2.

[21] E. Turkay, A. S. Gokhale, and B. Natarajan. Addressing the Middleware Configuration Challenges using Model-based Techniques. In *ACM Southeast Regional Conference*, pages 166–170, 2004.

[22] R. Van Renesse, K. Birman, and W. Vogels. A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003.

[23] P. Weinstein and W. P. Birmingham. Service Classification in a Proto-Organic Society of Agents. In *IJCAI Workshop on Artificial Intelligence in Digital Libraries*, 1997.

[24] K. Whitehouse, F. Zhao, and J. Liu. Semantic Streams: A Framework for Composable Semantic Interpretation of Sensor Data. In *3rd European Workshop on Wireless Sensor Networks (EWSN 2006)*, pages 5–20, 2006.

Giovanna Di Marzo Serugendo is Lecturer at the School of Computer Science and Information Systems, Birkbeck College, University of London. Her main interests are in the software engineering of decentralised systems with self-organising and self-adaptive capabilities. She has recently been appointed Editor-in-Chief of ACM Transactions on Autonomous and Adaptive Systems.

John Fitzgerald is Reader in Computing Science at Newcastle University. His main area of work is on formal methods, particularly industry application of model-oriented techniques and proof. He is currently leading work on metadata-based description of resilience mechanisms within the EU ReSIST Network. He is Chairman of Formal Methods Europe.

Alexander Romanovsky is Professor at the School of Computing Science, Newcastle University. The main focus of his work is on fault tolerance and exception handling. In recent years he has been involved in a number of EU and EPSRC projects on various aspects of dependability. Now he is coordinating the EU RODIN project.

Nicolas Guelfi is Professor at the Faculty of Sciences, Technology and Communications of the University of Luxembourg. His main research activities concern the engineering and evolution of reliable and secure distributed and mobile systems based on semi-formal methods and transformations. He is a leading member of the Laboratory for Advanced Software Systems (LASSY). He has been involved in several European projects and is chair of the ERCIM working group on rapid integration of software engineering techniques (RISE).