Reflective Component-based Technologies to Support Dynamic Variability*

Nelly Bencomo, Gordon Blair, Carlos Flores, Pete Sawyer

Computing department, InfoLab21, Lancaster University, LA1 4WA, United Kingdom
email: {nelly, gordon, floresco, sawyer} @comp.lancs.ac.uk

Abstract

In this paper we propose an approach to support dynamic or runtime variability in systems that must adapt dynamically to changing runtime context. The approach is founded on reflective component-based technologies to support the dynamic variability at the architectural level. Adaptive behaviour is encoded in reconfiguration policies that are consulted at run-time when changes in the underlying environment are detected. Specifically, the reconfiguration policies dictate the component-based architecture to be used in actively changing contexts. However, the increasing number of variants and their interdependency relationships add to the complexity of variability management. Therefore, the paper also proposes a notation and associated models to address the management of dynamic variability. We describe our experience with applying this approach through a case study; the support and management of dynamic variability for service discovery protocols. ***Keywords****: dynamic variability, architectural reconfiguration, orthogonal variability models.*

1 Introduction

It is becoming common that systems should be able to adapt dynamically to changing contexts at runtime. Such systems exhibit degrees of variability that depend on runtime fluctuations in their contexts. We call this kind of variability *dynamic variability* or runtime variability. Although dynamic variability has been addressed by long-established concepts in the field of system families [16, 17, 34], we advocate that this work is insufficient to meet the needs of contemporary, dynamically adaptive distributed systems. In the case of systems that adapt dynamically, approaches to support variability cannot be just component specializations or conditions on variables [34]. Decisions should involve more powerful mechanisms that allow dynamic reorganization of the architecture. In other words, these mechanisms should be able to manage whole sets of components, their connections and associated semantics. In this paper we introduce an approach based on component frameworks [35] and reflection [23] as a mechanism to support the realization of dynamic variability of adaptive systems. Using this approach, unanticipated variability and interdependency relationships between variable software artefacts (such as components, connections, and components configurations) and context and environment conditions can grow to such a level that rigorous support for variability management is needed. Therefore, this paper also discusses notation and models to manage dynamic variability. The application of the proposed approach consider using a case study.

In the reminder of the paper we discuss dynamic variability in adaptive systems and the need for its management (Section 2). We then introduce the fundamental concepts of our approach (Section 3) and present a case study (Section 4). A discussion about the contributions of our research and related work are presented (Section 5), and finally, some remarks and future work are given (Section 6).

2 Dynamic Variability

2.1 Overview

One of the reasons for software variability is to delay a design decision [34]. Instead of deciding on what system to develop in advance, a set of components and a common system family (reference architecture) are specified and implemented during a process called *Domain Engineering* [12]. Later on, during *Application Engineering*, specific systems are developed to satisfy the requirements and reusing the components and architecture. Variability is expressed in the form of *variation points*. A variation point denotes a particular location in a software-based system where decisions are made to express the selected *variant* [34]. Eventually, one of the variants should be chosen to be achieved or implemented. The time when it is done is called *binding time*.

*This work has been supported in part by the EPSRC project EP/C010345/1 The Divergent Grid

Traditionally, decisions have been deferred to architecture design, implementation, compilation, linking, and deployment [1, 7, 12, 22, 26, 34]. Currently the aim is to postpone these decisions to even later points in time to allow dynamic variability at runtime. This raises several research challenges, such as the management of variabilities in dynamically adaptive systems, which are discussed in the next section.

2.2 Dynamic Variability in Adaptive Systems

A dynamically adaptive system operates in environments that impose changing contexts and requirements. The challenge that it causes comes from the need to support unanticipated adaptation or customization of the systems [32] according the needs of the fluctuating environment. The unanticipated conditions are related to:

(i) Environment or context variability: commonly the evolution of the environment cannot be predicted at design time; therefore the total range of contexts and requirements may be unknown at design time.

(ii) Structural variability: it covers the variety of the components and the variety of their configuration. This is consequence of the variability explained above. In order to satisfy the set of requirements for the new context, the system may add new components or arrange the current structural configuration (reconfiguration). Hence, the solutions cannot be restricted to a set of known-in-advance configurations and components.

The system should be prepared to deal with these two dimensions of variability described above. Adaptive systems must be prepared to identify a new context unknown at design time. Under the conditions of the new contexts, the system must be prepared to discover and include new components to meet new requirements or simply to improve the current state of the system when new components become available [32] and according some quality of service (QoS) properties. Moreover, solutions to manage the latter structural variability cannot be just the traditional component replacements and/or specializations, but decisions should involve more powerful mechanisms able to manage whole sets of components, their connections and semantics (configurations). A balance between the support for unanticipated adaptive capabilities and the guarantee of the correct structural composition and state of the system is another important challenge that must be taken into account.

The classification proposed in [36] distinguishes two different types of dynamic adaptation: *dynamic behaviour adaptation* and *dynamic reconfiguration*.

In dynamic behaviour adaptation, systems recognize new environmental conditions not envisioned during development. In this systems, control and order is emergent rather than predetermined [13, 37]. This kind of adaptation is proposed by researchers using mechanisms based on genetic algorithms or neural networks. Research on this kind of adaptation is still at an early stage to propose sound solutions for complex adaptive systems.

Dynamic reconfiguration requires that all feasible variants of behaviour can be somehow predefined before execution. During execution, the current state of the system and its environment and context is evaluated and the most appropriate behaviour variant is selected; i.e. the system is dynamically reconfigured using the most appropriate variant (configuration). Dynamic reconfiguration can be realized using two approaches: *software-based configuration* and *hardware-based configuration*. The latter is omitted in this research as the authors are concerned only with software-based reconfiguration.

Software-based reconfiguration can be achieved using two approaches: *pre-determined reconfiguration* and *online-determined reconfiguration*. *Pre-determined reconfiguration* is based on a set of configurations with known impact defined before the deployment of the application. In this case, the system only supports the configurations that are hard coded and fixed in advance by the developers. Therefore the system is only reconfigured (i.e. the system adapts) when in a predefined and hardcoded configuration. The implementation of a new configuration requires the system to be reinitiated. This case is very restrictive. The last case, *online-determined reconfiguration*, is a solution in-between pre-determined reconfiguration and dynamic behaviour adaptation. With online-determined reconfiguration, the system has a mechanism to identify the possible configurations at runtime. Online-determined reconfiguration supports dynamic extension of the application by adding, changing and removing artefacts (e.g. components and connections) at runtime. This approach "*requires a complex reconfiguration framework*" [36].

3 Achieving Dynamic Variability: our approach

To address the challenges posed by dynamic variability explained above, we propose the use of component frameworks and reflection as a flexible mechanism for supporting runtime variability. At Lancaster University, we have gained experience developing adaptive systems and middleware platforms using *component frameworks* and *reflective technologies* [8–10]. Component frameworks are collections of components that address a specific area of concern and accept "plug-in" components that add or extend behaviour [3, 9]. Reflective capabilities support introspection to observe and reason about the state of the system to make decisions on architectural reconfigurations. Adaptive behavior is defined by sets of *reconfiguration policies*. These

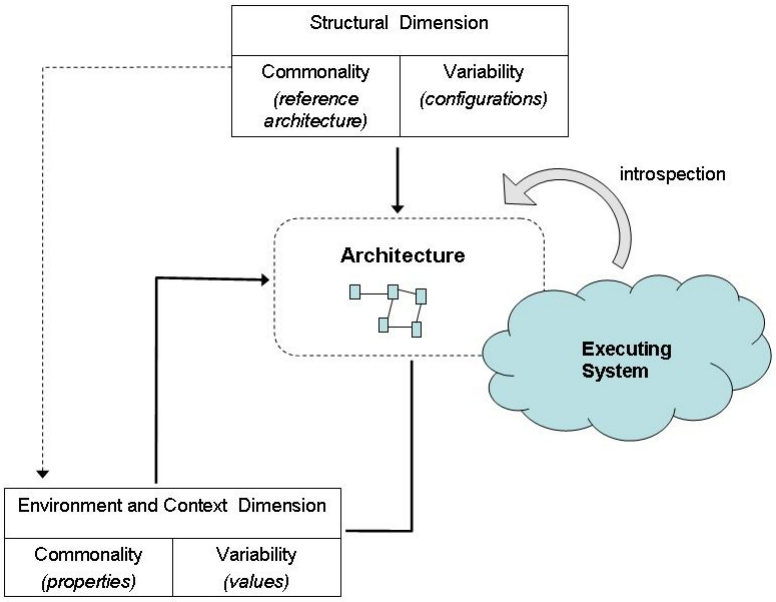

Figure 1. Dynamic Variability Dimensions

policies are of the form *on-event-do-actions* and *actions* are architectural changes using the component frameworks. A *context engine* receives relevant *environmental events* that are employed to identify the reconfiguration policy to be used. Crucially, component frameworks offer the medium to provide structural variability. Reflective capabilities offer the potential to reason about the possible variation points and their variants during execution. The proposed solution use the online-determined reconfiguration category of dynamic adaptation explained above.

Dynamic Variability: Dimensions

The approach deal with the two dimensions of dynamic variability identified, see Figure 1. The architecture defined by the component framework (reference architecture) basically describes the structural commonalities. Different configurations or *structural variants* will exist that follow the well-defined constraints imposed by the component frameworks. Policies describing the contexts and requirements will drive the evolution and execution (using reconfigurations). Essentially, the policy mechanism will set the basis for dealing with the *environment and context variability* identified above. The approach separates the application-specific functionality from the adaptation concerns, thereby reducing complexity [27].

Dynamic Variability: Models

The increasing number of variants and their relationships can make the management of variability a challenge. We propose a model-driven approach to manage the two dimensions of dynamic variability. A model of the structural variability specifies the architecture of the system that will evolve over time during the execution. A model of the environment and context variability specifies the conditions and events that will trigger changes in the architecture. Crucially, these models can be constructed using the domain-specific language-based tool called Genie [2, 4]. From the models designed using Genie, generation of different software artefacts including component source code, component framework configurations, and the reconfiguration policies [31] can be performed. The next section illustrates the case study introducing the fundamental concepts of the proposed approach.

4 Case Study: Dynamic Service Discovery

This section introduces an example of how our approach supports runtime variability for adaptive systems. Firstly, we present the motivation for dynamic service discovery and discuss the domain problem. We follow with the description of how commonalities and variants are identified and finally the variability model and its notation and application are described. The case study is in the context of mobile computing environments applications which need to

dynamically discover services from a wide range of options. A service discovery application proposes a good example of kind of systems that need support for runtime variability.

4.1 The Need to Dynamically Discover Services

Nowadays mobile computing is pervasively taking over traditional computing [14, 19, 25]. Mobile devices are characterized by sudden and unexpected changes in execution context. Applications running on these devices need to dynamically adapt according to the changing contexts. If devices (PDA, mobile, or laptop) are capable of detecting changes in the current environment, then they can also notify the user about new available services according to predefined preferences (e.g. comparison prices and categorized sales in a supermarket or printing services in an internet cafe). Service discovery protocols (SDP) were conceived to simplify the discovery and use of network resources such as printers, video cameras, directories, and mail servers, with minimum user intervention. Many different approaches to tackle different challenges related to heterogeneity of technology have led to a variety of proposed designs for SDPs [24]. Consequently it is not possible to completely foresee at design time which protocols will be used to advertise services in a given context or environment. The next section presents a solution to overcome the challenges posed by heterogenous service discovery protocols.

4.2 Family of Service Discovery Protocols for Adaptive Systems

Flores et all [15] present a configurable and reconfigurable middleware solution for the dynamic discovery of services advertised using heterogenous protocols in diverse environments. The solution takes into consideration a set of common core architectural elements that individual discovery protocols follow. Using the final architecture, individual discovery platforms can be implemented and dynamically plugged-in to the discovery middleware. Hence, different SDP personalities can be used to discover services advertised by heterogeneous platforms. This middleware solution has been evaluated with the development of four existing ad-hoc service discovery protocols: *ALLIA*, *GSD*, *SSD*, and *SLP* (i.e. 4 personalities). The offered solution enhances configurability and re-configurability and minimizes resource usage through reusable assets such as components and patternsof interaction [15].

Service Discovery Agents

A service discovery interaction platform uses three kinds of agents to advertise and discover services:

-**User Agent (*UA*)** to discover services on behalf of clients,

-**Service Agent (*SA*)** to advertise services, and,

-**Directory Agent (*DA*)** to support a service directory where *SA*s register their services and *UA*s send their service requests. A *DA* stores temporal service advertisements, matches requested services against advertisements, and replies to requesting clients when a positive match is found.

The agents identified above can be seen as *roles* that individual protocols assume. Depending on the required functionality, participating nodes using a given protocol personality might be required to support 1,2, or the 3 roles at any time.

Service Discovery Family Architecture

The architecture is shown in Figure 2. The six components of the architecture are detailed below:

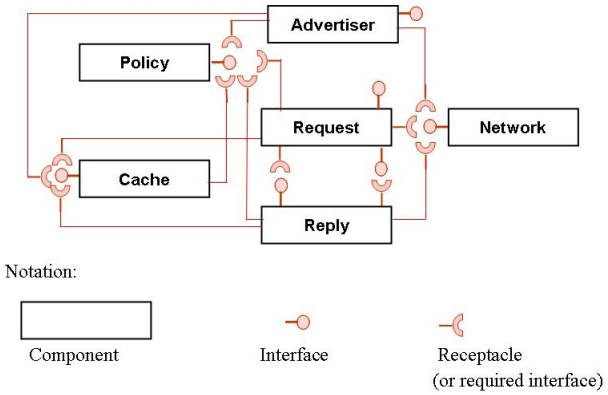

Figure 2. The Service Discovery Family Architecture

***-Advertiser Component*:** this component is utilized by *SA*s to advertise its services and by *DA*s to process incoming service advertisements storing them in cache. This component also deals with protocol messages related with the maintenance of a directory overlay network.

***-Request Component*:** this component is utilized by *UA*'s to generate service requestes. It is also employed by *DA*s to process incoming service requests, match them against local services previously stored in a cache. It can also forward request messages in both roles.

***-Reply Component*:** this component is used by both *UA*s and *DA*s to generate service replies when a positive match request-service occurs or to notify applications from a received replied request respectively.

***-Cache Component*:** common tasks performed by this component are the management of temporary data, storage

of received service advertisements, description of local services and location of neighbouring directories.

***-Policy Component*:** this component stores and deals with user preferences, application needs and/or inclusive context requirements.

***-Network Component*:** this component allows components connected to it to transmit and receive messages utilizing different routing schemes.

4.3 Commonalities and Variabilities

The common architecture explained above dictates the rules to be followed by the possible variants (i.e. configurations). In our specific case, any SDP personality in any environment and under any context needs the network component to interface with networks services or clients, and policies and cache components are always required since they interact with either discovery role. Therefore, the *Network*, *Cache* and *Policies* components will always be present in any valid configuration. The other three components and their bindings will be part of the configuration or not, depending on the roles the protocol might perform (i.e. *SA*, *UA*, or *DA*). Hence, roles (agents) directly define the structural variants.

Figures 3(a) and 3(b) show how the architecture can be configured to support either a *UA* or *SA* role by restricting the number of components to only those required to provide the determined functionality. By using a complete framework configuration, a *DA* can also be supported and the configuration to be used is shown in Figure 3(c). Hence, by configuring individual protocols according to the role (i.e. *UA*, *SA* or *DA*), the number of resources required by a multi-personality middleware service discovery can be significantly reduced to improve the footprint and potentially the performance of the system [15]. Notably, with the multiprotocol middleware platform, heterogeneous discovery platforms can be implemented with a common component architecture. This simplifies the configuration process since the component types and connection bindings remain the same for any protocol implementation. Thus, because of the common configuration pattern, the execution of a simple single component replacement algorithm is enough to re-configure the architecture. Similar common algorithms are required to perform a coarse-grained re-configuration when loading a new discovery personality or when changing the role in a given personality is required. Fine-grained and coarse-grained changes can be made in the framework to support context changes in the environment. Individual protocols can be changed in a fine-grained manner to, for instance, replace the network component with a new one to support a different routing scheme.

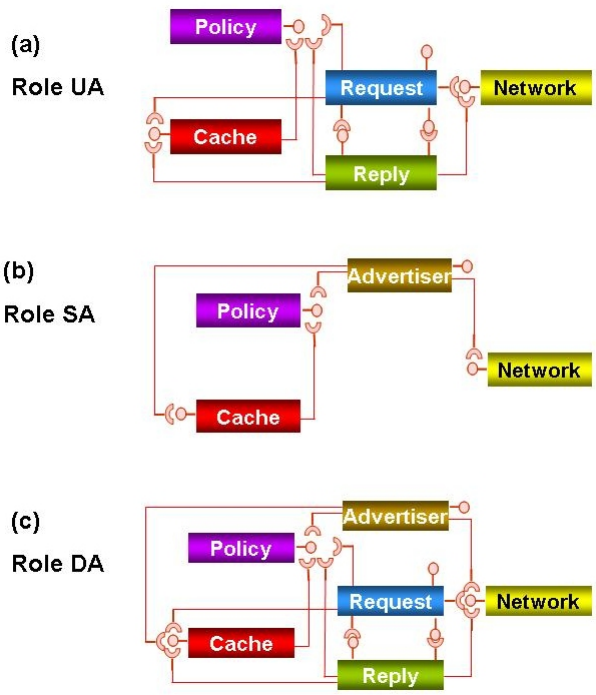

Figure 3. Configurations for the Different Variants

4.4 Variability Model

The approach presented above notably enhances reconfigurability. However, the increasing number of variants and their relationships make it crucial the structured management of variability. This section shows the notation and models we propose to address variability management.

4.4.1 Modelling Structural Variants

The model is based on the orthogonal variability modeling approach proposed in [28]. An orthogonal variability model defines the variability of a system family, i.e the variability information is in a separate model in the form of variation points (VPs) and variants. It associates the VPs and variants defined with other software development models such as design models or component models. An orthogonal variability modelling approach offers many advantages like (i) it promotes a good separation of concerns as the orthogonal models provide a cross-sectional view of the variability across other development artefacts (using the relationship 'artefact dependency'), (ii) in our specific approach, it proposes the basis for the management of traceability between the runtime variability models, the implementation models and the requirement models [31].

Figure 4 shows the model for the role variants explained

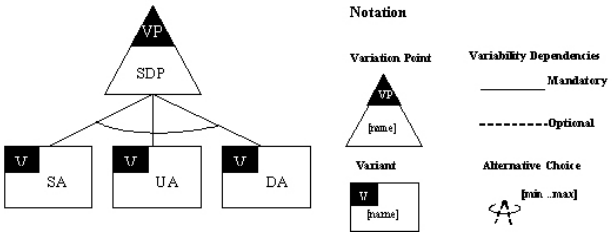

Figure 4. Variability Model

above. Essentially, different configurations associated with roles correspond to variants. The variability diagram describes the variation point "SDP" (Service Discovery Protocol) with three (structural) variants *SA*, *UA*, and *DA*. These variation points and variants associated correspond to the management of the *Structural Variability* described in 2.2

Figure 5 shows the use of the relationship 'artefact dependency', each structural variant is associated with the corresponding configuration. The configurations are represented using either XML or binary files to describe or perform their topologies. The specified variation points are specialized by a runtime selection between alternative component configurations. These configurations are designed using the domain-specific language (DSL) tool Genie to automatically generate the corresponding XML file or hard code as explained in [2].

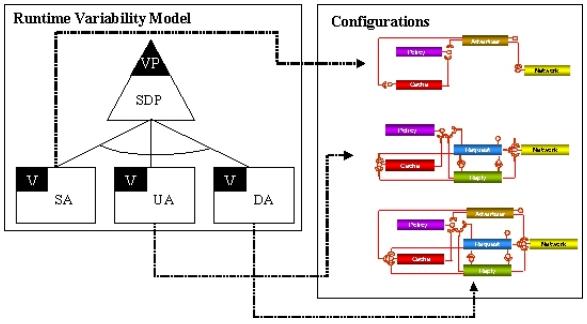

Figure 5. Variants SA, UA, and DA and their configurations

Other VPs and variants are also specified; for example the VP "Personality" defines the personality variants, i.e. *ALLIA*, *GSD*, *SSD*, *SLP* or any other specified in the future (see Figure 6).

4.4.2 Modelling Reconfigurations

An interesting aspect of adaptive systems like SDPs is the need to dynamically reconfigure the system from one variant to another when the context has changed. Examples of opportunities for reconfigurations (i) changes of the underlying network protocols if the operation changes to an ad-hoc domain, (ii) the use of a different role strategy if the system size increases (for scalability reasons), (iii) the use of a different role strategy to save energy (this example is explained below). This is quite distinct from traditional system families where, once a member (product) of the family is created, it does not change significantly during the lifetime of the software product. Using our approach a member of the family may be "transformed" into another one to adapt the system to meet the new requirements and suit a new context. To do this, the system should monitor specific aspects of the runtime environment and react to given changes while keeping a valid state. The system should be able to decide what kind of reconfiguration has to be performed if any. To model this behaviour it is necessary to define what adaptation means in terms of configurations.

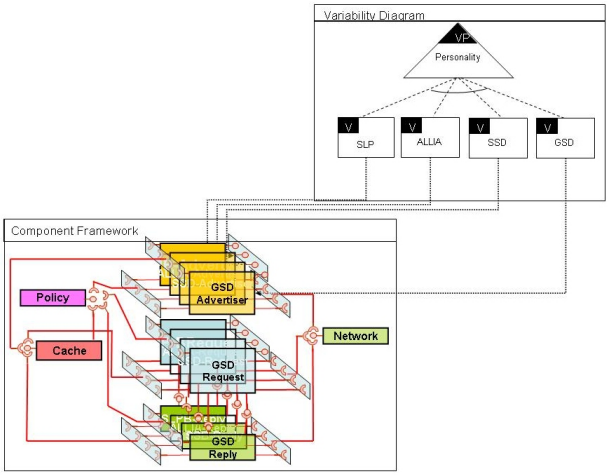

Figure 6. Variants Personalities

An *adaptation* is defined as the process of having the system going from a given configuration *Ci* to another configuration *Cj* given the conditions of the context *Tk*. The possible adaptations will be captured by the variability model.

The variability model of the case study is extended to show how an adaptation from one role (agent) to another is performed, i.e. it shows the context and requirements and the reconfiguration involved. Essentially, this is modeled using *transition diagrams*. A screenshot of the Genie model that specifies the transition diagram designed for service discovery protocols is at the bottom of Figure 7. An adaptation policy is associated with the relationship (arc) between the configuration for the variant *UA* (*Ci*) and the configuration for *DA* (*Cj*) for a given context *Tk* specified by the policy. The number of transitions (arcs) and adaptation policies to be inserted in the transition diagram will depend on how adaptable the system should be or is conceived. The transition diagrams and the policies associated correspond

to the management of the *Environment and Context Variability* described in Section 2.2.

Figure 8 shows an overview of the two dimensions of dynamic variability for the case of Service Discovery Protocols.

The following examples illustrate reconfiguration opportunities identified for the case of the protocol personality *SSD*.

Example 1: Nodes operating *SSD* protocols might run periodically consensus algorithms to reelect the *DA* nodes in charge of giving directory services to other nodes. Therefore, if a node *UA* has been chosen as a *DA*, this node should be reconfigured to match its new role. A pseudo code of the reconfiguration policy that guides the adaptation of the example is as follows:

```
if ( Elected-DA ) then
    reconfigure(UA,DA)
end
```

The reconfiguration policy (expressed in XML) associated with this reconfiguration is shown in Figure 7. Actually, the XML files of policies can be generated using the tool Genie.

Example 2: If a node *DA* has low battery and it was originally a node with the role *SA*, the node should be reconfigured to its original *SA* configuration. The same could happen if after the consensus algorithms to reelect the *DA* nodes another node is elected. The policy is as follows:

```
if(!Elected-DA ||(Low-Battery && RSA))then
    reconfigure(DA,UA)
end
```

The case study we used has necessarily been a simple one for reasons of space. In this sense, the transition diagram of the case study explained above just covers aspects associated with service discovery concerns using the multi-protocol component framework. However, aspects associated with networking issues can also be considered. In this case, two component frameworks would be associated with each structural variant in the transition diagram: the *Service Discovery* and the *Network* component frameworks, and the triggers specified in the arcs of the transition diagrams should also include properties and conditions associated with networking issues. Examples of situations that would be taken into account are (i) in a mobile application it is possible that a new network would come within range using a different technology, and, therefore, it may be necessary to reconfigure the *Network* component framework to satisfy the new network, or (ii) the battery life is in threat and, therefore, a *BlueTooth*-based networking might be preferred instead of a relatively power-hungry *WiFi*-based connection.

Other component frameworks that can be used in the specification of the structural variants of the transition diagrams are the *Spanning Tree* component framework (for the description of the topology of nodes in a network), the *Interaction* component framework (to choose between the different interaction types, e.g. publish-subscribe, group communication, peer-to-peer, data sharing and others), the *Security* and *Resource Management* frameworks. The inclusion of these component frameworks in the definition of the structural variants depends on the nature and concerns of the application to be developed.

Our approach has also been applied in the case of a real-world scenario: a wireless sensor flood forecasting application deployed on the River Ribble in the north west of England [21]. This case study includes adaptation concerns associated with the reconfiguration of the topology of the sensor networks and networking concerns using the *Spanning Tree* and *Network* component frameworks respectively. Some preliminary results are shown in [5, 18, 31].

5 Discussion: Contributions and Related Work

This section discusses the novel contributions of our research and contrasts the proposed approach with related work. Current approaches of system families (or product lines) base their support for variability on the configuration knowledge which is expressed explicitly when synthesizing a product (variant). This is enough in situations when the configuration is done statically. Traditionally, variability is meant to be solved at a predelivery moment [20]. In our case, the problem domain is in the field of customization of systems at runtime that noticeably takes place postdelivery. In this new field and as explained above not all the structural elements (such as components, configurations) or requirements are known at design time. It is not flexible enough to offer a fix set of variation points (hot spots) where different versions of components are replaced. With our approach sets of configurations are replaced in answer to context changes following the reconfiguration policies. Further more, new reconfiguration policies can be added at runtime changing dynamically the behaviour of the system. These policies are explicitly modelled in our approach what potentially improves the traceability during the software development process.

The proposed approach focuses on the structured managements of variation points that are bound at runtime. The dependencies between structural variability (architectural elements) and environment and context variability are made explicit. In a nutshell, the approach focuses on some of the (runtime) variability issues stated in [11, 16], such as no first-class representation of the concept of variability points, implicit dependencies, inflexible binding mechanisms, high

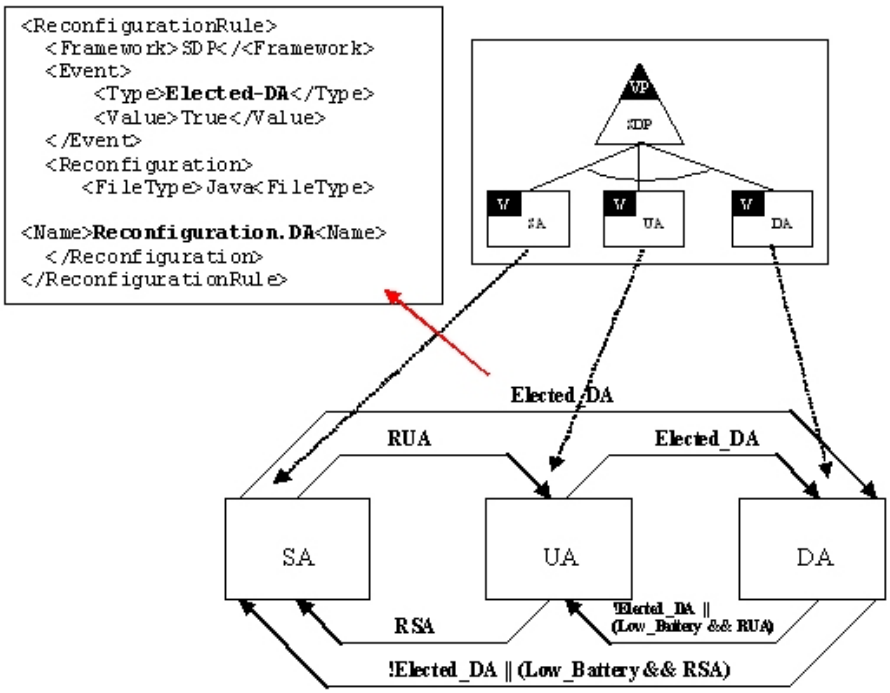

Figure 7. Reconfiguration Graph for the Variants SA, UA, and DA

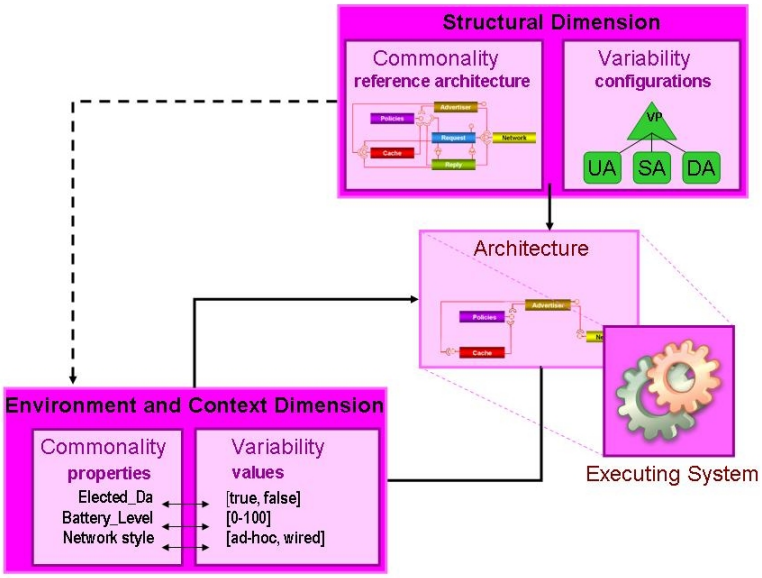

Figure 8. Dynamic Variability in the Service Discovery Protocols

resource costs, predictability, and addition of variants.

Many mechanisms for runtime variability management have been proposed. They are mainly focused on exchange of runtime entities, parametrization, inheritance for specialization, and preprocessor directives [16, 17, 29, 34]. Our approach proposes the management of whole sets of components, their connections and semantics (i.e. a more coarse grained approach). However, our approach is still

complementary to the finer grain styles cited above or in [38]. For example in each configuration traditional fine grain management of variability can be used to describe specific component replacements or specializations. Research work on MADAM [14] shares some of the principles of our approach as component frameworks to support variability. They also take into account the benefits of coarse-grained variability mechanisms. However our approach is more general as their focus is only on mobile computing applications. A similar research is found in [32]. They introduce the concept of composable components which is similar to our component frameworks. They apply recursive composition according to external requirements using ADLs what can be to some extent equivalent to our reconfiguration policies. However, they do not offer reflection capabilities, i.e their systems cannot reason about the current state or configuration of the system. Reflection offers support to determine where the points for variation are, what are the possible set of variations, or the state of the system at any point in time. However, using reflection has some drawbacks as the effect on performance and integrity issues. When developing reflective systems a trade-off between flexibility and performance has to be studied and a rigorous system development has to be performed.

In [18, 30, 31] we explain how the policy mechanisms contribute to providing a clear trace from user requirements to adaptation requirements [6] and their implementations. In this sense, the research related to requirements-driven composition in [32] is similar to our research.

6 Conclusions and Future Work

In this paper, we address how to manage effectively and in a structured way variation points that have to be bound at runtime. We focus on the development of systems that adapt to fluctuating environments following the established principles of systems families. The central elements of our approach are the concepts of component frameworks and reflection. The architecture offered by the component frameworks defines the invariant crucial for the reuse of common assets. The modeling approach proposed focusing on the explicit identification and documentation of the dynamic variability of the family.

A strong point of our approach is that it proposes a solution for the problem of unanticipated configurations and decisions that depend on the runtime context. It allows the specification of new reconfiguration policies, to discover and use new components and to vary the structural configuration (reconfiguration) of the system. Component frameworks and its specification, as presented in this paper, proposes the necessary flexibility, while offering formality to get the expected behaviour.

Substantial research remains to be done. For example, a concern is the combinatorial explosion related to the number of reconfiguration paths in the transition diagrams (i.e. the number of policy-based reconfigurations). However in the case study the number of reconfiguration paths is manageable, it might not be the case for other domains. We think that the combination of the specificity of *on-event-do-action* policies and higher-level policies that focus on general properties of the system can mitigate the problem.

Another concern is tool support for modeling variability and its integration into our Genie toolkit [33]. We have already some partial results shown in [5]. Tool support will help the scalability of the approach.

Acknowledgments We thank Paul Grace for his discussions on the above material.

References

[1] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355 – 398, 1992.

[2] N. Bencomo and G. Blair. Genie: a domain-specific modeling tool for the generation of adaptive and reflective middleware families. In *6th OOPSLA Workshop on Domain-Specific Modeling*, Portland, 2006.

[3] N. Bencomo, G. Blair, G. Coulson, and T. Batista. Towards a metamodelling approach to configurable middleware, 2005.

[4] N. Bencomo, P. Grace, and G. Blair. Models, runtime reflective mechanisms and family-based systems to support adaptation. In *Workshop on MOdel Driven Development for Middleware (MODDM)*, 2006.

[5] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair. Genie: Supporting the model driven development of reflective, component-based adaptive systems. *Submitted to ICSE 2008 - Research Demonstrations Track*, 2008.

[6] D. Berry, B. Cheng, and P. J. Zhang. The four levels of requirements engineering for and in dynamic adaptive systems. In *11th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'05)*, Porto, Portugal, 2005.

[7] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming. Special issue: Software variability management*, 53(3):333–352, 2004.

[8] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The design and implementation of open orb 2. *IEEE Distributed Systems Online*, 2(6), 2001.

[9] G. Blair, G. Coulson, and P. Grace. Research directions in reflective middleware: the lancaster experience. In *3rd Workshop on Reflective and Adaptive Middleware*, pages 262–267, 2004.

[10] G. Blair, G. Coulson, J. Ueyama, K. Lee, and A. Joolia. Opencom v2: A component model for building systems soft-

ware. In *IASTED Software Engineering and Applications*, USA, 2004.

[11] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obink, and K. Pohl. Variability issues in software product lines. In *4th International Workshop Software Product Family Engineering*, Bilbao, Spain, 2001.

[12] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[13] K. Dooley. Complex adaptive systems : A nominal definition. 1997.

[14] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *Software IEEE*, 23(2):62–70, 2006.

[15] C. A. Flores-Cortés, G. S. Blair, and P. Grace. An adaptive middleware to overcome service discovery heterogeneity in mobile ad-hoc environments. *IEEE Distributed Systems Online*, 2007.

[16] M. Goedicke, C. Köllmann, and U. Zdun. Designing runtime variation points in product line architectures: three cases. *Science of Computer Programming Special Issue: Software variability management*, 53(3):353 – 380, 2004.

[17] M. Goedicke, K. Pohl, and U. Zdun. Domain-specific runtime variability in product line architectures. In *8th International Conference on Object-Oriented. Information Systems*, pages 384 – 396, 2002.

[18] H. J. Goldsby, P. Sawyer, N. Bencomo, D. Hughes, and B. H. Cheng. Goal-based modeling of dynamically adaptive system requirements. In *15th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS)*, 2008.

[19] P. Grace, G. Blair, and S. Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *ACM SIGMOBILE Mobile Computing and Communications Review*, 9(1):2–14, 2005.

[20] J. V. Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Working IEEE/IFIP Conference on Software Architecture (WISCA'01)*, page 45, 2001.

[21] D. Hughes, P. Greenwood, G. Coulson, G. Blair, F. Pappenberger, P. Smith, and K. Beven. Gridstix:: Supporting flood prediction using embedded hardware and next generation grid middleware. In *4th International Workshop on Mobile Distributed Computing (MDC'06)*, Niagara Falls, USA, 2006.

[22] J. Lee and D. Muthig. Feature-oriented variability management in product line engineering. *Communications of the ACM*, 49(12), 2006.

[23] P. Maes. *Computional reflection*. PhD thesis, Vrije Universiteit, 1987.

[24] R. Marin-Perianu, P. Hartel, and H. Scholten. A classification of service discovery protocols. Technical Report TR-CTIT-05-25, University of Twente, 2005.

[25] C. Mascolo, L. Capra, and E. Wolfgang. Mobile computing middleware. *LNCS 2597*, pages 20–58, 2002.

[26] R. v. Ommering. *Building Product Populations with Software Components. PhD Thesis*. PhD thesis, Rijksuniversiteits Groningen, 2004.

[27] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications*, 14(3):54–62, 1999.

[28] K. Pohl, G. Böckle, and F. v. d. Linden. *Software Product Line Engineering- Foundations, Principles, and Techniques*. Springer, 2005.

[29] E. Posnak and G. Lavender. An adaptive framework for developing multimedia. *Communications ACM*, 40(10):43–47, 1997.

[30] P. Sawyer, N. Bencomo, P. Grace, and G. Blair. Handling multiple levels of requirements for middleware-supported adaptive systems. Technical Report COMP 001-2007, Lancaster University, 2007.

[31] P. Sawyer, N. Bencomo, P. Hughes, Danny andl Grace, H. J. Goldsby, and B. H. C. Cheng. Visualizing the analysis of dynamically adaptive systems using i* and dsls. In *REV'07: Second International Workshop on Requirements Engineering Visualization*, Delhi, India, 2007.

[32] I. Sora, V. Cretu, P. Verbaeten, and Y. Berbers. Managing variability of self-customizable systems through composable components. *Software Process: Improvement and Practice*, 10(1):77–95, 2005.

[33] SSE. Varmod-prime tool-environment, university of duisburg-esse. http://www.sse.uni-essen.de/wms/en/index.php?go=256, 2006.

[34] M. Svahnberg, J. v. Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705 – 754, 2005.

[35] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 2002.

[36] M. Trapp. *Modeling the Adaptation Behavior of Adaptive Embedded Systems. PhD Thesis*. PhD thesis, University of Kaiserslautern, 2005.

[37] M. Waldrop. *Complexity: The Emerging Science at the Edge of Chaos*. New York: Simon and Schuster, 1992.

[38] J. Zhang and B. H. Cheng. Model-based development of dynamically adaptive software. In *International Conference on Software Engineering (ICSE'06)*, China, 2006.