Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems*

Nelly Bencomo, Pete Sawyer, Gordon Blair, Paul Grace
Computing department, InfoLab21, Lancaster University, LA1 4WA, United Kingdom
email: {nelly, sawyer, gordon, gracep}@comp.lancs.ac.uk

Abstract

In this paper we propose an approach to support the design and operation of dynamically adaptive systems. We apply the concept of variability modeling from software product lines to define how systems adapt at runtime to changes in their environment. Our approach models two dynamic variability dimensions; environment variability, which defines the conditions under which a system must adapt, and structural variability which defines the resulting architectural configurations. The variability dimensions identified are modeled using domain-specific languages (DSMLs) tailored to adaptive middleware technologies that provide support for runtime variability according to reconfiguration policies. We describe our experience with applying this approach through a case study; the design of a flood warning system.

Keywords*: dynamic variability, architectural reconfiguration, orthogonal variability models, domain specific modeling languages, dynamic software product lines.*

1 Introduction

It is increasingly common for systems to be required to adapt dynamically to changes to their environment at runtime. One solution to this requirement is to use the adaptation mechanisms provided by reusable application-independent middleware components [6, 10, 17, 25]. Middleware platforms support the capability to reconfigure component compositions at run-time. In contrast to solutions where application behaviour and adaptation are entwined, using a middleware platform to enable adaptation separates the application-specific functionality from the adaptation concerns, thereby reducing complexity [22].

A dynamically adaptive system (DAS) developed using the support offered by a middleware platform can be conceptualized as a dynamic software product line (SPL) in which variabilities are bound at run-time. Each component-based configuration can be considered as a product or variant of the DAS. This means that variants of the DAS product line can be produced at runtime. The DAS product line defines a core reference architecture that constrains each product variant's component configuration.

In this context, the developer of a middleware-based DAS must deal with many variability decisions. Currently, developers of such systems have to make these decisions by reasoning at low levels of abstraction. To decide which components are required and how the components need to be configured and reconfigured in response to data about the state of the environment, DAS developers need detailed knowledge of the supporting middleware infrastructure. By contrast, architects and domain experts work at a much higher level of abstraction. Their job is to understand the characteristics of the DAS's operational environment and the domain concerns of the stakeholders. There is therefore a wide semantic gap between the way domain experts, system architects and programmers work in DAS development projects [1].

The concept of variability management originally developed for SPLs offers a means to bridge this semantic gap. Variability management refers to a repeatable, systematic approach to structure, implement and document the variability in a software family. Variability requires a scalable and consistent approach that exploits reuse independently from specific contexts. A number of researchers have applied SPL engineering techniques to the development of DASs [8, 17, 20, 29]. Our approach [2–4, 13, 26] provides variability management by systematically promoting software reuse and the use of models as first class entities. We raise the level of abstraction at which DAS developers can work by specifying structure and behaviour of a DAS using domain concepts and support for structured variability management.

We use domain-specific modelling languages (DSMLs) to model the solution architecture and the set of valid system configurations. Code and XML descriptions of component

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

configurations are automatically generated from these models. The reconfigurations in terms of components and the conditions that trigger the reconfigurations are also modeled and used to generate the reconfiguration policies that drive the system adaptations at run-time and which are supported by the middleware platforms.

The remainder of this paper is structured as follows. In section 2, we discuss dynamic variability and present two dimensions of variability in a DAS. Section 3 presents our model-driven design approach for DASs. In section 4, we illustrate our approach with an adaptive flood warning system built atop a wireless sensor network. Section 5 places our work in the context of related work. Section 6 concludes the paper with a summary and briefly highlights future work.

2 Adaptive Systems as Dynampic SPLs

At Lancaster University, we have developed a family of middleware platforms that use component-based technologies and the component model called OpenCOM [7]. Using OpenCOM both middleware and applications are conceived in form of components that follow the same model, i.e. the developers see no hard distinction between the application and the middleware platforms. OpenCOM offers the concept of component frameworks. A component framework is a collection of components that address a specific area of concern of middleware behaviour and offers a set of constrains. Component frameworks underpin the dynamic plug-in mechanism of components to change or add new behaviour at runtime. Adaptive behavior is specified using reconfiguration policies which are performed during the execution of the system. These policies are of the form *on-event-do-actions*. Actions consist of architectural changes that plug and unplug components into and from component frameworks. A context-aware engine monitors relevant environmental properties and events to identify the appropriate reconfiguration policy that holds. Crucially, the middleware platforms support DAS development by separating the concerns of the application functionality from the adaptation requirements.

Given the above, a DAS is described as a dynamic software product line. The variants of the DAS are defined by an n-tuple of component frameworks in which component configurations will be determined at runtime according to changes the environment of the DAS. Each DAS variant is called as structural variant. The component frameworks associated with each structural variant are established according to the domain problem.

GridStix Case Study

GridStix was a flood warning system deployed on a remote stretch of the River Ribble in North West England [18]. GridStix is supported by the Gridkit middleware platform [15], one of the members of the OpenCOM family of reflective middleware systems developed for grid applications. GridStix is used in this paper to demonstrate the proposed approach. This flooding warning system was conceived as a wireless network of sensor nodes able to perform local computation and communicate with other nodes. GridStix would therefore act as a lightweight computational grid, permitting distributed processing of sensor data and the execution of predictive models. These functionalities require heavy support for heterogeneous network technologies. However, a trade-off has to be done as, on the one hand, networking support must be power efficient to facilitate the operation of nodes for extended periods of time. On the other hand, applications such as image-based processing of pictures of the river for flow prediction also require high performance (and therefore power demanding) networking support. The above should also take into account varying resilience requirements. In that respect, during quiescent periods and when flooding is unlikely, data may reach the off-site cluster with an allowed high delay. During these quiet periods, low energy consumption is a major requirement to maximize the life-time of the nodes. By contrast, when a flood is imminent, the network should react promptly, while providing a high degree of resilience (e.g. a low sensitivity to disruptions), even if this means its energy supplies run down much more quickly.

GridStix has given us an opportunity to work with a real self-adaptive application. A possible set of reconfiguration opportunities (transitions) for this case study is shown in Figure 1. Three structural variants of the system were identified that correspond with the three possible states of the river Normal, Alert, and Emergency. Each structural variant is composed by two sets of component frameworks, one per domain of concern. These component frameworks are the Spanning Tree and the Network component framework that support the routing algorithm for data transmission between the nodes (domain 1) and the network technology (domain 2) to be used. The Spanning Tree component framework has associated two possible component configurations, the Shortest Path Spanning Tree and the Fewest Hop Spanning Three. The Network component framework has associated two possible component configurations that correspond with the WiFi and the Bluetooh technologies. According to events and properties of the environment different component configurations of these two frameworks will be chosen during execution.

As for the properties of the different component configurations to use, the Fewest Hop Spanning Tree offers better performance than the Shortest Path Spanning Tree; however the former is battery consuming. Similar situations are associated with the WiFi and the Bluetooh technologies as WiFi technology is more efficient than Bluetooh but also requires more energy. A trade-off between these properties

has to be done during the requirements specification, section 4.2 explains more details. The adaptations shown in Figure 1 are result of these trade-off decisions. We also show in that figure a representative pseudo-code for two reconfiguration policies (one per component framework) relating to the transition at the bottom.

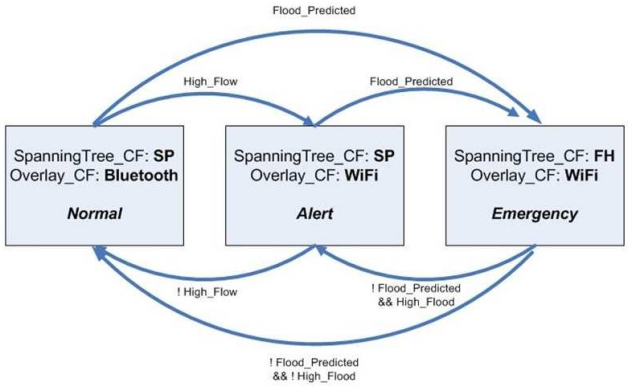

Figure 1. Transition diagram

The concepts derived from our middleware research that are relevant to this paper are depicted in Figure 2. Figure 2 is a UML class diagram that serves as a conceptual model to help explain the description that follows.

Component Frameworks are management units for sets of components that address specific solution Domains, such as routing algorithms, networking technologies and service discovery. A DAS will typically employ a number of Structural Variants which are sets of Component Configurations. The Component Configurations which managed by the Component Frameworks and its constrains will vary, with components being added, removed and replaced, as the DAS adapts to changes in its Environment. Every Component Configuration must conform to a Reference Architecture which describes the structural commonalities of a DAS that always hold. Effecting adaptation may require the ability to introspect about the system architecture at runtime. This capability is provided by reflection, with reconfiguration policies defining the adaptive behaviour. As said above, reconfiguration policies take the form *on-event-do-actions*. Actions are changes to component configurations while events in the environment are notified by a context engine.

Each system configuration can be considered as a product in a product-line in which the variability decisions necessary to instantiate the product are made at run-time. Figure 2 depicts this by assigning the roles SPL to the DAS and Product to the Structural Variant.

Structural Variability and Environment Variability are dimensions of dynamic variability (also called runtime variability) [2]. Both Environment and Structural Variability can be modeled as variation points as depicted in Figure 3 that uses the case study GridStix introduced above. Component Frameworks offer Structural Variability in response to Environment Variability as shown in (1) of Figure 3. In Environment Variability, the variation points represent properties of the environment. In Structural Variability, the variation points represent different configurations permissible within the constraints of the DAS's Reference Architecture. For the DAS to operate in the context of any Environment Variant, it needs to be configured as the appropriate Structural Variant, see (2) of Figure 3.

3 A model-driven approach for developing DSPLs

Even with the reusable solution-domain abstractions provided by component frameworks, DASs are complex to develop. Software engineering support for DASs has lagged the rate of advances in adaptive infrastructure, concentrating on support for downstream activities by developing, for example, language support and architectural description languages. In recent years, however, researchers have begun to develop upstream support by adopting concepts from SPLs, particularly feature modeling and variability modeling [8, 17, 20, 29].

The contribution of our work to software engineering support for DAS development is to use a model-driven approach to manage the two dimensions of dynamic variability. Models of the Structural Variability specify the architecture of the system that will evolve over time during system execution. A model of the Environment Variability specifies the conditions and events that will trigger changes in the architecture. The Structural Variants are the link between these two kind of models. The proposed models can be constructed using the Genie approach and associated tool [3, 4]. From Genie's models, component source code, component framework configurations, and reconfiguration policies [4] can be generated. Figure 4 illustrates the roles played by the middleware and Genie in the development process.

At level 3 in Figure 4 is the highest level addressed by Genie. This is where Genie uses the results of the requirements analysis [2, 13, 26] to model Environment Variability using the first of Genie's two domain specific modeling languages (DSMLs). Genie uses a Transition Diagram DSML in which each state represents a Structural Variant, and the transitions between these variants represent the adaptation scenarios. Each Structural Variant must satisfy both the global Goals of the DAS and the requirements that are specific to each Environment Variant. The Structural Variabil-

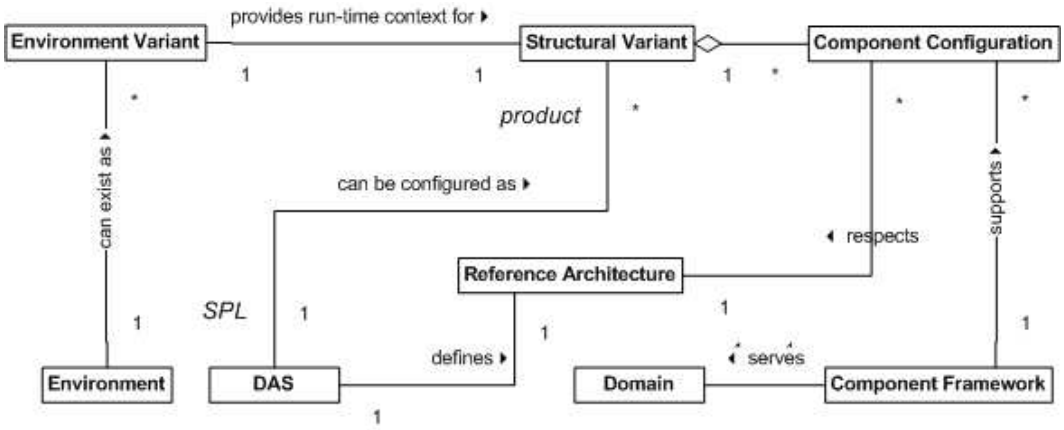

Figure 2. Conceptual Model

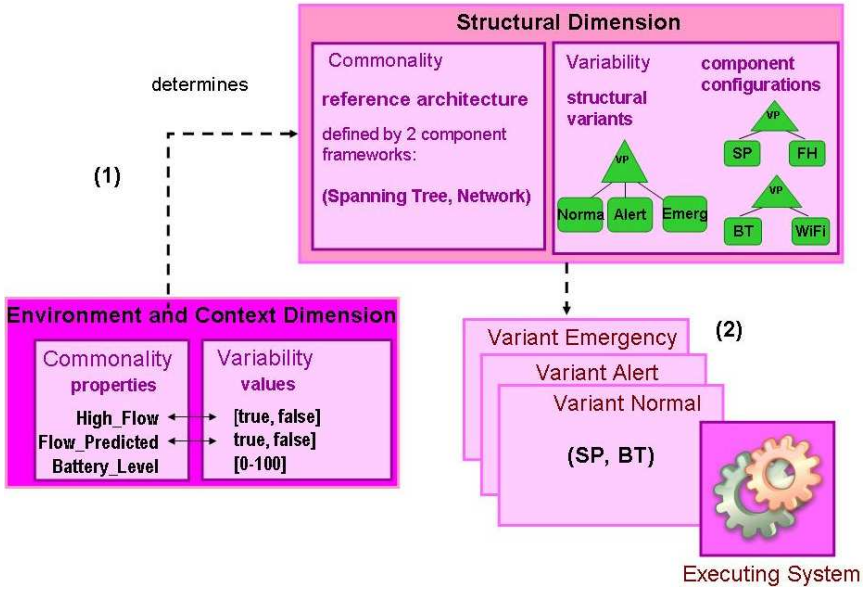

Figure 3. Dynamic variability dimensions for the flooding warning system

ity and the common Reference Architecture with which all Structural Variants must comply is provided by one or more Component Frameworks where each Component Framework addresses a particular solution domain (Figure 2).

Variability requires careful management and it is important that DAS developers are aware of the different variants they must deal with. To help promote this need, Genie supports the use of Orthogonal Variability models [23] to capture the environment and structural variabilities. An Orthogonal Variability model is developed to capture the Environment Variability and for the variabilities defined for each Component Framework. These models are mapped onto the Transition Diagram DSML and so make explicit the relationship between Environmental Variants and Structural Variants. Their use is further illustrated in the following section. Figure 5 shows a hypothetical example with three structural variants in a transition diagram. Each structural variant of the DAS is composed by two component frameworks associated with two domains of middleware behaviour (domains a and b).

At level 2 in Figure 4 the second Genie DSML, the OpenCOM DSML, is used to elaborate on the structural variabilities identified in the Transition Diagram DSML and the Orthogonal Variability models. The OpenCOM DSML is essentially an architectural description language (ADL) that, in our implementation of the Genie tool, is tailored to the OpenCOM family of reflective middleware systems [5, 6]. Each Structural Variant is defined in level 2. The Reference Architecture which each Structural Variant must respect is also defined here. The Reference Architecture is defined

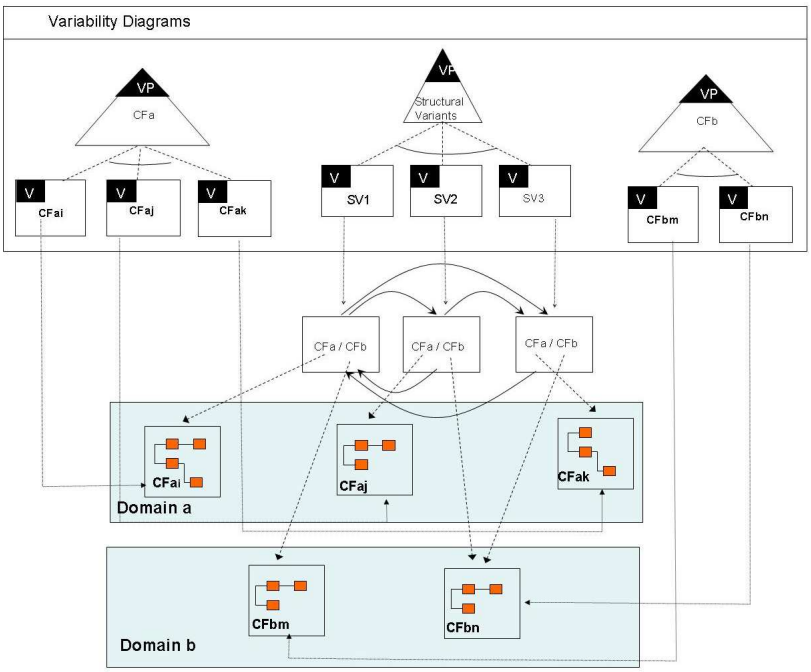

Figure 5. Structural Variants for two domains of middleware behaviour

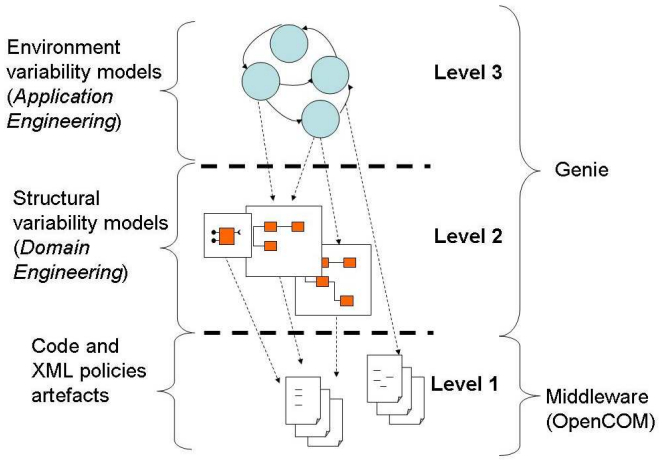

Figure 4. The development model

according to the Component Frameworks used and the variabilities are defined in terms of configurations of components.

Crucially, the modelling of Structural Variability is realized during the Domain Design in Domain Engineering. Likewise, the modelling of the Environment and Variability is mainly realized during the Design Analysis in Application Engineering [1].

The lowest level, level 1, in Figure 4 represents the mechanisms that define the configuration and reconfigurations of the components. The code and policy rules that drive these mechanisms are produced at level 2 using the OpenCOM DSML which has generative capabilities. The generality of the Genie model-based approach illustrated in Figure 4 increases from bottom to top. Levels 1 and 2 provide principled development and pattern-based reuse of OpenCOM-based applications. Level 3 is generic to any DAS for which the Environment can be conceptualized as a discrete set of Environment Variants.

In the next section we illustrate a full development lifecycle of a DAS that implements a flood warning system called GridStix [15, 18]. We show how Genie is used to model the variabilities. We describe the definition of the GridStix architecture and Component Configurations, and ultimately the generation of the policy rules and configuration code that implement GridStix within its run-time Environment.

4 Case Study: the GridStix Flood Warning System

4.1 Modelling Dynamic Variability

This section shows the application of our approach in the development and operation of the GridStix which was introduced in the last section.

The properties of the Transition Diagram for GridStix was derived from the LoREM requirements analysis of the envi-

ronment variabilities. Space does not permit us to describe it here but, LoREM and its application to GridStix is explained in [2, 13, 26]. An overview of the reconfiguration opportunities (transitions) that were identified during the requirements analysis for the case study is shown in Figure 1. The Transition Diagram developed with the Genie tool is shown in Figure 6.

Each state represents a structural variant intended to satisfy the requirements specific to one of the environment variants. The transitions between structural variants represent events derived from the requirements analysis of the adaptation scenarios. For example, the adaptation from the Alert variant to that for the Emergency variant occurs on the event FloodPredicted. FloodPredicted is an event generated by GridStix's predictive model that is based on Water Depth and Flow Rate measured by GridStix's sensors.

In addition to depicting the Transition diagram DSML, Figure 6 illustrates the the use of orthogonal variability modeling. Both the environmental and structural variabilities are modeled. In Figure 6 the central orthogonal variability model depicts the environment variants. The other two model the component frameworks that provided GridStix's structural variability. Each structural orthogonal variability model represents one of the component frameworks per domain needed to provide key GridStix functionality. The component frameworks used were selected from the set offered by the middleware platform Gridkit/OpenCOM.

Component frameworks provide architectural patterns and sets of components that can be configured within the constraints imposed by the architectural patterns. In this sense, component frameworks' architectural patterns represent reference architectures. Hence, development of component frameworks that support domains such as, for example, routing algorithms and networking technologies represents domain engineering. The available component frameworks thus support application engineering, of which GridStix's development was an example. The component frameworks can also be thought of as product families that embody commonalities represented by the architectural patterns, and variabilities represented by the possible component configurations. This is how they are represented by Genie's orthogonal variability models. The orthogonal variability models are mapped onto the structural variants in the transition diagram, making explicit the relationship between the environmental and structural variants.

Two component frameworks provide the pluggable components needed by GridStix: Spanning Tree and Network. We will focus on the Spanning Tree component framework. Here, we merely note that the other, the Network component framework, was used to allow GridStix to switch dynamically between different short-range wireless networking technologies according to requirements. The Spanning Tree component framework provides components to implement a virtual topology of the network for routing data between nodes. Hence, for example, a component configuration that supported the relatively energy-efficient Shortest Path Spanning Tree was required by the Normal and Alert variants, while a more fault-tolerant Fewest Hop Spanning Tree configuration was required for the Emergency variant where node failure was more likely.

The Transition diagram DSML defined a behavioural model of GridStix that incorporated an abstract representation of the structural variabilities. The role of the OpenCOM DSML was to explicate the structural variabilities by modeling them as sets of component configurations. A separate penCOM DSML model was developed to define the component compositions for each variant defined by the component frameworks. For the Spanning Tree framework, Shortest Path and Fewest Hop models were developed.

The Spanning Tree component framework defines a three-component architecture pattern. Hence both the Shortest Path and Fewest Hop OpenCOM DSML models provide components that perform the following three roles (which are part of the invariant) [15]:

i. the Control component that encapsulates the distributed algorithm used to build and maintain the virtual network topology.

ii. the Forwarding component that determines how messages are routed over the virtual topology.

iii. the State component that maintains and offers access to generic state and information, such as a nearest neighbour list.

Figure 7 illustates the OpenCOM DSML specification of the configuration of the Shortest Path Spanning Tree that is used to transmit data across the GridStix network in the Normal and Alert variants. The topology of the component configuration conforms to the Spanning Tree pattern, with components selected that implement each of the Control, Forwarding and State roles to provide a Shortest Path Spanning Tree. A similar model was developed for the Fewest Hop configuration, and also the Bluetooth and WiFi variants for the Network framework.

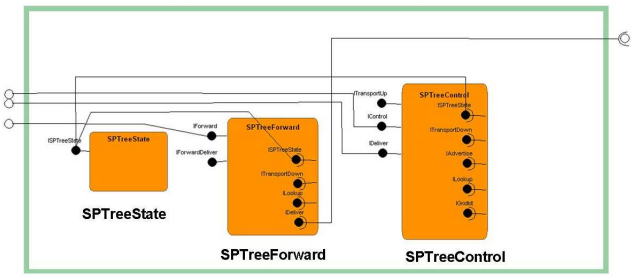

Figure 7. Shortest path spanning tree variant

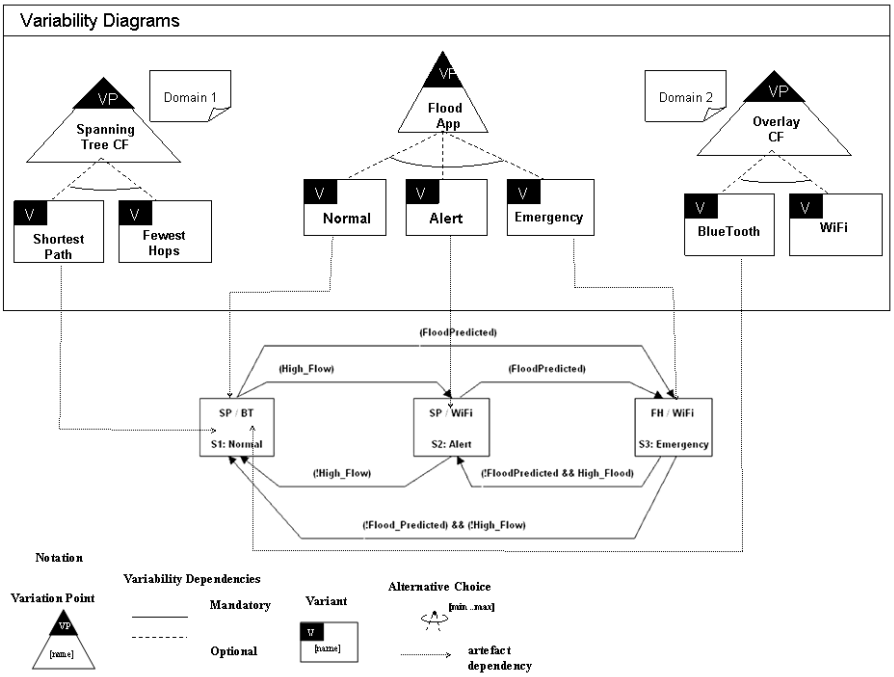

Figure 6. Transition diagram and the orthogonal variability models

From the configurations defined in the OpenCOM DSML and the events and transitions defined in the Transition diagram DSML, XML adaptation policies defining the reconfigurations of the the GridStix structural variants were generated using the Genie tool. The role of the orthogonal variability models here was essential to traverse the models to allow this generation. Thus defined, the GridKit runtime system managed the execution of GridStix, adapting GridStix as the environment fluctuated by adopting different component configurations. Figure 8 shows an example of one of the generated adaptation policies. Crucially, new adaptation policies can be added during runtime. This means that new reconfiguration diagrams can be designed and used to generate new adaptation policies to produce new adaptive behaviors.

To demonstrate the generality of the approach a further case study has been performed for a service discovery application and is described in [2, 4].

4.2 Discussion

The above case study has demonstrated the application and advantages of our approach to meet the challenges described in the introduction; specifically the wide semantic gap between the way domain experts, system architects and programmers work and the lack of structured management of variability. The following elaborates further and discusses how these challenges are addressed by our approach.

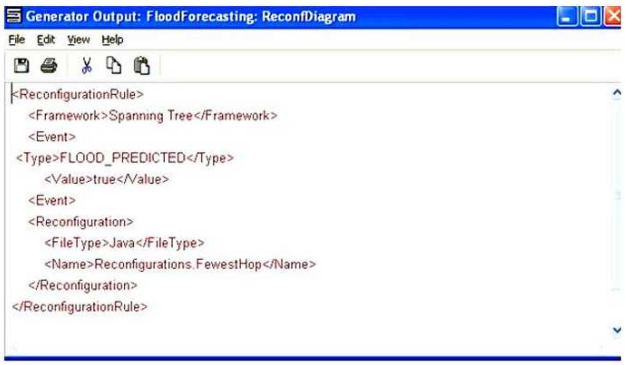

Figure 8. An example of a generated adaptation policy

Use of higher levels of abstraction to close the semantic gap between stakeholders

Without an approach like the solution supported by the Genie approach and its tool, developers work at the low-level of abstraction provided by the syntax of scripting languages or programming languages like Java or C++. Such programming languages do not convey either architecture-based design or domain semantics. The proposed approach, as seen in the case study, allows the developers to work

at a higher level of abstraction using models that specify the component frameworks with their configurations, and their reconfigurations. The use of models that show graphical representations of architectural concepts, as shown in the component configurations in Figure 7 reduce both effort and complexity during the lifecycle. Figure 7 shows just 3 components, 2 bindings, and 4 exported interfaces; however the generated XML file that describes the configuration has 102 lines. This XML file includes standard bindings (connections) to central components of the middleware platform that are included in the code that generates the XML file but are not shown in the model (they are included in the invariant part of the configuration). The complexity and effort needed when analyzing the configurations and their connections is lessened using the models provided. This is because these models are used to hide information that is not relevant for domain analysis and design.

However, it is not only the management of architectural (i.e. structural) aspects of configurations and components that can be realized with the DSMLs of the approach. We have demonstrated that specification of dynamic adaptive behaviour in terms of reconfigurations is also possible. With the approach, middleware developers use abstractions provided by the transition diagrams to reason, plan, and validate the reconfigurations. When the developer edits a transition diagram, she has an overview of all the reconfigurations that an application can go through. Transition diagrams offer an overall graphic-based view of the whole process of reconfigurations and are more amenable to analysis.

The transition diagrams described shows 3 structural variants and 6 possible reconfigurations (arcs) which represent 8 different reconfiguration policies. The reconfiguration policies are associated with two different component frameworks. The overall view offered by the transition diagrams described above contrasts with the partial text-based view offered by each reconfiguration policy. Using only partial views makes it very probable that the developers ignore, or simply lose sight of, important interdependency relationships. Overlooking dependencies can cause failures and inconsistencies during execution. Furthermore, identifying the source of the error may require significant effort and time. Another advantage of these models is the fact that transition diagrams models resemble state-transitions models which are a widely used and understood notations.

The use of transition diagrams models has given evidence of improving the interaction and communication between programmers, architects, requirements engineers, and domain experts. This is supported by the work on goal modelling to derive the requirements of the environment variability and required system behaviour using LoREM.

Structured management of variability

The models designed using the DSMLs offered by the approach support the definition of strategies for structured software reuse and variability management. The approach proposes variability notations and the associated models for the explicit management of the dynamic variability supported by the middleware platforms.

Such notations and models rely on the use of the component frameworks and the transition diagram models. The strategy proposed differentiates structural variability and environment or context variability. The architecture defined by the component frameworks basically describes the structural commonalities. Different configurations representing the variants of the component frameworks exist that follow the well-defined constraints imposed by their respective frameworks. Essentially, the environment and context variability models allow the specification of adaptations enabled by the middleware-based policy mechanism for reconfiguration. Using the two dimensions of dynamic variability that are proposed, the approach separates the application specific functionality offered by the component frameworks from the adaptation concerns, thereby reducing complexity ([22].

The proposed variability management structures document the variability in a standard and repeatable manner [19]. The proposed variability models have been applied in two case studies from very different domains (i.e service discovery in mobile computing scenarios and sensor networks as explained in this paper). Furthermore, the proposed use of the orthogonal variability models offers the potential to achieve traceability through the different models and resultant implementation artefacts, as well as the correspondence between the changes to the requirements and final behaviour of the system according to varying environmental conditions.

5 Related Work

Current approaches of 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 solved at predelivery [16] time. In our case, the customization needs to take place postdelivery; at runtime. Genie is at the heart of our approach. Genie supports the structured management of variation points that are bound at runtime. The dependencies between structural variability (architectural elements) and environment variability are made explicit. Genie is complemented by the LoREM goal-driven requirements approach. LoREM supports the formulation of the DAS's requirements, helping the analyst to understand the characteristics of the DAS's operational environment and the adaptation scenarios [2, 13, 26]. The goal models are used to derive the DSML models used in Genie. At the bottom, the middleware platform underpins the reorganization of the ongoing

architecture at run-time providing particular system support as the requirements imposed by the environment change.

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 [11, 12, 24, 27]. Our approach is more coarse-grained and uses the management of whole sets of components, their connections and semantics. However, our approach is complementary to the finer grain styles cited above. For example in each configuration, traditional fine-grained management of variability can be used to describe specific component replacements or specializations.

Of particular relevance to our work is MADAM [9, 17] which uses the adaptation capabilities offered by a middleware platform, and treats DASs as software product lines [17]. MADAM also takes into account the benefits of coarse-grained variability mechanisms, sharing some of the principles of our approach to support variability. In the MADAM approach, variants are treated as configurations, not simply components, in the same way that component frameworks support variants in Genie and OpenCOM. MADAM also uses the configurator pattern for event-based reconfiguration. Despite the similarities with MADAM, our research differs in several significant ways. In the case of the reconfiguration approach supported at the middleware level during execution, [17] has a common reconfiguration pattern based upon the context/utility functions. In our approach, we model reconfiguration explicitly using reconfiguration policies. Our approach is also more general since the focus of MADAM is restricted to mobile computing applications, which Genie can also support [2].

Wolfinger et al. [29] demonstrate the benefits of the integration of an existing product line engineering tool suite with a plug-in platform for enterprise software. As in our case, automatic runtime adaptation and reconfiguration are achieved by using the knowledge documented in variability models. Our differences exist mainly because of the different aims of each approach. Their work focuses on enterprise software while our work covers the domains grid and mobile computing, and embedded systems. While variability decisions in [29] are user-centered our variability decisions are environment-centered.

When performing dynamic reconfiguration we ensure that updates complete atomically and do not impact the integrity of the network. To do this, frameworks are placed in a quiescent state ensuring that the reconfiguration is complete and correct. In this sense, we are investigating the use of architectural patterns to drive the generation of software artefacts related to safe reconfiguration at that level. In this sense the work presented by Gomaa and Hussein [14] is relevant and complementary to our research.

6 Conclusions and Future Work

We have proposed combining the Genie model-driven development approach [2, 4], and tool, and middleware platforms to support the development and operation of dynamically adaptive systems from requirements to implementation. We argue that a DAS can be regarded as a product family line in which variabilities are bound at runtime instead of at pre-delivery time. Key to our approach is the recognition and separation of environment and structural (dynamic) variability. Here, environment variability offers a way to conceptualize and reason about how the system's environment will change during system execution and the implications of this variability for how the system must adapt.

The role of Genie is to develop from a definition of a DAS's environment variabilities and triggers for adaptation, a design that satisfies the requirements and is capable of adapting to the environment variabilities. Genie supports systems capable of performing online-determined reconfiguration. The Genie tool [4] is tailored to one example of such a software infrastructure: Gridkit , which is itself a member of the OpenCOM family of middleware systems. This permits the Genie tool to exploit long-standing domain engineering efforts that have produced Gridkit's library of reusable domain-specific component frameworks, using two complementary domain-specific modeling languages. It also permits Genie to perform code generation in the form of policy rules that define the configurations and reconfigurations of a system in order to implement the structural variants and triggers for their adaptation.

We have validated our approach using the GridStix case study [15, 18]. GridStix is a wireless sensor network for flood prediction that has been deployed on the River Ribble in North West England.

To date the Genie approach is supported by the Genie tool. We aim to further systematize the integration of requirements analysis given by LoREM and Genie to help extend Genie's model-driven life-cycle coverage to higher levels of abstraction to encompass requirements. Traceability will also be aided if we are successful. We are also working on RELAX [28], our new proposed language for the requirements specification of self-adaptive systems. The RELAX philosophy and its vocabulary explicitly acknowledges the need to deal with the levels of uncertainty, which are unavoidable when introducing self-adaptation capabilities to systems. The model-driven approach, variability management, and runtime support offered by Genie and the middleware platforms offer an implementation context for the self-adaptive systems specified using RELAX.

A further open research issue is concerned with the scalability of our approach. There is a potential for a combinatorial explosion of the number of reconfiguration paths

in the transition diagrams policy-based reconfigurations. In the GridStix case study the number of reconfiguration paths is manageable, but this might not be the case for other domains. Our partial results on this topic are in [21].

References

[1] N. Bencomo. *Supporting the Modelling and Generation of Reflective Middleware Families and Applications using Dynamic Variability, PhD Thesis*. PhD thesis, 2008.

[2] N. Bencomo, G. Blair, C. Flores, and P. Sawyer. Reflective component-based technologies to support dynamic variability. In *2nd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'08)*, 2008.

[3] 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.

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

[5] 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.

[6] 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.

[7] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. A component model for building systems software. In *Software Engineering and Applications (SEA'04)*, USA, 2004.

[8] D. Dhungana, P. Grunbacher, and R. Rabiser. Domain-specific adaptations of product line variability modeling. In *IFIP Working Conference on Situational Method Engineering: Fundamentals and Experiences*, Geneva, 2007.

[9] 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.

[10] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10), 2004.

[11] 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.

[12] 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.

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

[14] H. Gomaa and M. Hussein. Model-based software design and adaptation. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS'07)*, 2007.

[15] P. Grace, D. Hughes, B. Porter, G. Blair, G. Coulson, and F. Taiani. Experiences with open overlays: A middleware approach to network heterogeneity. In *Proc. 3rd ACM International EuroSys Conference*, Glasgow, Scotland, 2008.

[16] 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)*, 2001.

[17] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using product line techniques to build adaptive systems. In *SPLC '06: Proceedings of the 10th International on Software Product Line Conference*, pages 141–150, Washington, DC, USA, 2006. IEEE Computer Society.

[18] D. Hughes, P. Greenwood, G. Coulson, G. Blair, F. Pappenberger, P. Smith, and K. Beven. An intelligent and adaptable flood monitoring and warning system. In *5th UK E-Science All Hands Meeting (AHM06) (Best Paper Award)*, 2006.

[19] M. Jaring. *Variability Engineering as an Integral Part of the Software Product Family Development Process. PhD Thesis*. PhD thesis, University of Groningen, 2005.

[20] M. Kim, J. Jeong, and S. Park. From product lines to self-managed systems: an architecture-based runtime reconfiguration framework. In *Workshop on Design and Evolution of Autonomic Application Software*, 2005.

[21] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jezequel, A. Solberg, V. Dehlen, and G. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *MODELS'08 Conference*, France, 2008.

[22] 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.

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

[24] E. Posnak and G. Lavender. An adaptive framework for developing multimedia. *Communications ACM*, 1997.

[25] M. Roman, F. Kon, and R. H. Campbell. Reflective middleware: From the desk to your hand. *IEEE DS Online, Special Issue on Reflective Middleware*, 2(2), 2001.

[26] 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*, India, 2007.

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

[28] J. Whittle, P. Sawyer, N. Bencomo, and B. Cheng. A language for requirements engineering of self-adaptive systems. In *SOCCER Workshop*, 2008.

[29] R. Wolfinger, S. Reiter, D. Dhungana, P. Grunbacher, and H. Prahofer. Supporting runtime system adaptation through product line engineering and plug-in techniques. In *ICCBSS*, pages 21 – 30, 2008.