Using Architecture Models to Support the Generation and Operation of Component-based Adaptive Systems ⋆

Nelly Bencomo and Gordon Blair

Computing Department, InfoLab21,
Lancaster University, LA1 4WA, United Kingdom

Abstract. Modelling architectural information is particularly important because of the acknowledged crucial role of software architecture in raising the level of abstraction during development. In the MDE area, the level of abstraction of models has frequently been related to low-level design concepts. However, model-driven techniques can be further exploited to model software artefacts that take into account the architecture of the system and its changes according to variations of the environment. In this paper, we propose model-driven techniques and dynamic variability as concepts useful for modelling the dynamic fluctuation of the environment and its impact on the architecture. Using the mappings from the models to implementation, generative techniques allow the (semi) automatic generation of artefacts making the process more efficient and promoting software reuse. The automatic generation of configurations and reconfigurations from models provides the basis for safer execution. The architectural perspective offered by the models shift focus away from implementation details to the whole view of the system and its runtime change promoting high-level analysis.
Keywords: software architecture, dynamic adaptation, model-driven engineering, middleware, dynamic variability.

1 Introduction

Adaptability is an increasingly important requirement for many applications, in particular those deployed in dynamically changing environments such as environmental monitoring and disaster management [29, 15]. A well established approach to enable adaptation is to use the support of middleware platforms [8, 24, 9]. Such configurable and reconfigurable middleware platforms are ideally situated to monitor changes, and manage individual adaptations. The primary role of middleware platforms is to ease the development and operation of distributed applications. The approach allows the application developers and domain experts to focus on the application logic, rather than complex runtime, adaptation concerns that the middleware platforms deal with.

Making simpler the development of distributed applications by loading the middleware platform with distribution and runtime concerns causes that development of these platforms require highly technical knowledgable developers. Therefore, middleware developers need to work at very low levels of abstraction. Hence, the development

⋆ This work was partially funded by the Divergent Grid project, an ESPRC funded project and the DiVA project (EU FP7 STREP).

of components and planning of configurations and reconfigurations involve a large number of variability decisions related to fluctuations of the environment. These decisions are frequently implemented using programming environments and tools with low levels of abstraction (i.e. using constructions offered by programming languages like C++ and Java). The above results in a gap between the way application developers, domain experts and middleware developers operate. Furthermore, the development process frequently uses repeated ad-hoc solutions that many times are carried out manually.

The above reveals the need for both new software development approaches and operational paradigms. These approaches and paradigms should be able to (i) promote the overall view of the system that domain experts and application developers need when planning the adaptation logic, and at the same time (ii) deal with the levels of detailed and technical knowledge required by middleware developers, (iii) bridge the gap in between the different levels of abstraction of (i) and (ii), and (iv) be more systematic and efficient, exploiting software reuse whenever possible. The approach presented in this paper is aiming to address these issues.

Modelling architectural information is particularly important in this context because of the crucial role of software architecture in raising the level of abstraction during development. Such a fundamental role is repeatedly emphasized by the numerous definitions of software architecture [26, 11, 27, 17, 2]. For example, according to Oreizy et all [33] "*a software architecture represents software system structure at a high level of abstraction, and in a form that makes it amenable to analysis, refinement, and other engineering concerns*". This definition is particularly relevant because it also highlights the opportunities for high-level analysis provided by architectural descriptions [17]. The importance of architecture has been recently revisited in the Future of Software Engineering session at ICSE 2007 [38, 25].

The authors argue that MDE and generative software development [13] help to produce new development paradigms to support the life cycle of flexible and dynamically configurable middleware platforms. In the MDE area, research has focused mainly on using models during the phases before execution (i.e. design, implementation and deployment) with emphasis on the generation of software artefacts to be used in those phases (e.g. source code or deployment descriptors) [20, 1]. Moreover, abstractions used for model-based transformations have frequently been related to low level design concepts. Abstractions can also be related to the support for dynamic management and evolution of software [16]. In this sense, model-driven techniques can be used to model software artefacts that take into account the architecture (i.e. high-level structural organization) of the system and its changes according to the fluctuation of the environment.

We propose an approach called Genie. Genie uses domain-specific modelling and dynamic variability (i.e. variability that needs to be solved at runtime) as relevant concepts for the construction of models of the dynamic fluctuation of the environment and contexts, and their impact on the variation of the architecture of the middleware and applications during execution. Genie offers management of dynamic variability during development and allows the systematic generation of middleware related artefacts from high level descriptions (models). To this end, two kinds of dynamic variability are identified, namely *structural variability* and *environment and context variability*. Using the mappings from the models to implementation artefacts, generative techniques

will allow the (semi) automatic generation of implementation artefacts making the process more efficient and promoting software reuse. The authors argue that generating the code associated with configurations and reconfigurations directly from the models provides the basis for defining safer execution by reducing coding errors. The architectural view offered by the models improves high-level analysis shifting focus away from implementation details to the whole view of the system and its runtime change.

The remainder of this paper concentrates on the conception, design, and application of the approach proposed. Section 2 discusses the case of dynamic variability for adaptive systems. Section 3 describes the Genie approach in detail. Section 4 discusses the application of the approach in a specific real case study, the development and operation of an adaptive flood warning system. Finally, Sections 6 and 7 present some related work and conclusions respectively.

2 Dynamic Variability

2.1 Overview

One of the reasons for software variability is to delay design decisions [37]. 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* [13]. Later on, during *application engineering*, specific systems are developed to satisfy the requirements 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* [37]. Eventually, one of the variants should be chosen to be achieved or implemented. The time when it is done is called *binding time*. Traditionally, decisions have been deferred to architecture design, implementation, compilation, linking, and deployment as shown in [37, 31, 7, 28, 13]. 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 their impact in the ongoing architecture of the system and the management of variabilities in dynamically adaptive systems. These challenges are further discussed in the next section.

2.2 Dynamic Variability in Adaptive Systems

A dynamically adaptive system operates in an environment that imposes changing contexts and requirements. The challenge comes from the need to support adaptation or customization of the system according to the needs of the fluctuating environment. The conditions associated with the adaptations may not be completely known before installation of the system. These conditions are related to:

(i) Environment or context variability: the evolution of the environment often cannot be completely predicted during development; therefore the total range of contexts and requirements may be unknown before the system is installed to start execution.

(ii) Structural or architectural variability: this covers the variety of components and the variety of their configurations (architecture). This is a consequence of the environmental variability explained above. In order to satisfy the set of requirements for the

new context, the running system may dynamically add new components or rearrange the current structural configuration (architectural reorganization). Hence, solutions cannot be restricted to a set of known-in-advance configurations and components. New sets of components may be added during execution (see Figure 1).

The system should be prepared to deal with the two dimensions of variability described above. Under 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 [36] and according to some quality of service (QoS) properties. 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 (architecture). Moreover, the correct match between the architectural changes and the environmental context should be maintained.

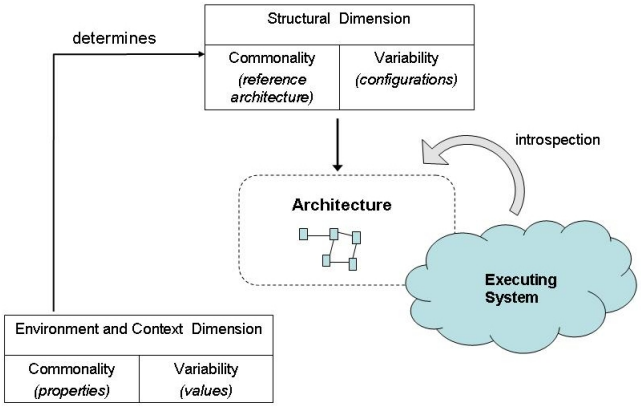

Fig. 1. Dynamic Variability Dimensions

2.3 Architectural Reorganization supported by Middleware Platforms

At Lancaster University, we have gained experience developing adaptive systems and middleware platforms using *component frameworks* and *reflective technologies* [12]. We use of component frameworks and reflection as flexible mechanisms for supporting runtime variability of dynamically extensible systems. Component frameworks are collections of components that address a specific area of concern and accept "plug-in" components that add or extend behaviour [12]. 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 policies are of the form *on-event-do-actions* and *actions* are architectural changes using the component frameworks. During runtime, 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 architectural variability. Furthermore, reflective capabilities offer the potential to reason about the possible variation points and their variants during execution. The support offered by the middleware platforms provides the technique to implement variability called ***infrastructure-centered architecture*** [37]. When using this technique, connections between components are treated as first-class entities. This means that required interfaces of components are not hard-coded. Dynamic replacement of a component in the architecture or indeed dynamic reorganization of the architecture is eased if the architecture and the location where such modifications could be carried out is made explicit. "*Used correctly, this technique yields perhaps the most dynamic of all architectures*" [37]. The next section elaborates how the middleware platforms act as the reconfiguration framework and introduces the approach proposed. The approach leverages the middleware platforms to guide the domain experts and developers during the modelling and generation of software artefacts, and during the operation of middleware platforms and applications. This is a key assumption that the underlying middleware platform ensures consistency and integrity using change transactions [12].

3 The Genie Approach

3.1 Overview

The proposed approach is called Genie. Genie uses domain specific languages (DSLs) for the construction of the models associated with both the structural (architectural) and the environment variability. Using models and generative techniques, software artefacts can be generated more efficiently. Supported by the middleware platforms, applications can be dynamically reconfigured from one structural *variant* to another according to changes in the context or environment. The system monitors specific properties of the runtime environment and reacts to given changes while keeping a valid architecture. The system is able to decide what kind of architectural reorganization (reconfiguration) has to be performed, if any.

To model the adaptive behaviour described above it is necessary to define what adaptation means in terms of configurations (architecture) and conditions:

*An **adaptation** is defined in the scope of this research as the process of having the system transforming itself from a given configuration **Ci** to another configuration **Cj** given the set of conditions **Tk**.*

The set of conditions correspond to variants of the context and environment variability and the configurations (components and connections) correspond to variants dictated by the architectural and structural variability. The next section describes the proposed approach.

3.2 Description of the Genie Approach

The Genie approach allows the use of DSLs to specify:

i. **the structural variability**. The DSL associated with the structural variability allows the modelling of the component configurations to be expressed in terms of

the architecture dictated by component frameworks. The modelling elements to be used are generic architectural elements such as components, required and offered interfaces, and bindings.

ii. **the environment and context variability**. The *transition diagram* DSL is used to specify the conditions that represent the dynamic nature of the environment and context. Basically, this DSL is used to specify adaptations of the form described above: from the configuration *Ci* and on the set of conditions *Tk*, go to configuration *Cj*. These models are in essence *transition diagrams*.

Using generators capable of traversing the models created with the DSLs and their trace relationships, different software artefacts can be generated:

iii. **components and configurations of components** associated with the component frameworks are generated from the structural variability models. The constraints specified by the component frameworks are captured in the models to allow validation of the configurations and ensure consistency of the resultant artefacts. The middleware platforms allow the newly generated components and component configurations to be added during the execution of the system.

iv. **reconfiguration policies** are generated from the transition models. As in (iii), validation of the diagrams should be performed to avoid inconsistencies. The middleware platforms allow the generated policies to be inserted during execution. The newly added reconfiguration policies are used as long as the "new" component(s) or component configuration(s) for the right match are also provided.

3.3 Levels of Abstraction

An overview of the different levels of abstraction promoted by Genie is given in Figure 2. The figure shows the specific artefacts that populate the layers which correspond to different levels of abstraction (abstraction levels are raised from bottom to top).

(1) The first level at the bottom is populated by different software artefacts like source code, XML configuration files describing the different configurations associated with component frameworks, and the XML files of reconfiguration policies.

(2) The second level corresponds to the architectural models associated with *structural variability*, i.e. the models of component frameworks and their components and configurations. These models offer visual representations of the component configurations, their components and interfaces.

(3) The third level at the top corresponds to the *environment and context variability*. Here the developer plans the adaptations based on transition diagrams. At this level the developer reasons in terms of structural variants (associated with specific domains of concern) and conditions of the environment that trigger the reconfigurations.

The first and second levels are similar to existing approaches using architecture description languages (ADLs) and offering tool chain support for the development of component-based systems. Actually, the DSL associated with the structural variability can be considered as an ADL with generative capabilities. The major contribution of the Genie approach is the support to high level analysis and automation provided by level 3 and its relationships with the other two levels.

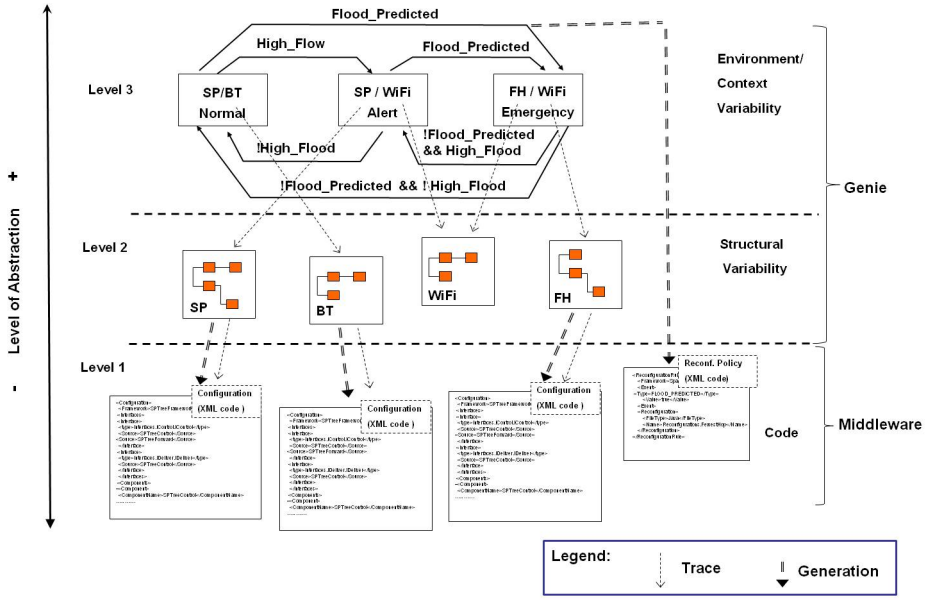

Fig. 2. Different levels of abstraction supported by Genie

Each node in the transition diagrams of level 3 is considered as a *structural variant* of the system. Structural variants are "*coarser grain*" configurations than configurations associated with individual component frameworks in the sense that they are described by a set (or n-tuple) of component frameworks. Structural variants can be seen as configurations of component frameworks. The set of component frameworks are associated with the problem domain. Thus, for example, if the problem domain identified requires architectural changes (in terms of reconfiguration) of the routing protocols and the topology of nodes in a sensor network, the component frameworks to be used in each structural variant should represent concepts associated with routing protocols and topologies of nodes. The proposed approach aims then at partitioning each structural variant into a set of specialized and focused domains of concern.

Figure 2 shows the *trace* relationships between the levels. At the top, each structural variant has the references to both the related component frameworks in level 2 and the files of reconfiguration policies in level 1. The component frameworks in level 2 make reference to the files associated with the configuration files. In turn, the reconfiguration files point to the executable code associated with the components. In the example of Figure 2, each structural variant of the transition diagram is described in terms of two component frameworks (the *Spanning Tree* component framework for routing protocols and the *Network* component framework) that opportunistically correspond to the case study described in the next section. From the initial architecture (configuration) the system will evolve over time according to the conditions of the environment specified in the arcs of the diagrams. The places where the architecture can be changed and the consequences of the changes will be driven by the transition diagrams.

The use of DSLs in the approach described above promotes higher levels of abstraction beyond programming and code. The benefits from raising the levels of abstraction using models is twofold. First, automation levels are improved as the models allow the specification and application of repetitive patterns that are used in the generation of software artefacts. As a result software reuse is encouraged. Second, the gap between the way requirements engineers, domain experts, software architects and programmers operate is reduced, thereby promoting their joint collaboration as shown in [21]. Furthermore, the use of generative techniques increases the levels of efficiency and automation. The approach can be applied using different middleware platforms that work with concepts of components and component frameworks like in the case of OpenCOM or Fractal as reported in [3] and [30].

Next section discusses the application of the approach in the specific case of the development and operation of an adaptive flood warning system [23]. The approach has also been applied in the context of dynamic service discovery scenarios for mobile applications with results reported in [3].

4 Case Study: a wireless sensor network for flood management

4.1 Overview

The *Genie tool* is the implementation of the Genie approach for the case of the OpenCOM based middleware platforms at Lancaster University and is described in detail in [3, 5]. The DSL associated with the structural variability is called the OpenCOM DSL in the case of the *Genie tool*.

The use of the architectural models supported by Genie and its tool is explained using the case study GridStix [23]. GridStix is a wireless sensor network for flood management that has been deployed in prototype form on the flood plain of the River Ribble in North Yorkshire, England. About 15 nodes have been deployed. Sensors route the data collected in real-time using a spanning tree topology to one or more designated *root nodes*. From these nodes, the data is forwarded (via General Packet Radio Service, GPRS) to a *prediction model* that runs on a remote computational cluster. Each sensor node includes a 400MHz XScale CPU, 64MB of RAM, 16MB of flash memory, and Bluetooth and WiFi Networks. The designated root nodes are also equipped with GPRS. Each GridStix is powered by a 4 watt solar array and a 12V 10Ah battery. Linux 2.6 and the Java virtual machine 1.4 are used in contrast to conventional sensor network deployments, where sensors are simply responsible for transmitting sensor data off-site. This deployment permits the use of local processing. Local processing supports computation for the local prediction of future environmental conditions.

Level 3 of Figure 2 shows the transition diagram that guides the reconfiguration and adaptation process of GridStix. Three possible states were identified: *Normal*, *Alert*, and *Emergency*. Each state of the system has a specific structural (architectural) variant. The problem domain identified requires structural changes (in terms of reconfiguration) of the routing algorithms and the networks interfaces to be used in the sensor network. The component frameworks to be used in each *structural variant* represent concepts associated with the overlays component framework, specifically the *Spanning Tree* and the *Network* framework.

The *Spanning Tree* component framework supports the routing algorithm that has two possible variants: *Shortest Path* (SP) and *Fewest Hop* (FH). The *Spanning Tree* component framework and its variants follow the overlay pattern architecture (control, forward, and state). The overlays pattern is shown in Figure 3. This is part of a more generic pattern that allows the overlay of individual plug-ins to be inserted into component frameworks. As such, the reuse of architecture design is one of its main contributions. The use of models to generate artefacts further exploits this contribution. The second one, the *Network* component framework, describes the type of network to be used and offers two possible variants: *BlueTooth*(BT) and *WiFi*.

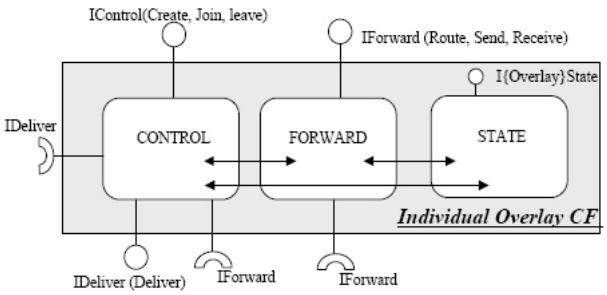

Fig. 3. Overlays Pattern Architecture

The 2-tuples associated with the structural variants used in the case study are (*SP,BT*), (*SP,WiFi*), and (*FH,WiFi*) and correspond with the three possible states identified above. Figures 4 and 5 show the *Shortest Path* variant and *WiFi* variant models developed with the *Genie tool* respectively.

The system will evolve over time according to changes in the conditions of the environment specified in the arcs of the diagrams. The places where the architecture can be changed and the consequences of the changes will be driven by the transition diagrams. How different variants of these component frameworks are chosen will depend on the possible multiple variations of conditions in the environment and context. This variation is specified using the triggers associated with the transitions in the diagram (i.e. arcs). Triggers of reconfiguration policies are specified in the arcs between states. The number of transitions in the transition diagrams will depend on how adaptable the system should be or is conceived.

According to the transition diagram in Figure 6, if the application is operating as *Normal*, and the prediction model of GridStix predicts an imminent flood (i.e. the *FloodPredicted* monitoring condition is true), the nodes adapt to the *Emergency* state bypassing the *Alert* state. This adaptation is effected by reconfiguring the *Network* to use *WiFi* instead of *BlueTooth*, and the *Spanning Tree* to a *Fewest Hop* topology.

One of the advantages of using *transition diagram* models is that they offer a complete view of the reconfiguration opportunities of the system offered to the user. The architectural perspective offered by these models shift focus away from the source code of isolated policies to the whole view of the reconfiguration opportunities and

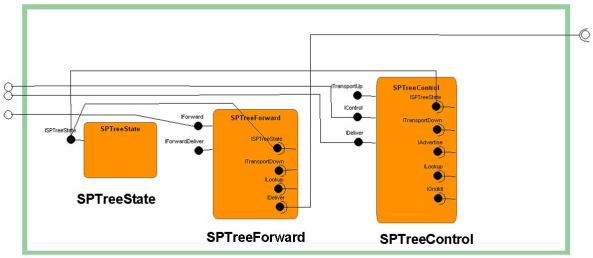

Fig. 4. Shortest Path Variant for the Spanning Three component framework

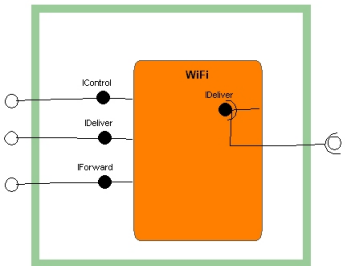

Fig. 5. WiFi Variant for the Network component framework

the component frameworks involved. Different stakeholders can abstract away irrelevant implementation-related details and focus on the big picture: the system structure and its runtime change. It should be contrasted with the partial view when working with individual policies using traditional approaches. Figure 6 shows 3 examples of the reconfiguration policies that are invoked when the specified monitoring conditions are met. From this specific example, a total of 8 reconfiguration policies can be generated. For readability purposes, the transition diagram proposed is simple, as the number of policies increases rapidly with the number of triggers considered.

4.2 Orthogonal Variability Models

To complement the approach described above, the orthogonal variability models proposed by *Klaus Pohl et all* [34] are used. An orthogonal variability model (OVM) defines the variability of a system family in a separate model. It relates the variability specified to other software development models such as component models in our case. Figure 7 shows the variability diagrams used to model the variants in the case study. The three structural variants, *Normal*, *Alert*, and *Emergency* are associated with the variation point *VP:Flood App* marked by (a). Each state variant of the graph is described using two component frameworks, i.e. the *Spanning Tree* and the *Network* component framework as seen above. The *Spanning Tree* and the *Network* component frameworks have variation points associated themselves, marked by (b) and (c).

The OVM has mainly been used by developers to document variability. However, in our work these variability models have been useful not just to document variability.

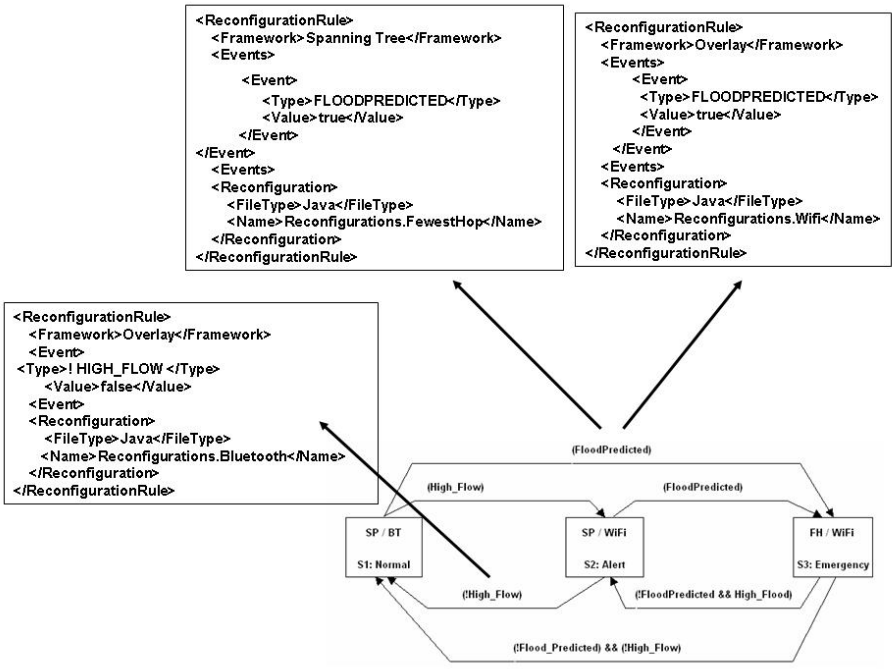

Fig. 6. The transition diagram of the case study and three generated reconfiguration policies

Particularly, OVMs have proofed to be useful (i) when traversing the models to generate the reconfiguration policies, (ii) to keep links to adaptation requirements (using goal models) [21], and (iii) when managing the traceability relations between the structural variants of the transition diagrams (level 3) and the component frameworks configurations (level 2).

4.3 Artefacts Generated by the *Genie tool*

The developer designs models to specify the components, component frameworks and configurations, structural variants and the transition diagrams using the DSLs provided by the *Genie tool*. Using generators that traverse these models, different software artefacts of level 1 can be generated (see Figure 8).

From the models specified using the OpenCOM DSL the source code of components and configurations of components associated with the component frameworks is generated. Similarly, using the models associated with the transition diagrams, the reconfiguration policies are generated. To ensure consistency of the generated artefacts, the constraints specified by the models are used to validate the configurations before any generation is carried out. The middleware platforms enable extensibility and evolution of the system allowing newly generated artefacts (e.g. components, component

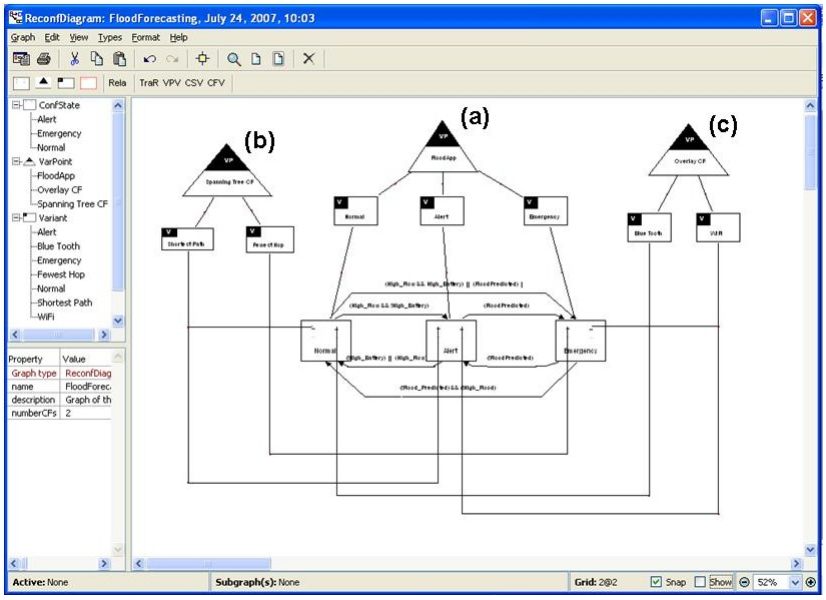

Fig. 7. Variability and Transition Diagrams

configurations, and reconfiguration policies) to be added during the execution of the system.

5 Discussion

In this section we discuss the novel contributions of our research and briefly describe ongoing research that extends the Genie approach.

5.1 Contributions

Architectural changes take place according to environment variations and following the adaptation policies. With Genie, new reconfiguration policies can be modeled and generated off-line while the system is running. Using the capabilities provided by the middleware platforms, the newly generated policies can also be added to the running system, changing dynamically the behaviour of the system. The fact that these policies are explicitly modelled using the Genie approach improves the traceability during the software development process. 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. Identifying the source of the error may require significant effort and time [3]. Genie promotes joint collaboration between requirements engineers, domain experts, software architects, and developers

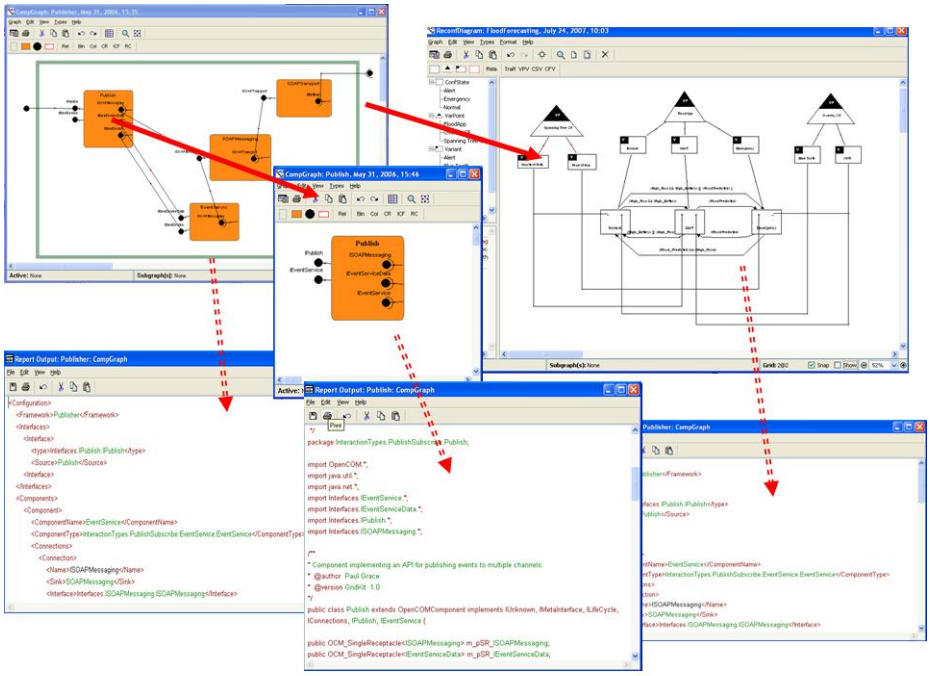

Fig. 8. Genie Models and Generated Artefacts

[21]. Furthermore, the proposed approach makes explicit the support the middleware platforms provide in separating the *system evolution* and *system adaptation* as two simultaneous processes in self-adaptive software [32]. System evolution ensures the consistent application of change over time, and system adaptation focuses on "*the cycle of detecting changing circumstances and planning and deploying responsive modifications*" [32].

5.2 Ongoing Research

Towards the use of Models@run.time

We are already extending and improving the Genie approach. As explained above and as Figure 6 shows, the generated policies mainly specify the trigger events and which reconfiguration scripts have to be loaded to adapt the system from one configuration (state) to another. These scripts are currently hand-written using the support offered by the underlying middleware platforms.

The reconfiguration scripts can also be generated from the DSL-based models during design-time or even at runtime. In the case of the dynamic generation during runtime and when adaptations (transitions) are triggered, the current configuration and the target configuration are compared. The comparison results in the identification of the components that should be added or deleted and allows the dynamic generation of the corre-

sponding reconfiguration script. This solution is possible as we are able to maintain a reference model at the meta-level of the reflective middleware platform. The reference model represents the current system and the possible modified model that is the result of the required adaptation (both models are supplied by Genie). Partial results of such ideas are already reported in [30]. This work opens some research questions as for example: what is the correct order of the deletion and incorporation of components during the reconfiguration process, or what is the impact on the performance of the application. This research is being carried out in the scope of the STREP European project DiVA [14] (work package diva@run.time) and the research topic Models@run.time [4].

Goal-driven requirements

Genie is complemented by the approach Levels of RE for Modeling (LoREM) [21]. LoREM is a goal-driven requirements approach that supports the formulation of the requirements of dynamically adaptive systems helping the analyst to understand the characteristics of the operational environment and the adaptation scenarios that the system can go through. The goal models in LoREM are in a fourth abstraction level and are used to derive the DSL-based models used in Genie. At the bottom, the middleware platform underpins the reorganization of the ongoing architecture at runtime providing support as the requirements imposed by the environment change, see Figure 9. Partial results applied to GridStix can be found in [21].

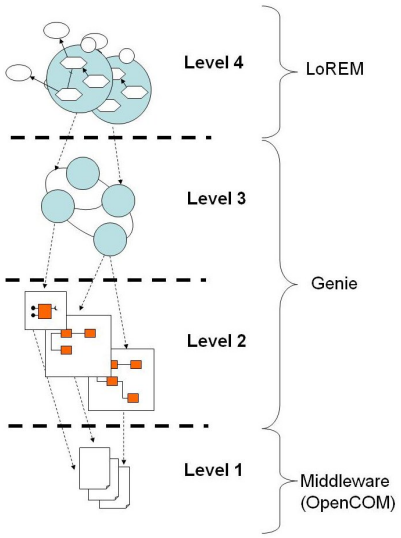

Fig. 9. Genie Models and Generated Artefacts

6 Related Work

Research work by *Floch et all* [15] on the use of architecture models for runtime adaptability, shares the basic principles of our approach as for example the use of 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. *Sora et all* [36] also use architecture-based abstractions for what they call self-customizable system. They composable components which are 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 potential support to determine where the points for variation are, what the possible set of variations are, 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. Neither of the solutions in [15] and [36] provides generative capabilities as we offer. In [21] 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 [36] is similar to our research.

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 [37, 19, 35]. Our approach proposes the model-based architecture management for whole sets of components, their connections and semantics (i.e. we offer a more coarse grained approach).

In [18], *Garlan* and *Schmerl* describe their research work on the use of system monitoring and reflective capabilities using architectural models. Specifically, they describe their approach to monitor the executing system to translate observed events to events that construct and update an architectural model that reflects the actual running system. The final goal is to compare the dynamically-determined model with the correct architectural model. *Garlan* and *Schmerl* argument how inconsistencies found after the comparison can be used to identify implementation errors, or, even possibly, to effect runtime adaptations to correct certain type of faults. Different from *Garlan* and *Schmerl*, we do not deal in this paper with the self-healing issues of adapted systems. Our focus is first, to offer the overall view of the system that domain experts and developers need when planning the adaptations and secondly, to bridge the gap in between the different levels of abstraction used by domain experts and middleware developers. There are different research projects on architecture-based dynamic adaptations that use ADLs including the research by *Garlan* and *Schmerl*. As explained in Section 3.3, the DSL associated with the structural variability in Genie can be considered as an ADL with generative capabilities. A difference between our approach and other ADL-based approaches such as ArchWare, Rainbow, ArchStudio [17, 38, 25] is that our architectural descriptions are always tied to a component framework. Component frameworks offer the architectural principles and constrains that address a specific area of concern (e.g. routing protocol or discovery service). Furthermore, in our case systems can be

assembled from component frameworks in a recursive way. A component plugged into a component framework may be an atomic component, but it can also be a compound component that is a component framework itself. The view provided by this recursivity along with the domain oriented nature provided by component frameworks is different from the often flat view offered by the research projects named above.

When performing dynamic reorganization of the architecture we ensure that updates are completed 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. 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 [22] is relevant and complementary to our research. Finally, we see potential use in combining our approach with the Fujaba project principles [10]. Fujaba supports gradual transitions from one configuration to another and the specification of timing constraints. The principle of gradual transitions can improve the prescriptive policies of our approach.

7 Conclusions and Future Work

This paper has presented an approach called Genie. The approach uses architecture models to support the generation and operation of component-based adaptive systems. We have identified two dimensions of dynamic variability namely *architectural or structural variability* and *environment and context variability*. Genie supports the use of domain-specific languages to specify and validate models based on abstractions of the dynamic variability dimensions. Models describe the architecture of reconfigurable applications and the conditions of the environment and context that trigger the reconfiguration of the architecture. From the models, different software artefacts (e.g. components configurations and reconfiguration policies) can be generated. These artifacts can be dynamically added to the system during its execution using the middleware platforms. Such artefacts support the dynamic architectural reorganization, runtime decision-making and system adaptation mechanisms. Specifically, transition diagram models allow the developer to work at higher levels of abstraction. An important contribution of Genie is the overall view of the different problem domains and the whole process of reconfiguration that the systems can undergo. The approach has been successfully applied in two different case studies.

Substantial research remains to be done. For example, a concern is the combinatorial explosion related to the number of reconfiguration paths in the reconfiguration graphs and the number of adaptation policies). The number of reconfiguration paths in the case study was manageable. However, it might not be the case for other domains. Furthermore, even if the designer specifies the reconfiguration graphs at a high-level of abstraction, the explosion of the number of configurations and transitions to be specified is a potential problem. We are investigating how to dynamically generate the reconfiguration scripts and how to avoid the enumeration of all possible configurations. Partial results are shown in [30].

References

1. Krishnakumar Balasubramanian, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema. Developing applications using model-driven design environments. *IEEE Computer*, pages 33 – 40, 2006.
2. Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2nd edition, 2003.
3. Nelly Bencomo. *Supporting the Modelling and Generation of Reflective Middleware Families and Applications using Dynamic Variability*. PhD thesis, Lancaster University, 2008.
4. Nelly Bencomo, Robert France, and Gordon Blair. 2nd international workshop on models@run.time. In Holger Giese, editor, *Workshops and Symposia at MODELS 2007*, Lecture Notes in Computer Science. Springer-Verlag, 2007.
5. Nelly Bencomo, Paul Grace, Carlos Flores, Danny Hughes, and Gordon Blair. Genie: Supporting the model driven development of reflective, component-based adaptive systems. In *ICSE 2008 - Formal Research Demonstrations Track*, 2008.
6. D.M. Berry, B.H.C. Cheng, and Proc 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. Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming. Special issue: Software variability management*, 53(3):333–352, 2004.
8. Gordon Blair, Geoff Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In Seitz J. Davies N.A.J., Raymond K., editor, *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pages 91–206, The Lake District, UK, 1998.
9. Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11):1257 – 1284, 2006.
10. Sven Burmester, Holger Giese, Martin Hirsch, Daniela Schilling, and Matthias Tichy. The fujaba real-time tool suite: model-driven development of safety-critical, real-time systems. In *ICSE*, 2005.
11. Paul Clements and Paul Kogut. The software architecture renaissance. *Crosstalk - The Journal of Defense Software Engineering*, 7(11), 1994.
12. G. Coulson, G.S. Blair, P. Grace, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems*, February 2008.
13. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.
14. DiVA. Diva-dynamic variability in complex, adaptive systems, http://www.ict-diva.eu/, 2008.
15. Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *Software IEEE*, 23(2):62–70, 2006.
16. Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In L. Briand and A. Wolf, editors, *FoSE*. IEEE-CS Press, 2007.
17. David Garlan. *Software Architecture: a Roadmap*. ACM Press, 2000.
18. David Garlan and Bradley Schmerl. Using architectural models at runtime: Research challenges. In *European Workshop on Software Architectures*, St. Andrews, Scotland, 2004.
19. Michael Goedicke, Klaus Pohl, and Uwe Zdun. Domain-specific runtime variability in product line architectures. In *8th International Conference on Object-Oriented. Information Systems*, pages 384 – 396, 2002.

20. Aniruddha Gokhale, Krishnakumar Balasubramanian, and Tao Lu. Cosmic: addressing crosscutting deployment and configuration concerns of distributed real-time and embedded systems. In *OOPSLA '04 Companion Book*, pages 218–219, NY, USA, 2004. ACM.
21. Heather J. Goldsby, Pete Sawyer, Nelly Bencomo, Danny Hughes, and Betty H.C. Cheng. Goal-based modeling of dynamically adaptive system requirements. In *15th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS)*, 2008.
22. Hassan Gomaa and Mohamed Hussein. Model-based software design and adaptation. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS'07)*, 2007.
23. Danny Hughes, Phil Greenwood, Geoff Coulson, Gordon Blair, Florian Pappenberger, Paul Smith, and Keith 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.
24. Fabio Kon, Fabio Costa, Gordon Blair, and Roy Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.
25. Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *FoSE '07: 2007 Future of Software Engineering*, pages 259–268. IEEE Computer Society, 2007.
26. Philippe Kruchten and Christopher Thompson. An object-oriented, distributed architecture for large scale ada systems. In *Tri-Ada '94*, Baltimore, Maryland, 1994.
27. Harold Lawson, Vassilka Kirova, and William Rossak. A refinement of the ecbs architecture constituent. In *International Symposium and Workshop on Systems Engineering of Computer Based Systems*, Tucson, Arizona, 1995.
28. Jaejoon Lee and Dirk Muthig. Feature-oriented variability management in product line engineering. *Communications of the ACM*, 49(12), 2006.
29. Philip K. McKinley, Seyed M. Sadjadi, Eric P. Kasten, and Betty H.C. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.
30. Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jezequel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *MODELS'08 Conference*, France, 2008.
31. R. van Ommering. *Building Product Populations with Software Components. PhD Thesis*. PhD thesis, Rijksuniversiteits Groningen, 2004.
32. Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications*, 14(3):54–62, 1999.
33. Peyman Oreizy, David S. Rosenblum, and Richard N. Taylor. On the role of connectors in modeling and implementing software architectures. Technical Report 98-04, University of California, Irvine, 1998.
34. Klaus Pohl, Gnter Böckle, and Frank van der Linden. *Software Product Line Engineering-Foundations, Principles, and Techniques*. Springer, 2005.
35. E. Posnak and G. Lavender. An adaptive framework for developing multimedia. *Communications ACM*, 40(10):43–47, 1997.
36. Ioana Sora, Vladimir Cretu, Pierre Verbaeten, and Yolande Berbers. Managing variability of self-customizable systems through composable components. *Software Process: Improvement and Practice*, 10(1):77–95, 2005.
37. Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705 – 754, 2005.
38. Richard N. Taylor and Andre van der Hoek. Software design and architecture the once and future focus of software engineering. In *International Conference on Software Engineering ICSE 07 (FoSE 2007)*, 2007.