MODELS AT RUNTIME TO SUPPORT DYNAMIC ADAPTATION

Brice Morin, INRIA
Olivier Barais and Jean-Marc Jézéquel, INRIA and IRISA, University of Rennes
Franck Fleurey and Arnor Solberg, SINTEF ICT

An approach for specifying and executing dynamically adaptive software systems combines model-driven and aspect-oriented techniques to help engineers tame the complexity of such systems while offering a high degree of automation and validation.

Today's society increasingly depends on software systems deployed in large companies, banks, airports, and so on. These systems must be available 24/7 and continuously adapt to varying environmental conditions and requirements. Such *dynamically adaptive systems* exhibit degrees of variability that depend on user needs and runtime fluctuations in their contexts. Engineers can develop DASs by defining several variation points. Depending on the context, the system dynamically chooses suitable variants to realize those variation points. These variants may provide better quality of service (QoS), offer new services that did not make sense in the previous context, or discard some services that are no longer useful.

DASs range from small embedded systems to large systems of systems and from human-driven to purely self-adaptive systems. A DAS can be conceptualized as a *dynamic software product line* in which variabilities are bound at runtime.[1] Similar to traditional SPLs,[2] the number of possible configurations of a DSPL grows combinatorially with the number of variation points and variants. In an SPL, products are derived by human decisions and are totally independent; a DSPL also must handle the migration paths between those configurations.

The configurations produced by a DSPL are thus highly dependent: The system should evolve at runtime between its current configuration (source) and a new configuration (target) through a safe migration path (transition). Several stimuli trigger these transitions: context changes, user preferences, and so on. The DSPL must provide support for describing the adaptation logic.

A DSPL's execution can be abstracted as a highly connected state machine,[3,4] where the states are the possible system configurations and the transitions the migration paths. Fully specifying this state machine lets the designer perform extensive simulation, validation, and testing of the system's dynamic variability before actually implementing the system.[5] Model-driven engineering (MDE) techniques then make it possible to fully generate the adaptive system's code4 from the state machine specification.

 Published by the IEEE Computer Society

However, this approach suffers from two main drawbacks related to adaptation management and evolution management:[6]

- *Explosion in the number of artifacts.* Even when the designer specifies the state machine at a high level of abstraction, the number of configurations and transitions to be described grows rapidly. For example, in combining the features of one dynamic customer relationship management (DCRM) system, we counted 92,160 configurations;[7] this leads to 92,160 92,159 = 8,493,373,440 possible transitions and triggers among these configurations. Validating all these artifacts can soon become a problem: The number of configurations explodes in a combinatorial way with regard to the number of variants, and the number of transitions is quadratic with regard to the number of configurations. While it is still possible to specify this state machine for simple adaptive systems, this rapidly becomes a daunting task in the case of large systems comprising a wide range of variation points.[8]
- *Evolution of the adaptive system.* Once the adaptive system is deployed and running, it can evolve based on new user needs, detection, and correction of limitations or security weaknesses. Evolving an adaptive system involves dynamically changing the adaptation state machine: adding and removing states and transitions. Applying a classic MDE approach to generate all the application code from higher-level specifications would be impractical: The system would have to be stopped and decommissioned before a new system—based on modifications to the state machine—could be deployed and started. This would make the system unavailable for a long period, which in many cases would be unacceptable.

In our work on the EU-ICT DiVA project (Dynamic Variability in complex, Adaptive systems; www.ict-diva.eu), we address these two drawbacks by using software models at runtime as well as at design time. We rely on four metamodels supported by design tools, such as graphical or textual editors and validators or simulators, to assist in modeling the DSPL at design time. Models conforming to these four metamodels are the main data manipulated by the runtime infrastructure responsible for dynamically adapting component-based applications at runtime. These models provide a high-level basis for reasoning efficiently about relevant aspects of the system and its environment and offer enough details to fully automate the dynamic adaptation process. It is possible to make the design specifications evolve at any time, before initial deployment or while the system is already running.

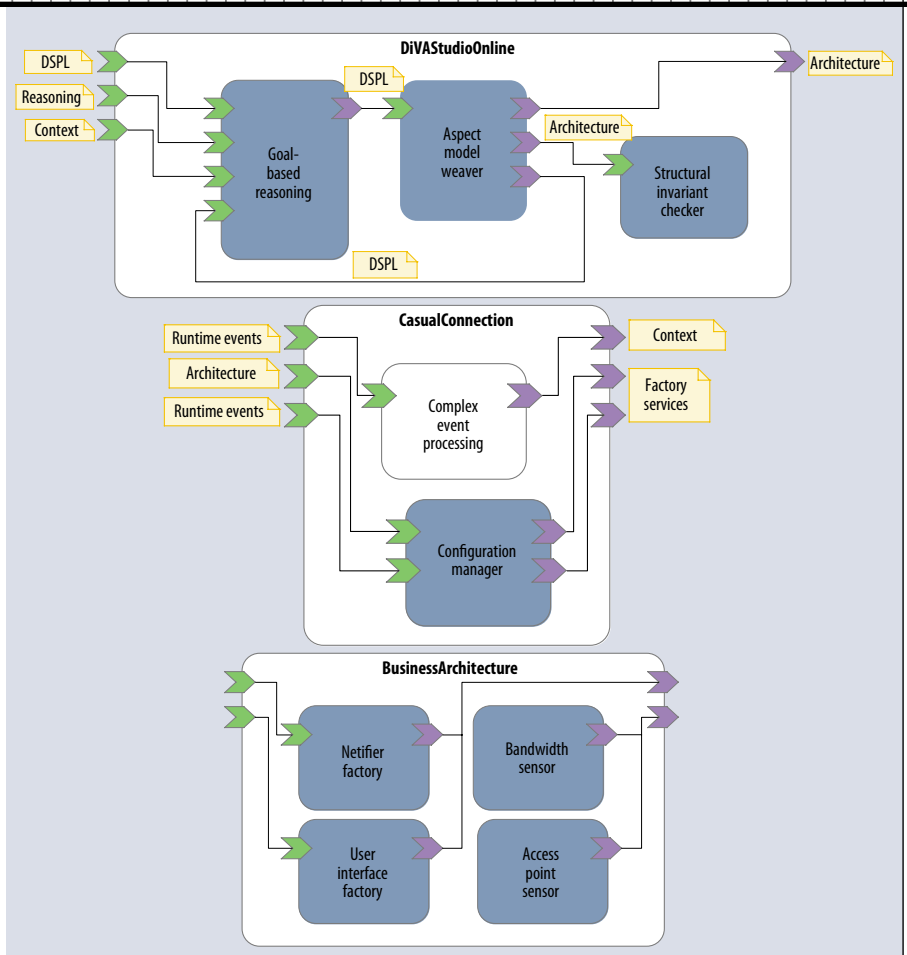

Figure 1. Runtime architecture to support dynamic software product lines.

MODEL-ORIENTED ARCHITECTURE

Figure 1 shows the architecture for managing DSPLs at runtime, comprising three layers:

- *DiVAStudioOnline*, a platform-independent layer that only manipulates models;
- *CausalConnection*, a platform-specific layer that links the model space to the runtime space; and
- *BusinessArchitecture*, an application-specific layer that

DESIGNING FEATURES AS ASPECT MODELS

In DiVA, we leverage aspect-oriented modeling (AOM) techniques to refine features and automatically build complete configurations before the actual adaptation. A base model refines the system's commonalities—elements present in all the configurations—as an architecture made of components and the connections between them, or bindings. Aspect models refine the system's variants by specifying their precise architectures: Each model is an architectural fragment that contains all the information needed to be easily plugged into the base architecture.

As Figure A shows, an aspect model consists of three parts:

- *Advice model.* This architectural fragment specifies what is needed to realize the associated variant. An advice need not be fully consistent. The base model should bring the missing elements needed to make the advice model consistent when weaving the aspect.
- *Pointcut model.* This architectural fragment specifies the components and bindings that the aspect model expects from the base model to be woven—that is, where the aspect should be woven. The most precise is the pointcut model, the smallest is the set of potential places where the aspect can be woven, and vice versa For example, if a component's type is not specified in the pointcut model, this component would be matched by any of the base architecture's components, irrespective of its real type.
- *Composition protocol.* This describes how to integrate the advice model into the pointcut model. When weaving the aspect, the places matching the pointcut model automatically contextualize the composition protocol to actually weave the aspect into the base model.

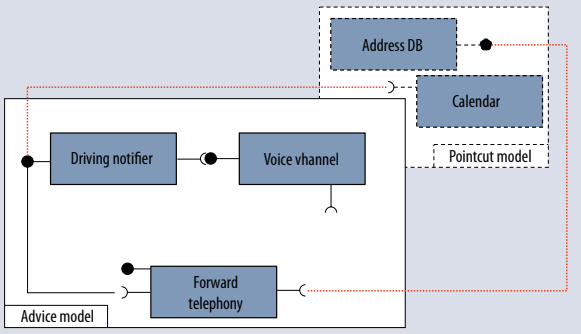

Figure A. Aspect model. An advice model specifies what is needed to realize the associated variant, a pointcut model specifies the components and bindings that the aspect model expects from the base model to be woven, and a composition protocol describes how to integrate the advice model into the pointcut model.

Refining features as aspect models allows the designer to validate the adaptive system one step further. Since AOM relies on a strong theoretical background, such as graph theory, it is possible to perform analysis—for example, critical pair or confluence analysis—to detect previously undetected aspect dependencies and interactions.1 The designer can update the DSPL and reasoning models' constraints to avoid invalid interactions. It is also possible to refine the simulation step by actually producing, via aspect weaving, and validating some detailed configuration corresponding to foreseen contexts.

Reference

1. P. Jayarman et al., "Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis," *Proc. 10th Int'l Conf. Model-Driven Eng. Languages and Systems* (MoDELS 07), LNCS 4735, Springer, 2007, pp. 151-165.

defines factory components, which simply provide services to instantiate multiple component instances. This layer is not part of the infrastructure for managing DSPLs.

Five components—a complex event processor, a goal-based reasoning engine, an aspect model weaver, an online configuration checker, and a configuration manager—interact by exchanging models conforming to the metamodels.

Metamodels

The components exchange four kinds of metamodels: DSPL, context, reasoning, and architecture.

DSPL. This is a feature model that describes the system's variability. Commonly used in the SPL community,[2] feature models describe hierarchies with mandatory features, options, alternatives—*n* among *p* choices, and so on—as well as constraints (requires, excludes) among features. The DSPL model is a regular feature diagram model, with a naming convention to refer to architectural fragments, or *aspect models*, refining the features. In this way, engineers can exploit any existing feature model tools—graphical editors, checkers, and others—with no modification.

Context. This model specifies the system's environment. A set of context variables specifies those aspects of the environment relevant to adaptation. At runtime, the variable values are provided by context sensors, and these may trigger a system reconfiguration.

Reasoning. This model describes selection of the DSPL's features according to context. Several formalisms exist such as event-condition-action (ECA) rules[9] or goal-based optimization rules.[7] We do not impose any particular reasoning model. An ECA model will typically describe, for particular contexts, which features to select. A goal-based model will typically describe how features impact QoS properties—using, for example, help and hurt relationships[10]—and specify when QoS properties should be optimized, for example, when a property is too low.

Architecture. This model describes component-based architectures. Designers can use any metamodeling framework, such as the Unified Modeling Language (UML) or Service Component Architecture (SCA), or any architecture description language, to describe the architecture. Because our dynamic adaptations are currently reconfigurations, our focus is on components and bindings, concepts that are present in metamodels. In practice, we have defined our own minimal metamodel[11] to reduce memory overhead at runtime. Other metamodels map to our metamodel via model transformations in Kermeta (www.kermeta.org).[12]

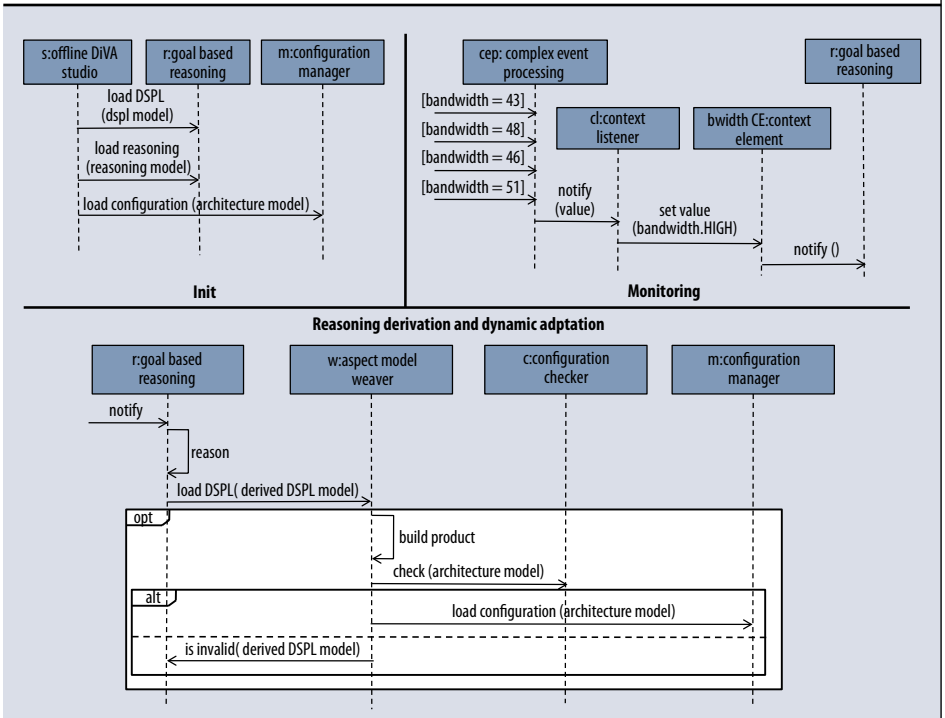

Figure 2. Interactions between the components: (a) initialization; [(b) monitoring; (c) reasoning, derivation, and dynamic adaptation.

The architecture model refines each leaf feature of the DSPL model into an architectural fragment. As the "Designing Features as Aspect Models" sidebar describes, we use aspect-oriented modeling (AOM) techniques to design and compose features into a core model containing the mandatory elements.

Designing the models

Engineers design these models offline before the initial system deployment or while the system is already running, but independently of the running system, and leverage them at runtime to drive the dynamic adaptation process. The quality and correctness of models conforming to these metamodels are crucial and must be checked as early as possible. Since the components of the DiVAStudioOnline layer only exchange models conforming to these four metamodels, it is very easy to validate the adaptation logic: A test component simply produces a set of input models, and an another test component analyzes the models produced by the DiVAStudioOnline components. For example, a test component provides a DSPL model, a reasoning model, and a scenario (sequence of context models) to simulate the system's adaptation logic,[5] and another test component checks the produced configurations.[8] These configurations must ensure the constraints (cardinalities, requires/excludes) defined in the DSPL model and include features suitable to the reasoning model.

The choice of the four metamodels, which type the interaction between the components, is open. We currently use SCA to design architectures and two ad hoc metamodels for the DSPL and contexts. We previously used ECA models to specify the adaptation logic;[5] we are now using a goal-based optimization model. Once a designer has chosen the four metamodels, they strongly type the architecture.

Leveraging the models

The sequence diagrams shown in Figure 2 specify the interactions between the architectural components. The key idea is to maintain a context model and an architectural model that both synchronize with the runtime system.

The context model is updated when relevant changes appear in the running system's execution context—for example, CPU load, free memory, or bandwidth. This model is not causally connected with the runtime system—it reflects what happens at runtime (in the execution context), but it should not be directly modified to adapt the running system. On the contrary, the context model serves as a basis for reasoning about the environment and determining a new configuration more adapted to the current context, if necessary.

The architectural model is updated when the running system evolves—that is, when components and bindings are added or removed. Note that this model is not directly manipulated to adapt the running system. On the contrary, the aspect model weaver component produces another architectural model (configuration) when the system should

adapt, depending on the current context model. This configuration is then checked, and the causal connection layer finally realizes the dynamic adaptation,[8] without having to write low-level and error-prone reconfiguration scripts. If the new configuration is not valid, the aspect model weaver component simply discards it and does not proceed to the causal connection layer. Indeed, since the running system has not yet been adapted, it is not necessary to perform a rollback.

Components

Each of the five architectural components has a clear role and well-defined interactions with other components.

Complex event processor. This component observes runtime events generated by probes integrated into the system. When a sequence of events matches a query, expressed in the Event Query Language (EQL), it notifies an observer. This observer knows exactly which model element of the context model it has to update. When this model element is updated, it notifies the goal-based reasoning component. Complex event processing components, such as Esper (http://esper.codehaus.org), allow defining advanced queries on runtime events with time windows and aggregation functions—min, max, average, and so on. Unlike hard thresholds, these queries simplify dealing with permanent context oscillations, for example, by defining thresholds on average values computed on a time slot.

The DCRM's objective is to provide accurate client-related information depending on the context.

Goal-based reasoning engine. When the context model is updated, this component computes a derived DSPL that only contains the mandatory features and a selection of variable features, adapted to the current context. Although we use a goal-based reasoning model, other reasoning models are available. This component is initialized with a DSPL model and a reasoning model. At any time, it is possible to update the DSPL or the reasoning model: add, remove, or update features or reasoning rules, and so on.

Aspect model weaver. This component receives a derived DSPL from the reasoning engine. For all the features of this DSPL, the weaver composes the corresponding aspect to produce a global configuration. This configuration is then checked before it is submitted, if valid, to the configuration manager.

Configuration checker. This online component checks that aspect weaving obtains a consistent configuration.[8] It checks general invariants, which should be enforced in any application, as well as user-defined invariants, which depend on a given application. If the configuration is valid, the aspect model weaver then sends the configuration to the configuration manager.

Configuration manager. This component receives a configuration from the aspect model weaver. It is responsible for configuring and reconfiguring the business architecture. To create and bind components, the configuration manager calls the services offered by the factories (component types).[8] This component maintains a model representing the running system by using the introspection and observation mechanisms provided by the platform. When a new configuration is produced, it compares this new configuration (model) with the current model and deduces a safe sequence of reconfiguration commands. These commands consist of adding or removing components or binding.

DYNAMIC ADAPTATION IN ACTION

We illustrate our approach using the DCRM system developed by CAS Software, our industrial partner in the DiVA project. A simplified version of the system is freely available at www.ict-diva.eu/DiVA/results/tools-and-prototypes/demo.zip/view.

The DCRM's objective is to provide accurate client-related information depending on the context. For example, when the user is working in his office, the system can notify him by e-mail, via a rich Web-based client. He can also access critical resources as he is connected to a trusted network. When the user is driving his car to visit a client, messages received by a mobile or smart phone should notify him of information that is either critical or related to the client. If he is using a mobile phone, he can be notified via the short message service or audio/voice, and Java Telephony API forwards phone calls from his office. If he is using a smart phone, he can also use a lightweight Web client.

Figure 3a shows two reconfiguration scripts generated on the fly. Figure 3b shows the system interface, which consists of two parts:

- *Monitoring.* In the left part of the interface (green rectangle), the user can simulate different environmental variables. Actual runtime sensors are replaced by a check box to simulate Boolean or enumeration (for example, the access point) or sliders to simulate continuous values (for example, the bandwidth). When these elements are activated, they generate exactly the same kind of events as the real sensors. The user thus simulates the environment in a transparent way for the complex event processing component.
- *Reasoning.* The right part of the interface (orange rectangle) graphically represents the DSPL model. Three

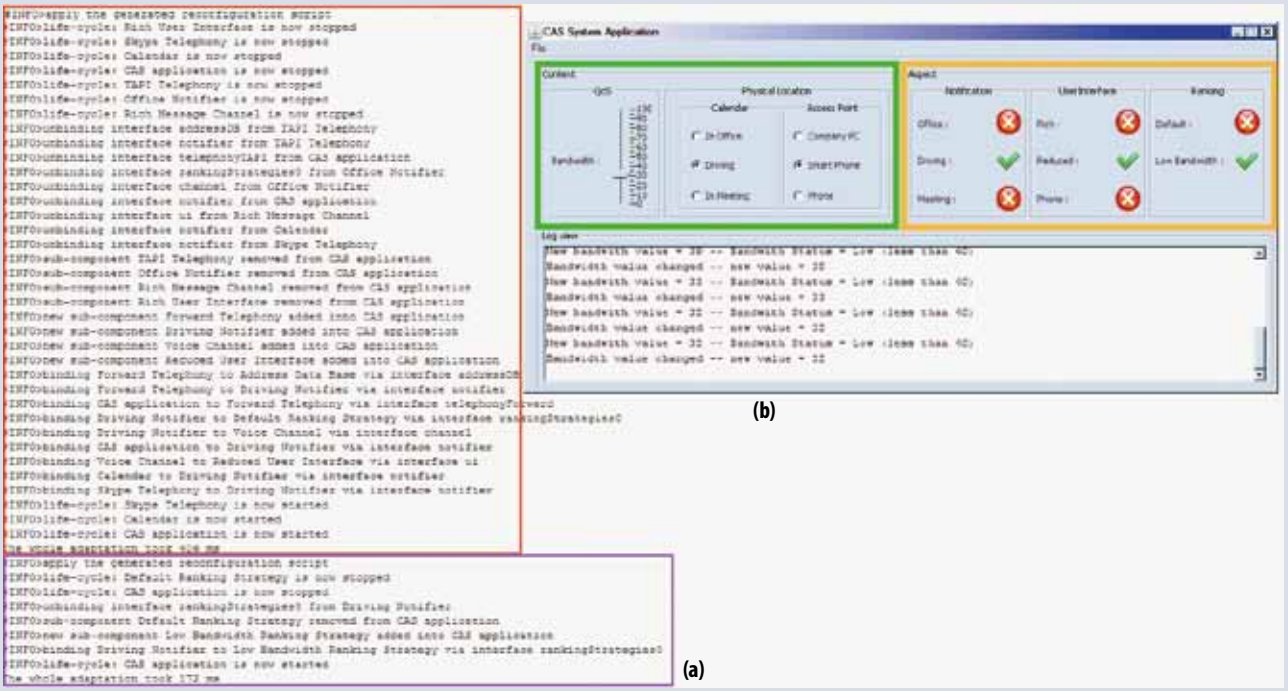

Figure 3. Using the DiVA architecture to manage CAS Software's DCRM system: (a) two reconfiguration scripts generated on the fly; (b) the interface's monitoring and reasoning components.

main features comprise the system: notification, user interface, and ranking. For each feature, the system defines several subfeatures. For example, the system can adapt to provide different notification mechanisms: office, driving, or meeting. An aspect model refines each of these subfeatures. The current configuration, determined by the goal-based reasoning component, is displayed in green.

In the initial context, the user is working in his office. His electronic calendar notifies him that he must visit a client in 30 minutes. When the user logs off his PC and logs on to his smart phone, the reasoning engine computes a new configuration corresponding to a mobile environment. The system replaces the office notifier by a driving notifier, and the user interface switches from rich to reduced. Weaving the associated aspect produces a new configuration. After validating this new configuration, the system automatically generates a reconfiguration script and executes it at runtime, as shown in the bottom left part of Figure 3 (red rectangle).

We assume that the user encounters bandwidth limitation. As the monitoring sequence diagram (Figure 2) shows, when the value exceeds 40 percent of the maximum bandwidth value for 10 seconds, the system updates the context model with bandwidth = high. As soon as the value is below 40 percent, it updates the context model with bandwidth = low. Note that if the bandwidth's value oscillates around 40 percent, the context model will remain stable—bandwidth is low. In Figure 3, this value is currently 32 percent. The reasoning component thus decides to switch from the default ranking strategies to the low bandwidth strategy. Similar to the previous reconfiguration, this generates a script, as illustrated in the top left part of Figure 3 (purple rectangle).

RELATED WORK

Several other research approaches use architectural models to support dynamic adaptation and software evolution. A decade ago, Peyman Oreizy and colleagues[6] promoted an architecture-based approach to self-adaptive software systems. They stressed that an adaptive system should be open to introducing new behaviors and adaptation plans at runtime.

David Garlan and colleagues[13] also used architectural models for system monitoring and reflection. Specifically, they monitored the executing system to translate observed events to events that construct and update an architectural model that reflects the actual running system. They found that detected inconsistencies could be used to effect runtime adaptations to correct certain type of faults. Similar to our work, they sought to compare the dynamically determined model with the correct architectural model.

In the context of mobile applications, Jacqueline Floch and colleagues[14] used architectural models and utility functions to describe the dynamic variability of such ap-

plications. Their system's main adaptation mechanism replaces the implementation of components at runtime. For each possible implementation (variant) of a component, a fine-grained utility function specifies a precise context in which the variant is useful. Depending on the context, the system integrates most useful variants into the architecture. However, it does not provide support to simulate or validate the adaptation logic at design time.

The Genie approach[3] also uses architectural models to support the generation and execution of adaptive systems leveraging component-based middleware technologies. A state machine specifies the system's adaptive logic. Each state represents a system configuration, and each transition describes when and how—via reconfiguration scripts—to dynamically switch from one configuration to another. From these models, Genie generates various artifacts such as configuration files and ECA adaptation policies. These artifacts can be dynamically inserted during execution.

Our approach focuses on taming the explosion in the number of artifacts while providing a high degree of automation and validation.

Our approach goes one step further than previous efforts. We explicitly design four fundamental aspects of a DAS: its variability (using a feature diagram), the system's environment and the context (valuation of the environment), the adaptation logic, and the system architecture. We particularly focus on taming the explosion in the number of artifacts while providing a high degree of automation and validation. We use AOM techniques to automatically build architectures by composing aspects associated with features, instead of fully specifying all the possible configurations. After validation, we then use MDE techniques to produce reconfiguration scripts that make the system switch from its current configuration to a target configuration more adapted to the current context.

Dynamically adaptive systems play an increasingly vital role in today's society. In addition to CAS Software's DCRM system, we have applied our process to a house-automation system currently deployed in the Rennes metropolitan area in Britanny, France, to help elderly or disabled people remain at home.8 In this DAS, dynamic adaptation is mostly driven by humans and depends on such factors as the evolution of physical handicaps and the installation of new devices. The number of possible configurations (1014) and transitions (1028) in this system literally explodes. In the context of the DiVA project, we will also apply our approach to an airport crisis management system that should adapt to different crisis types and deal with different roles—for example, airport staff, firemen, and medical staff.

Our tool-supported approach relies on a clear and modular architecture in which components exchange models related to the system's variability, environment, and architecture, and variability is dynamically bound to the context. We do not impose specific metamodels to describe these models. In the DiVA project, we have reused some existing metamodels and designed other ones, and we have used the OSGi platform to implement this architecture.

By explicitly defining a DAS as a DSPL, engineers can avoid designing by hand all of the system's possible configurations and transitions. Depending on the environment or user needs, they explicitly construct a suitable configuration using AOM techniques; they validate this configuration using traditional MDE techniques: invariant checking, simulation, and so on. Finally, the system automatically generates a safe reconfiguration script to actually adapt the running system. If the produced configuration is not consistent, the designer simply discards the configuration and derives a new one. Since the running system has not been adapted yet, it is not necessary to perform a rollback. This process is open to evolution—designers can make the DSPL evolve by seamlessly adding or removing variants, constraints, rules, and so on.

In future work, we plan to improve our reasoning framework and the dynamic adaptation process. In a critical context, the system must react quickly—it can, for example, choose a predefined, prevalidated configuration. In a noncritical context, the system can spend some time to reason and build a suitable configuration. We will investigate the use of bacteriological algorithms to implement reasoning algorithms that can find solutions in a given time budget. Currently, the dynamic adaptation process relies on our aspect model weaver implemented in Kermeta. To make aspect model weaving an efficient solution for dynamic adaptation, we will rely on the compilation feature offered by the language's new version. n

Acknowledgments

This work was partially funded by the DiVA project (EU FP7 STREP, contract 215412, www.ict-diva.eu). The case study presented in this article is provided by CAS Software AG (DiVA industrial partner, www.cas.de/english).

References

1. S. Hallsteinsen et al., "Dynamic Software Product Lines," *Computer*, Apr. 2008, pp. 93-95.

2. P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, 3rd ed., Addison-Wesley Professional, 2001.
3. N. Bencomo et al., "Genie: Supporting the Model-Driven Development of Reflective, Component-Based Adaptive Systems," *Proc. 30th Int'l Conf. Software Eng.* (ICSE 08), ACM Press, 2008, pp. 811-814.
4. J. Zhang and B.H.C. Cheng, "Model-Based Development of Dynamically Adaptive Software," *Proc. 28th Int'l Conf. Software Engineering* (ICSE 06), ACM Press, 2006, pp. 371-380.
5. F. Fleurey et al., "Modeling and Validating Dynamic Adaptation," *Models in Software Eng.*, LNCS 5421, Springer, 2008, pp. 97-108.
6. P. Oreizy et al., "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems*, May 1999, pp. 54-62.
7. F. Fleurey and A. Solberg, "A Domain-Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems," to appear in *Proc. ACM/IEEE 12th Int'l Conf. Model-Driven Eng. Languages and Systems* (MoDELS 09), ACM Press, 2009.
8. B. Morin et al., "Taming Dynamically Adaptive Systems with Models and Aspects," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), IEEE CS Press, 2009, pp. 122-132.
9. P.C. David and T. Ledoux, "An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components," *Software Composition*, LNCS 4089, Springer, 2006, pp. 82-97.
10. H.J. Goldsby et al., "Goal-Based Modeling of Dynamically Adaptive System Requirements," *Proc. 15th Ann. IEEE Int'l Conf. and Workshop Eng. of Computer Based Systems* (ECBS 08), IEEE CS Press, 2008, pp. 36-45.
11. B. Morin, O. Barais, and J.M. Jézéquel, "K@rt: An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines," *Proc. 3rd Int'l Workshop Models@run.time* (MoDELS 08); www.irisa.fr/triskell/publis/2008/Morin08e.pdf.
12. P.A. Muller, F. Fleurey, and J.M. Jézéquel, "Weaving Executability into Object-Oriented Meta-Languages," *Proc 8th Int'l Conf. Model-Driven Eng. Languages and Systems* (MoDELS 05); www.irisa.fr/triskell/publis/2005/Muller05a.pdf.
13. D. Garlan et al., "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *Computer*, Oct. 2004, pp. 46-54.
14. J. Floch et al., "Using Architecture Models for Runtime Adaptability," *IEEE Software*, Mar. 2006, pp. 62-70.

Brice Morin *is a PhD student in the INRIA Triskell research team. His research interests include applying model-driven and aspect-oriented techniques to tame the complexity of dynamically adaptive systems, from design-time to runtime. He received an MS in computer science from the University of Rennes. Contact him at brice.morin@inria.fr.*

Olivier Barais *is an associate professor at the University of Rennes and member of the INRIA Triskell research team. His research interests include model-driven software engineering, component-based software engineering, and aspect-oriented modeling. He received a PhD in computer science from the University of Lille. Contact him at barais@irisa.fr.*

Jean-Marc Jézéquel *is a professor at the University of Rennes and leader of the INRIA Triskell research team. His research interests include model-driven software engineering for telecommunications and distributed systems. He received a PhD in computer science from the University of Rennes. Contact him at jean-marc.jezequel@irisa.fr.*

Franck Fleurey *is a research scientist in the Model-Driven Software Development Group at SINTEF ICT. His research interests include model-driven engineering, domain-specific languages, variability and adaptation modeling, and model composition. He received a PhD in computer science from the University of Rennes. Contact him at franck.fleurey@sintef.no.*

Arnor Solberg *is a senior research scientist and leader of the Model-Driven Software Development Group at SINTEF ICT. He currently serves as technical manager for the EU-ICT DiVA project. He is an expert on software architectures and software engineering practices. He received a PhD in computer science from the University of Oslo. Contact him at arnor.solberg@sintef.no.*