Using architecture models for runtime adaptability

Article *in* IEEE Software · April 2006

DOI: 10.1109/MS.2006.61 · Source: IEEE Xplore

Using Architecture Models for Runtime Adaptability

Jacqueline Floch, Svein Hallsteinsen, and Erlend Stav, *SINTEF ICT*

Frank Eliassen, Ketil Lund, and Eli Gjørven, *Simula Research Laboratory*

Exploiting architecture models and generic middleware enables runtime adaptation for mobile computing applications.

Every software system has an architecture. The architecture strongly influences the software system's properties, including maintainability and runtime properties such as performance and reliability. By describing the architecture in models, we can make the architecture explicit. Developers typically use software architecture models at design time to capture the significant decisions about a software system's organization and to describe and establish a common understanding about the system's abstract properties. They further use the models during implementation to ensure that the system will meet its requirements.

In current practice, developers usually don't represent architectural information explicitly at runtime. Rather, they transform and realize the architectural properties through runtime artifacts. This isn't a problem as long as the software system's set of requirements remains fixed at runtime. However, in settings where user needs and operating conditions vary dynamically, a system implemented to fulfill a fixed set of requirements will often fail to operate adequately. We need software systems that can adapt themselves to changing user needs and operating environments. Recently, several research projects have proposed using architecture models at runtime to achieve adaptability.[1,2]

Today, self-adaptation is complex and costly to implement and has typically been applied in domains where systems must provide continuous operation or guaranteed dependability—for instance, software for controlling telephone exchanges or space vehicles. Following the increasing mobility and pervasiveness of computing and communication, we need ways to make self-adaptation affordable for everyday systems.[3,4] Mobile systems face frequent and significant variability in their operating environments as well as varying user needs. To retain usability, usefulness, and reliability under such circumstances, these systems also must adapt to changing environments (see the related sidebar). In the MADAM (*m*obility- and *ad*aptation-en*a*bling *m*iddleware) project (www.ist-madam.org), we aim to facilitate adaptive application development for mobile computing. We follow an architecture-centric approach where we represent architecture models at runtime to allow generic middleware components to reason about and control adaptation.[5]

0740-7459/06/$20.00 © 2006 IEEE

MADAM at a glance

Figure 1 shows MADAM's main ideas. The middleware has three main functions:

- detect context changes,
- reason about the changes and make decisions about what adaptation to perform, and
- implement the adaptation choices.

Context consists of elements directly representing the operating environment and more complex elements aggregated or derived from other elements. For instance, context includes issues describing the system infrastructure (such as battery level and network resources) and the user (such as position, noise, and user needs).

To fulfill these functions, the adaptation middleware requires knowledge of the application structure and constraints, meaning that it should understand a software system's architecture. The architecture model available at runtime describes the information the middleware needs.

Why architecture models?

Several self-adaptation approaches exist.[1] To achieve self-adaptation, developers frequently use programming language features, such as conditional expressions, parametrization, and exceptions. However, these approaches introduce complexity by intertwining adaptation and application behaviors. Also, they scale poorly—for instance, when faced with multiple adaptation triggers—making software evolution difficult. Conversely, approaches that use application-independent middleware for adaptation relieve the applications from adaptation concerns.[6] Several recently proposed middleware-based adaptation approaches concentrate on adapting the underlying platform functionality (such as concurrency management, connection management, or service discovery[7,8]), while other approaches address adapting applications.[2,5]

External approaches based on middleware and architecture models simplify adaptive-applications development.[2] By separating application and adaptation concerns, external approaches make independent analysis and evolution of application and adaptation possible. The MADAM approach supports evolution at both design time and runtime. We achieve the latter through modularization and flexibility in plugging in new components and models. In addition to evolution, external approaches support reuse of adaptation mechanisms as the middleware components realize them.

Changing Environments: An Example

Consider, for instance, mobile service technicians responsible for inspecting and maintaining various geographically spread technical installations. To perform their work, the technicians use handheld PDAs to access support applications covering fault-report registration, repair-job definition, and work assignment.

During a mobile technician's working day, several changes can occur in the work environment. For example, the technician could experience varying network coverage or work for long periods without access to outlet power (thus depending on limited battery lifetime), and he might get his hands full while fixing equipment. When such changes occur, applications should adapt. For example, the application structure might adapt from a thin to medium or self-reliant client; the user interface might switch between keyboard- or voice-based.

Finding the optimal application variant in a given environment isn't obvious: several parameters could influence the adaptation strategy selection, and the adaptation could affect the operating environment's system resources. For example, we'd want a self-reliant client structure rather than a thin or medium client when network bandwidth is poor; however, this variant would also require more memory resources than the two other variants.

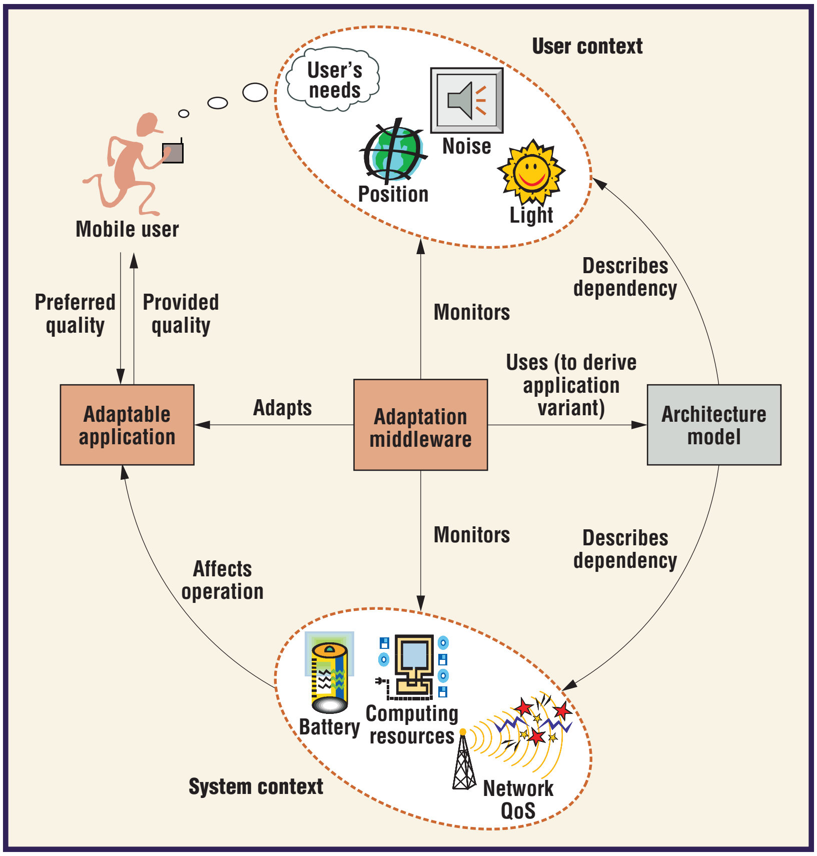


Figure 1. A mobile adaptive system in changing environments.

For the middleware to adapt an application, it requires a representation of the application architecture model that describes variability. Researchers have conducted much work on variability modeling in *product family* approaches.[9,10] Product family architectures describe rules governing the derivation of system variants with different properties from a common component framework. Those rules typically take the form of variation points in the architecture that specify optional or alternative system elements—for instance, alternative component implementations.

We use extended goal policies expressed as utility functions, leaving the system to reason on the actions required to implement those policies.

Self-adaptation

Although our approach leverages techniques for variability developed in the product family community, self-adaptation for mobile computing imposes particular constraints that require adjusting and extending these techniques. The following constraints, which our work addresses, strongly influence the way we build architecture models:

- Adaptation occurs at runtime, and a main challenge is minimizing adaptation's effect on service operation and performance. This means that we need to reduce the complexity of selecting and generating a variant.
- Adaptation reasoning takes place on mobile devices. As these devices have restricted computing and memory resources, we must limit the runtime overhead required to reason about and perform adaptation.
- Adaptation often impacts system resources requirements and offered service quality—particularly important in resource-constrained devices. This means that selecting an adequate adaptation form is far from trivial and requires advanced reasoning.

Researchers have proposed several techniques to simplify software product configuration.[11] However, these works focus on configuration at design or launch time, not at runtime. Self-adaptation techniques such as those proposed by the Rainbow approach[4] are more relevant to MADAM in that they attempt to separate adaptation from application logic concerns. Nevertheless, Rainbow makes no attempt to tackle mobile environment requirements. Furthermore, its developers appear to have based its adaptation strategies on *situation-action rules*, which specify exactly what to do in certain situations. In contrast, we use extended goal policies expressed as *utility functions*—a higher level of adaptation strategy specification that establishes behavioral objectives—leaving the system to reason on the actions required to implement those policies.

MADAM architecture models

Traditionally, the architect takes responsibility for describing an architecture matching a specific context. To support adaptation, the architect must encode variation and selection criteria so the middleware can automate the derivation of a variant for a specific context at runtime. The MADAM architecture model includes specification of the application structure, the application's variability and distribution aspects, the properties of each variant, and the utility functions for comparing variants.

Variability

Similar to the product family community's approach, MADAM uses *component frameworks* to design applications that can be adapted by reconfiguration. Clemens Szyperski defines a component framework as "a dedicated and focused architecture, usually around a few key mechanisms, and a fixed set of policies for mechanisms at the component level."[12] The component architecture regulates the interaction between components and their environment, defines the role of components, and standardizes interfaces.

In MADAM, a component framework describes a composition of *component types*. We achieve variability by plugging in different component implementations whose externally observable behavior conforms to the type. Each component plugged into a component framework may be an atomic component, or a composite component built as a component framework itself. In this way, we can assemble an application from a recursive structure of component frameworks. Figure 2 shows the component framework architecture for the mobile service technician application (described in the sidebar) using a notation similar to UML 2.0.

In addition to *compositional variability*, which is a coarse-grained variability mechanism, MADAM architecture models also support *parametrization*.[13] We use compositional variability to specify structural and algorithmic system component variability. Parametrization, which uses parameters to modify program variables and be-

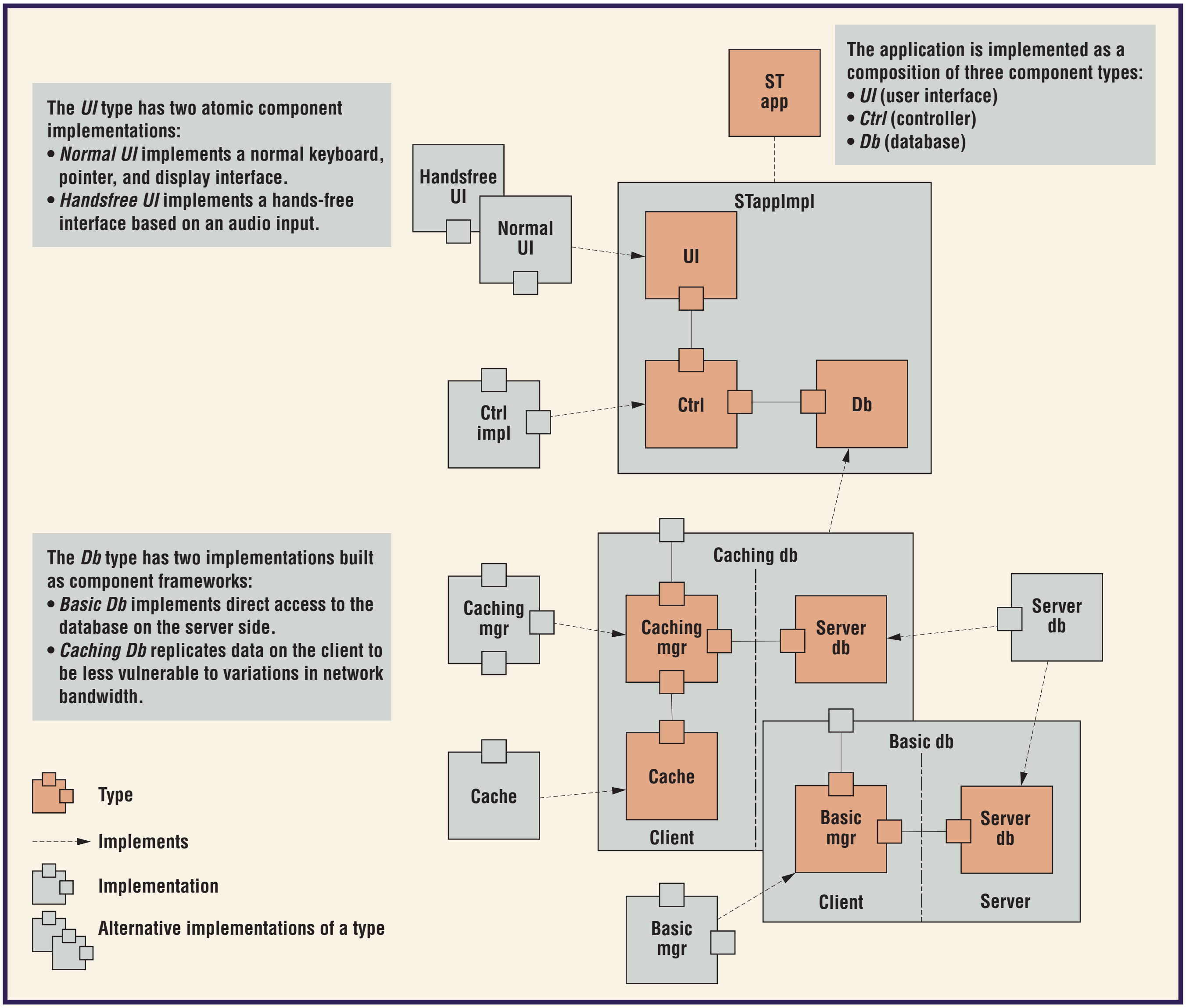

The *UI* type has two atomic component implementations:
• *Normal UI* implements a normal keyboard, pointer, and display interface.
• *Handsfree UI* implements a hands-free interface based on an audio input.
ST app
The application is implemented as a composition of three component types:
• *UI* (user interface)
• *Ctrl* (controller)
• *Db* (database)
Handsfree UI
Normal UI
STappImpl
UI
Ctrl impl
Ctrl
Db
The *Db* type has two implementations built as component frameworks:
• *Basic Db* implements direct access to the database on the server side.
• *Caching Db* replicates data on the client to be less vulnerable to variations in network bandwidth.
Caching mgr
Caching db
Caching mgr
Server db
Server db
Cache
Cache
Client
Basic db
Basic mgr
Server db
Client
Server
Basic mgr
Type
Implements
Implementation
Alternative implementations of a type

Figure 2. The component framework architecture for the mobile service technician example.

havior, is better suited for fine-grained adaptation and can handle a large amount of diversity.

Variant properties

To discriminate between alternative component implementations in MADAM architecture models, we annotate components with *properties*. Properties qualify the services that components offer or need. For example, properties can describe that a user interface component implementation supports hands-free operation. Properties can also specify requirements to system resources such as memory or network needs. Properties are tightly related to context elements. In this way, we can represent the dependencies between component implementations and context.

As components could provide several services and collaborate with multiple components, we must distinguish between service quality on each collaboration. To do so, we use the concept of a *port*. Components have ports through which they interact. Thus, a port represents the component's capability of participating in a specific interaction or service. Ports have attached properties. Services needed from the underlying platform, such as memory, are represented by implicit component ports, and platform-related properties are attached to these ports. Similarly, implicit component ports represent services provided to the user.

We should be able to compare the various component implementations that you can plug into a type. So, the component implementa-

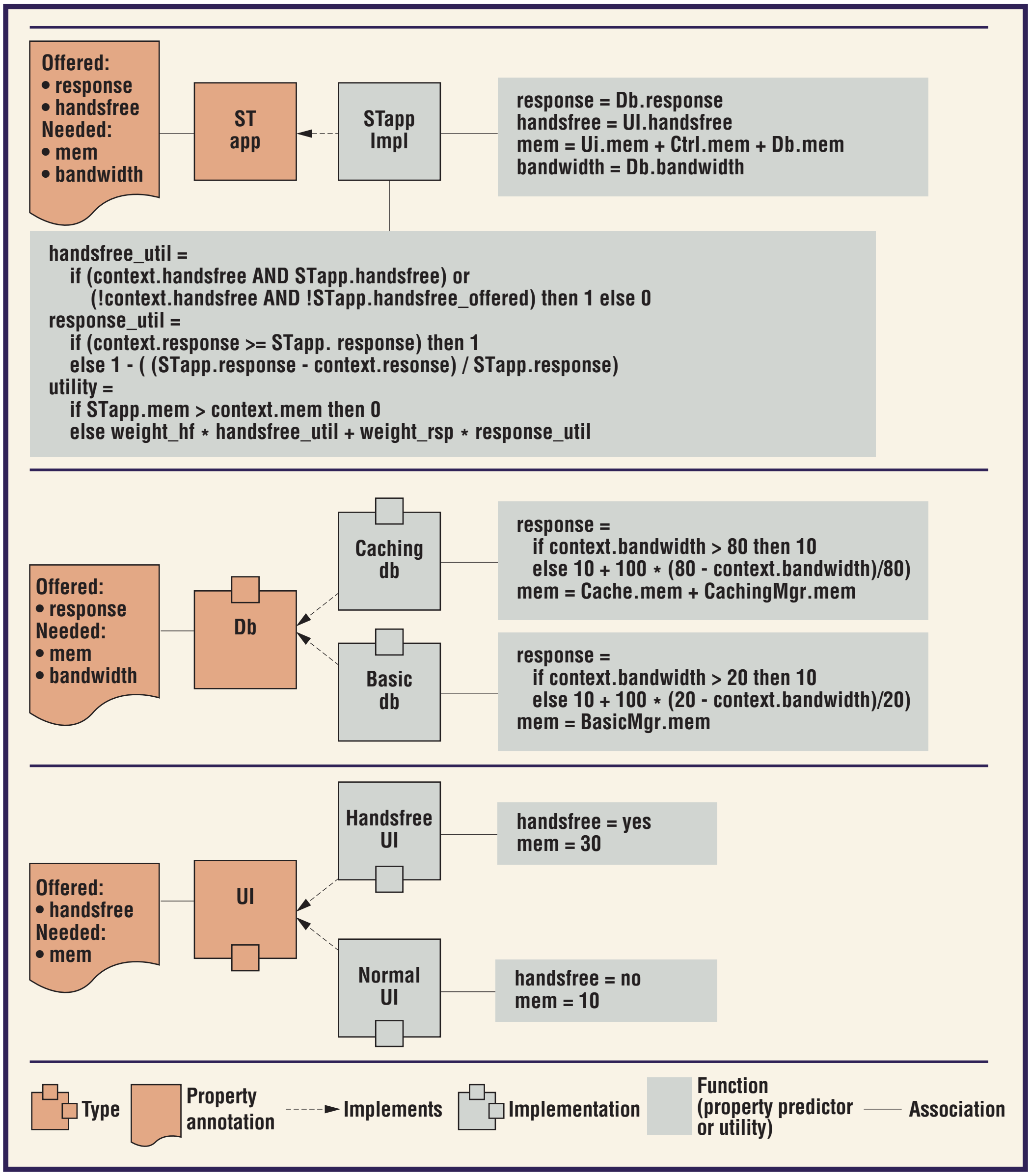

Offered:
• response
• handsfree
Needed:
• mem
• bandwidth
ST app
STapp Impl
response = Db.response
handsfree = UI.handsfree
mem = Ui.mem + Ctrl.mem + Db.mem
bandwidth = Db.bandwidth
handsfree_util =
if (context.handsfree AND STapp.handsfree) or
(!context.handsfree AND !STapp.handsfree_offered) then 1 else 0
response_util =
if (context.response >= STapp. response) then 1
else 1 - ((STapp.response - context.resonse) / STapp.response)
utility =
if STapp.mem > context.mem then 0
else weight_hf * handsfree_util + weight_rsp * response_util
Offered:
• response
Needed:
• mem
• bandwidth
Db
Caching db
response =
if context.bandwidth > 80 then 10
else 10 + 100 * (80 - context.bandwidth)/80)
mem = Cache.mem + CachingMgr.mem
Basic db
response =
if context.bandwidth > 20 then 10
else 10 + 100 * (20 - context.bandwidth)/20)
mem = BasicMgr.mem
Offered:
• handsfree
Needed:
• mem
UI
Handsfree UI
handsfree = yes
mem = 30
Normal UI
handsfree = no
mem = 10
Type
Property annotation
Implements
Implementation
Function (property predictor or utility)
Association

Figure 3. Properties and utility in the framework architecture example.

tions must share a common set of properties. The component type defines that set of properties and their associations with ports.

Associated with implementations, *property predictor functions* predict the properties of the implementations in a given context. We can specify these in various ways including constants—for instance, to express that a specific component implementation requires more than 500 Kbytes of memory—or functional expressions referring to other properties of the component itself or properties of collaborating or inner components. For instance, a component's response time could depend on the available network bandwidth, or the memory a component needs could depend on the memory inner components need.

Figure 3 illustrates property annotations for our framework architecture example. To simplify, we haven't named ports or drawn implicit ports representing the user and the platform. For the sake of readability, we use a pseudocode notation for property prediction functions in the figure. In our current implementation, we specify them in an XML-based notation or as Java code. In the expressions, we might qualify properties with type names for uniqueness.

Selecting variants

To decide what adaptation to perform, we first must decide which application variant best fits the current context. For this, MADAM architecture models describe *utility* functions that assign a scalar value to each possible application variant as a function of application properties in a given context. Adaptation management aims to determine the application variant with the highest utility in any given context (that is, when context changes) and adapt the application accordingly. Figure 3 illustrates the utility function for our framework architecture example.

The architect, not the service user, has the difficult task of specifying utility functions. To let the service user have some level of control of adaptation strategies, the architect specifies utility function as the weighted sum of the utility of each property dimension. The weights reflect the user needs priorities, which the user can assign values to through a suitable UI. For example, the janitor could assign a higher priority to the hands-free property than to the memory-consumption property when working on an installation problem.

MADAM middleware

Figure 4 shows the MADAM middleware architecture. We specify the architecture as a component framework, allowing middleware services to be composed in a flexible manner and thus supporting adaptability of the middleware. For example, we can define different adaptation algorithms.

Runtime models

When an application is launched, the middleware interprets the models the architect specified to produce the *framework architecture model's* runtime representation. When building this model, the middleware identifies all alternative components that it can plug into the component frameworks described by the compile-time models. In a system with few software updates, it probably suffices to build this model on application launch. In a dynamically evolving environment, building should also occur while the application is running. Deriving and evaluating application variants from the framework architecture model is called *planning*.[14] Planning is a recursive process that, for each variation point (component type) the architecture defines, searches for component implementations that fit the variation point. Planning occurs each time the middleware needs to derive new variants for an application.

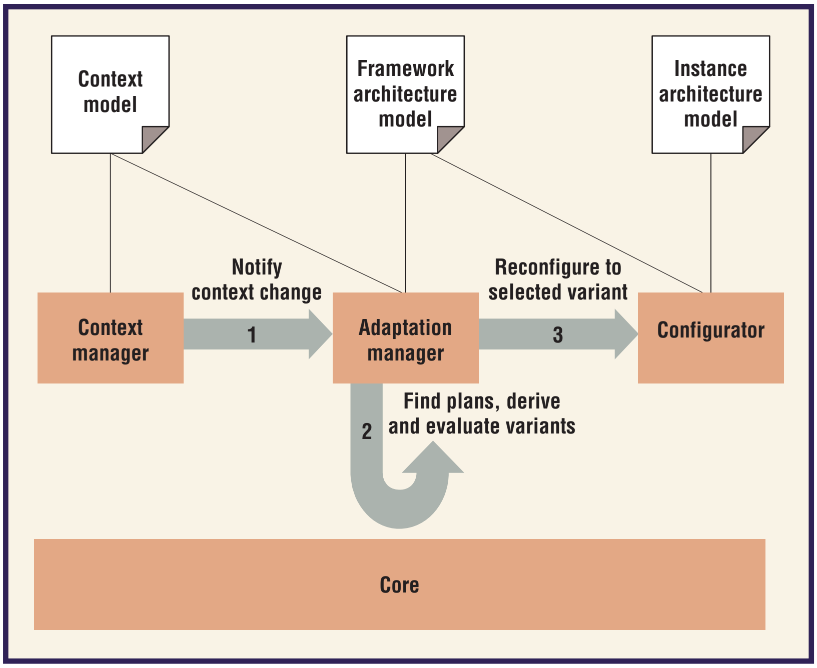

Context model
Framework architecture model
Instance architecture model
Notify context change
Reconfigure to selected variant
Context manager
1
Adaptation manager
3
Configurator
Find plans, derive and evaluate variants
2
Core

Figure 4. The MADAM middleware architecture and runtime models.

In addition to the framework architecture model, the middleware also builds and maintains an *instance architecture model*, which is a model of the running application variant. This model is important during the evaluation of the running application variant and during reconfiguration.

Context change detection

After building the framework architecture model, the middleware can determine the properties of interest for evaluating application variants. As these properties relate to context elements, assigning values to properties requires monitoring the context. We might also need context reasoning, such as aggregation, derivation, and prediction. The Context Manager middleware component is responsible for these functions. It must also provide the Adaptation Manager component with relevant context information when appropriate—that is, when context changes occur. We can consider the deployment of new components as a part of context changes. Such changes would initiate the computation of a new framework architecture model.

Reasoning about context changes

When a context change occurs, the Adaptation Manager must reason about the impact of changes on the application and select an application variant that best suits the current context and user needs. In the current implementation, the Adaptation Manager evaluates the variants as described by the framework architecture model by calculating the utility of each variant in the current context. It also evaluates the current application instance as described by the instance architecture model. If the Adaptation Manager determines that a variant has a better utility than the current instance, reconfiguration takes place.

The Model Driven Architecture approach should include the development of MADAM architecture models.

Application reconfiguration

The Configurator middleware component is responsible for reconfiguring an application. By comparing the application instance and new variant models, the Configurator derives the reconfiguration steps. Reconfiguration might require bringing components into safe state, deleting or replacing component instances, instantiating components, transferring states, and so on.

Operations on components

The building of runtime models and the reconfiguration steps require operations on components. In the MADAM middleware, the Core component provides platform-independent services for managing applications, components, and component instances. This includes operations for publication and discovery of component frameworks and implementations and for loading, unloading, and connecting components. The Core also provides platform-independent access to the execution platform's resources such as memory, CPU, and network information.

Experimentation

We first validated our approach's basic principles by implementing two simple case examples: an information service to support janitor inspections and a videostreaming application. By developing a prototype middleware and case examples, we showed the approach's feasibility and demonstrated interesting adaptive behavior. To further demonstrate the approach's practical applicability and usefulness, we implemented two industrial pilot services in collaboration with Condat and Integrasys, MADAM's industrial partners. We executed these pilot services in a simulated context environment. The pilot implementation encompassed the development of architecture models. It also adjusted the implementation of existing product components to support the reconfiguration interfaces the middleware required. Although these were fairly simple adjustments, defining properties and utility functions isn't as straightforward, and we plan to develop support for this.

The scalability of our current reasoning approach is a concern because evaluating all possible variants might lead to a combinatorial explosion. However, the problem is limited because we apply coarse-grained adaptation on the architectural level, and mobile applications running on small devices tend to be small and fairly simple. To determine the practical limits, we performed experiments with our current prototype middleware using a typical target device example (iPAQ 5550). We found that within 1 second, we could evaluate around 3,000 variants. Our two industrial pilot services have many fewer variants than this. We also observed that in the pilots, the number of relevant variants is much lower than the number of variants obtained by exploring the whole variation set.

If we use parametrization more extensively than in the current pilots, scalability can still pose a problem. You can use parametrization to effectively model and implement variability, but it can also lead to a large set of variants differing only marginally in properties. Another issue affecting scalability is reasoning on a set of concurrently running applications competing for the same resources.

As we specified the middleware as a component framework, we have the flexibility to experiment with different approaches for handling scalability—for example, by replacing the Adaptation Manager component. We are researching alternative approaches to improve the computational efficiency of evaluating variants, including using architectural constraints, domain-specific heuristics, historical data, and pre-runtime evaluation to limit the runtime search space.

The architect task

So, how will our approach—based on middleware and runtime representation of architecture models—appear to architects? We envision that the Model Driven Architecture approach should include the development of MADAM architecture models. MDA approaches actively use the architecture model throughout the software's whole life cycle, and transformations from platform-independent models automate part of the implementation. In our case, we are designing a UML profile for self-adapting systems, which the architect will use to model the architecture. The resulting design-time architecture models will include descriptions of both the variability and property annotations. From these UML models, transformations will allow the generation of compile-time architecture models as well as partial code for the component implementations.

To facilitate the architect's task in defining properties and utility function, we propose to develop a simulation environment for fine-tuning properties. Also, we intend to define guidelines for modeling mobile applications' variability. To reduce the complexity of adaptation reasoning, the architect must control the number of variants. Especially, the architect should be aware that complexity increases through parametrization when adding new parameters and parameter values.

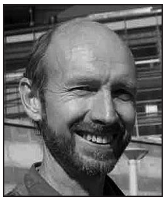
Besides supporting the development of adaptive applications, utilizing the architecture models at runtime also has positive side effects. In more traditional development, it is a challenge to ensure that the implementation follows the architecture and that the architectural description and implementation stay in sync as the system evolves. Some developers might even ask, is it worth the effort? MDA partially solves this problem because it supports transformations, allowing developers to derive part of the implementation directly from the design. When directly applying the architecture models at runtime as in our approach, we bring support for consistency one step further. As this increases the payback for making the models, this should encourage developers to make them in the first place.

Acknowledgments

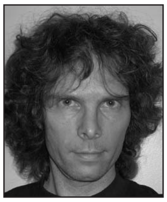
The European project MADAM (EU IST 4159) and the Research Council of Norway's FAMOUS (Framework for Adaptive Mobile and Ubiquitous Services) and QuA (Quality of Service Aware Component Architecture) research programs support our work.

About the Authors

Jacqueline Floch is a research scientist at SINTEF ICT (The Foundation for Scientific and Industrial Research at the Norwegian Institute of Technology, Information and Communication Technology). Her research interests include software architectures, model-driven service engineering, and service validation. She received her Dr. Ing. in telematics from the Norwegian University of Science and Technology. Contact her at SINTEF ICT, NO-7465 Trondheim, Norway; jacqueline.floch@sintef.no.



Svein Hallsteinsen is a senior scientist at SINTEF ICT (The Foundation for Scientific and Industrial Research at the Norwegian Institute of Technology, Information and Communication Technology). His research interests include component and service-oriented software architecture, software reuse, software product lines, and context-awareness and self-adaptation for mobile and ubiquitous computing. He received his Siv. Ing. from the Norwegian Institute of Technology. Contact him at SINTEF ICT, NO-7465 Trondheim, Norway; svein.hallsteinsen@sintef.no.



Erlend Stav is a research scientist at SINTEF ICT (The Foundation for Scientific and Industrial Research at the Norwegian Institute of Technology, Information and Communication Technology) and PhD student at the Norwegian University of Science and Technology. His research interests include software architecture, component software, and end-user development. Contact him at SINTEF ICT, NO-7465 Trondheim, Norway; erlend.stav@sintef.no.

Frank Eliassen is a professor in distributed systems at the University of Oslo and a senior researcher at Simula Research Laboratory. His research interests include distributed and parallel on-the-fly multimedia streaming computing, component and service-oriented architectures, reflective and adaptive middleware, context-aware and adaptive services and applications, middleware for sensor networks, and QoS management. He received his Cand. Real. in computer science from the University of Tromsø. He is a member of the ACM. Contact him at Simula Research Laboratory, PO Box 134, NO-1325 Lysaker, Norway; frank@simula.no.

Ketil Lund is a postdoc at Simula Research Laboratory. His research interests include component architectures, quality of service, middleware, and distributed multimedia systems. He received his Dr. Scient. in informatics from the University of Oslo. Contact him at Simula Research Laboratory, PO Box 134, NO-1325 Lysaker, Norway; ketillu@simula.no.

Eli Gjørven is a PhD student at Simula Research Laboratory. Her research interests include component architectures, quality of service, and middleware. She received her Cand. Scient. in computer science from the University of Oslo. Contact her at Simula Research Laboratory, PO Box 134, NO-1325 Lysaker, Norway; eligj@simula.no.

References

1. P. Oreizy et al., "Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems,* vol. 14, no. 3, 1999, pp. 54–62.
2. D. Garlan et al., "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *Computer*, vol. 37, no. 10, 2004, pp. 46–54.
3. M. Satyanarayanan, "Fundamental Challenges in Mobile Computing," *Proc. 15th ACM Symp. Principles of Distributed Computing*, ACM Press, 1996, pp. 1–7.
4. M. Satyanarayanan, "Pervasive Computing: Vision and

Challenges," *IEEE Personal Comm.*, vol. 8, no. 4, 2001, pp. 10–17.

5. S. Hallsteinsen, J. Floch, and E. Stav, "A Middleware Centric Approach to Building Self-Adapting Systems," *Software Eng. and Middleware*, LNCS 3437, Springer, 2004, pp. 107–122.
6. C. Mascolo, L. Capra, and W. Emmerich, "Mobile Computing Middleware," *Advanced Lectures on Networking*, LNCS 2597, Springer, 2002, pp. 20–58.
7. P. Grace, G.S. Blair, and S. Samuel, "Middleware Awareness in Mobile Computing," *Proc. 23rd Int'l Conf. Distributed Computing Systems Workshops*, IEEE Press, 2003, pp. 382–387.
8. M. Roman, F. Kon, and R.H. Campbell, "Reflective Middleware: From Your Desk to Your Hand," *IEEE Distributed Systems Online*, vol. 2, no. 5, 2001.
9. J. Bosch, *Design And Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, Addison-Wesley, 2000.
10. R. van Ommering et al., "Koala Component Model for Consumer Electronics Software," *Computer*, vol. 33, no. 3, 2000, pp. 78–85.
11. *Proc. Configuration Workshop at the 16th European Conf. Artificial Intelligence* (ECAI 04), 2004, www.ifi.uni-klu.ac.at/Conferences/ECAI04-Configuration-Workshop/CFGProceedings.pdf.
12. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed., Addison-Wesley, 2002.
13. P.K. McKinley et al., "Composing Adaptive Software," *Computer*, vol. 37, no. 7, 2004, pp. 56–64.
14. R. Staehli, F. Eliassen, and S. Amundsen, "Designing Adaptive Middleware for Reuse," *Proc. 3rd Workshop Reflective and Adaptive Middleware*, ACM Press, 2004, pp. 189–194.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.