IEEE distributed systems Expert-authored articles and resources ONLINE
July 2008 (vol. 9, no. 7), art. no. 0807-mds2008070001

Published by the IEEE Computer Society

**Self-Adaptive Applications**

# Architectural Constraints in the Model-Driven Development of Self-Adaptive Applications

**Mohammad Ullah Khan, Roland Reichle, and Kurt Geihs • University of Kassel, Germany**

*A component framework supports adaptation through application variability. The adaptation decision is made at runtime by resolving the variation points and computing the utility of all application variants.*

These days, many people carry a mobile device such as a personal digital assistant (PDA), smart phone, or laptop wherever they go. These devices usually have limited resources in terms of battery power, memory, and CPU capacity. They also often operate in vastly diverse and changing environments—communication bandwidth fluctuates, error rate changes, battery capacity decreases, or a noisy environment obliterates the effect of sound output. The performance and quality of applications running on those devices depend on the resource constraints and the dynamically changing properties of the execution context. To maintain their usefulness, such applications must automatically adapt to their current operational context.

Our overall goal is to facilitate the development of self-adaptive component-based applications. Here, we focus on dynamic compositional adaptation at runtime. Context dependencies and application variability are specified as part of the application architecture. An application component can be hierarchically decomposed into other components. Each component might have a number of different realizations that provide the same basic functionality but differ in their extra-functional characteristics, such as resource requirements and context dependencies. Therefore, different realizations for a particular component introduce variation points in the architecture of an application.

Following the model-driven architecture (MDA)-based development process, we build an application adaptation model in UML 2.0 that is transformed into source code by means of a model-to-text transformation. The source code is packaged and deployed to the adaptation middleware. If a context change occurs during application execution, the adaptation middleware computes, on the fly, all possible application variants and evaluates their utilities with respect to the current context situation (see www.intermedia.uio.no/display/madam/Home). We select and instantiate the best variant.[1]

Resolving all possible variation points can effectively create a huge number of different application variants, all of which must be evaluated for utility. However, not all computed variants are actually feasible. For example, selecting a particular realization for a component might imply a certain realization for another component. Likewise, certain component realizations might be incompatible with each other. Furthermore, the potential combinatorial explosion of variants can lead to a scalability problem requiring too much computational effort for a resource-scarce mobile device.[2] Filtering infeasible combinations helps avoid such compositions and reduces the number of variants to consider, improving scalability.

Enhancing our previous work,[1] we present here a convenient approach consisting of modeling, source code generation, and runtime evaluation of architectural constraints as part of the application variability and adaptation model.

## Proposed modeling approach

We base our solution on a feature model.[3,4] A *feature* describes a distinguishable visible aspect, quality, or characteristic of the realization. We apply constraints (that is, invariants)[5] to the features.

Whereas a feature corresponds to a characteristic property of a component realization, *invariants* describe interdependencies with regard to the resolution of variation points based on the features. The adaptation manager checks these invariants at runtime when computing the application variants.

### Modeling notation

We've developed a new UML 2.0 profile for modeling self-adaptive applications that contains modeling elements for architectural constraints (see figure 1).

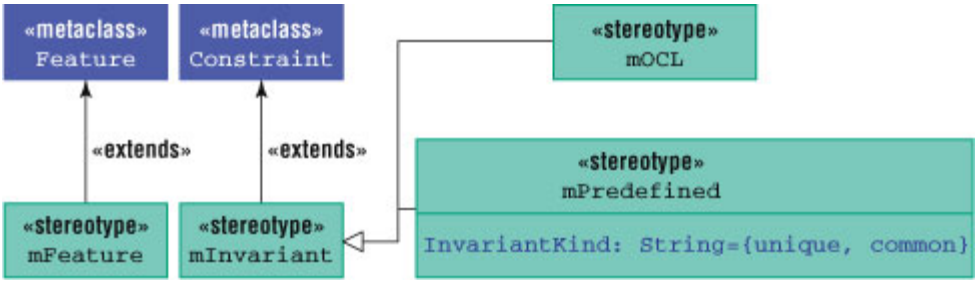


**Figure 1. Modeling notation for architectural constraints.**

The «mFeature» stereotype extends Class and Feature metaclasses to model features. The «mInvariant» stereotype extends UML Constraint, and it is either anObject Constraint Language constraint («mOCL») or some predefined («mPredefined») string represented by the "InvariantKind" tag. The tag can have two values: "unique" and "common." A "unique" value indicates mutual exclusiveness. The component realization carrying this feature is unique within a composition with respect to all components having the same feature with the "unique" invariant. Likewise, a "common" value demands that all component realizations having this feature in a composition must provide the same feature. In addition to these two predefined invariants, we also foresee the facility of specifying invariants using the OCL. Such invariants can support constraints on characteristic properties of the execution system, other related components, and parameters. We foresee modeling and transformation tools interpreting and checking for well-formedness of OCL constraints. However, interpreting OCL invariants by the middleware is a work in progress. Here, we provide only a rough idea of how OCL constraints can make our architectural constraints model more expressive.

### Modeling example

We explain the modeling approach using simplified and partial model extracts from the `SatMotion` application of the MADAM (Mobility and Adaptation Enabling Middleware) project (see www.intermedia.uio.no/display/madam/Home). Building the model is a step-by-step procedure that involves identifying features, building a feature hierarchy, and then associating features with components.

*Identifying features.* In software product families, features usually result from requirements analysis expressed as software functionalities and usually apply to a certain domain. In our approach, application variants shouldn't necessarily provide different functionalities; rather, they make sure that the application remains useful in a changing context. Therefore, when designing an adaptive application, the developer has in mind a set of operating modes applicable in different context situations. These operating modes—for example, using a communication link or operating in offline mode—are also reflected by the provisioning of component realizations corresponding to the different modes and therefore provide a distinguishable characteristic of the component realizations. Such characteristics, if they require or exclude the same characteristic in other component realizations, are identified as features. In some cases, the resulting set of features doesn't allow filtering out all infeasible application variants. The application developer must then carefully analyze the application architecture, the component framework, the available components, and their interdependencies with regard to the resolution of variation points to identify additional features. Figure 2 shows a composite

structure diagram, where the composition consists of four components (marked with the «mComponent» stereotype): `Controller`, `MathProcessor`, `Recorder`, and `UserInterface`.

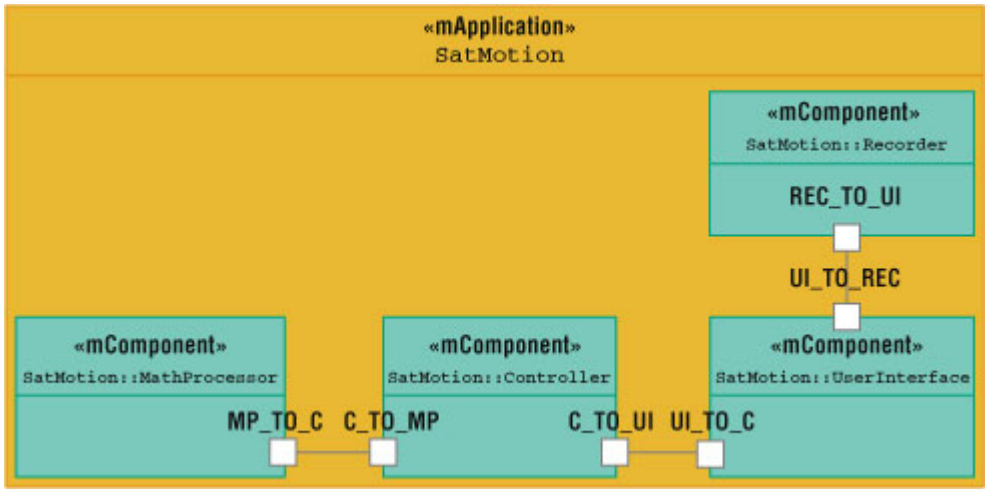


**Figure 2. A composition of components for the `SatMotion` application.**

Each component might have a number of different variants (that is, realization options). For example, the `UserInterface` has variants called `OneWayUI`, `TwoWayUI`, and `PlayBackUI` (see figure 3).

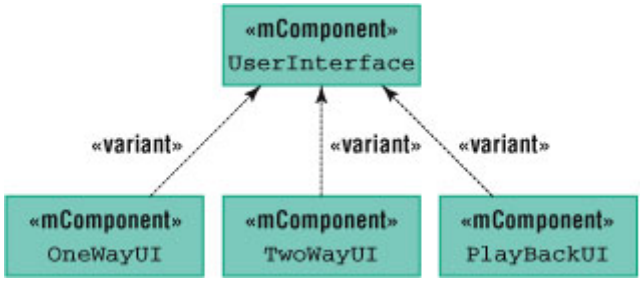


**Figure 3. Different possible variants for the `UserInterface` component.**

The `Controller` component might also have similar variants, and a constraint might be that "one-way controller is compatible only with one-way user interface." In addition, another constraint might be, for example, "a composition having a one-way controller must not have any variant of the math processor." Thus, we identify two features: a feature called `TypeMatching`, with its variants `OneWayType`, `TwoWayType`, and `PlayBackType`, as well as a feature called `CtlrMPIncompatibility`, with its variant `CtlrMPOWIncompatibility`.

*Building a feature hierarchy.* Like the components of the architecture model, each realizing feature is considered abstract and can also have different realizing variants (see figure 4).

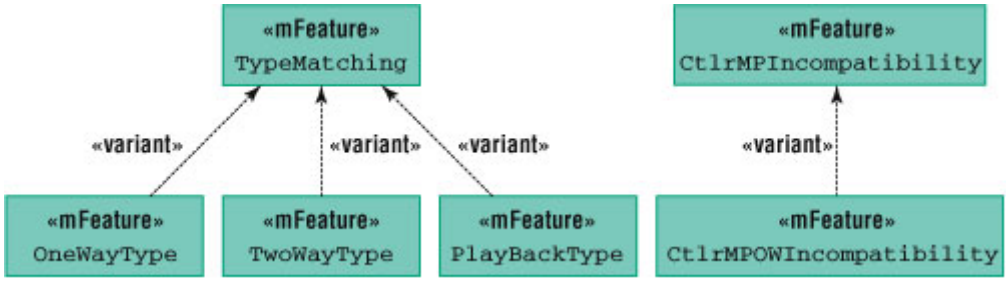


**Figure 4. Part of a feature hierarchy.**

Therefore, the features are arranged in a hierarchical manner, leading to a feature hierarchy. Different hierarchy levels of the feature model and the architectural model likely correspond, as features associate with the components and the feature variants with the corresponding component realizations or variants.

*Associating features with components*. Associating features with the components in the composite structure diagram is the next step. Adding the two features to the composite structure in figure 2 results in the composite structure diagram in figure 5.

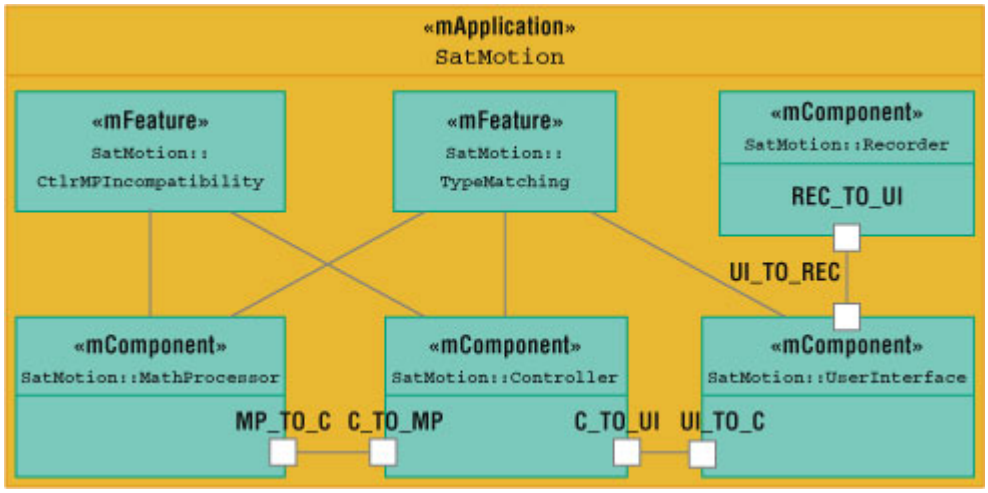


**Figure 5. Adding two features, `TypeMatching` and `CtlrMPIncompatibility`, enhances the `SatMotion` application architecture in figure 2.**

The composite structure diagram in figure 5 indicates that the adaptation middleware should consider the descendants of the `TypeMatching` feature for `UserInterface` at its variation points; for `MathProcessor`, it should consider the variants of `CtlrMPIncompatibility` feature. Both of these features affect `Controller`. With the component variants at this level, the corresponding feature variants are associated (see figure 6). When variation points are resolved for the model in figure 5, the feature associations in figure 6 dictate that if we choose a `OneWayUI` for `UserInterface`, then for `Controller`, we must choose `OneWayController` (constrained by the "common" value of the `InvariantKind` tag). On the other hand, if we choose `OneWayController` for `Controller`, then we can choose neither `LSMathProcessor` nor `HSMathProcessor`, as the "unique" value dictates. The OCL invariant dictates that we can select a realization of this component only if it consumes less than 10 percent of the available system memory and the speed of the realized `Controller` (see figure 5 for the association between `UserInterface` and `Controller`) is greater than 1000 units.

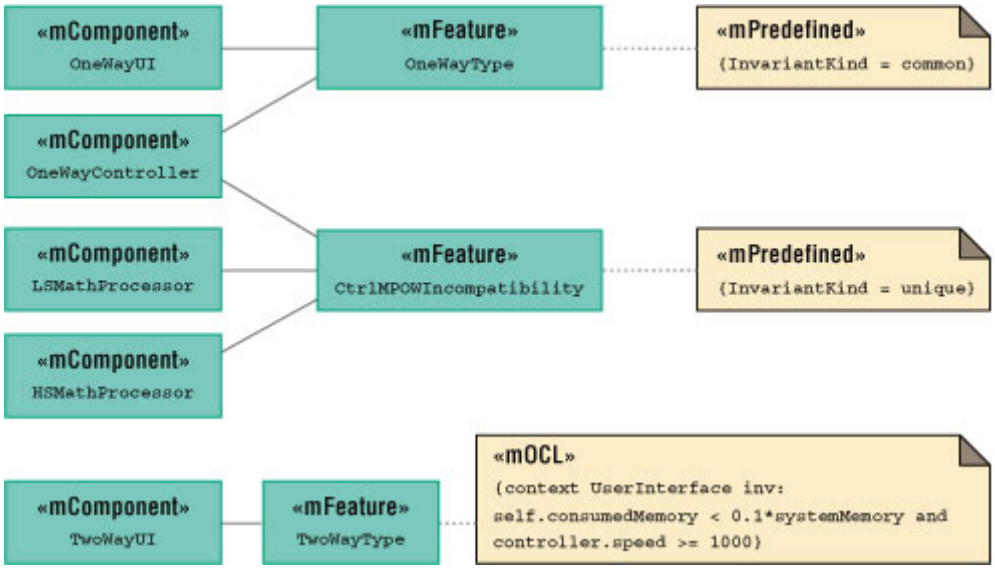


**Figure 6. Association of features with the variants of the components used in figure 5.**

Besides compositional adaptation in the application architecture, we also support parameter adaptation by allowing the adaptation of parameters with discrete value ranges.[6] By associating features not only to whole components but also to certain parameter settings, our approach can also extend to supporting such an adaptation technique.

## Source code generation and runtime evaluation

The built model is automatically transformed into source code using transformation tools. The generated code is then compiled and packaged to deploy on the middleware.

### Source code generation

In the comprehensive tool chain realm (see figure 7), we generate source code using the MOFScript (Meta-Object Facility) model-to-text transformation tool (www.eclipse.org/gmt/mofscript). The whole transformation procedure is integrated within the Eclipse environment. A transformation using MOFScript requires the model to be expressed in a format that conforms to the Eclipse UML2 meta-model, which is a subset of the Object Management Group UML 2.0 meta-model. Therefore, an Eclipse Modeling Framework-based modeling tool such as Omondo, IBM Rational Software, or Borland Together Architect is a direct choice for such a transformation. For Enterprise Architect, which doesn't produce UML2 output, we've developed an XSLT (Extensible Stylesheet Language Transformations) stylesheet that transforms the UML 2.0 model (in XML Metadata Interchange, or XMI, exported from Enterprise Architect) to UML2 (XMI) format, which MOFScript can then use as input.

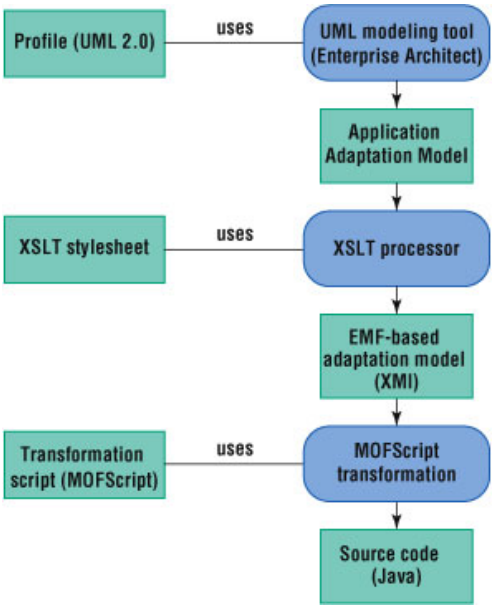


**Figure 7. The tool chain for the modeling and transformation.**

Figure 8 shows a code fragment generated for the architectural constraint specification of the composition shown in figures 5 and 6.

```
//Feature specification for the composition of Figure 5
Vector featureSpec_StandardComposition = new Vector();

Vector featureAssoc1Components= new Vector();
featureAssoc1Components.add("MathProcessor");
featureAssoc1Components.add("Controller");
FeatureAssociation featureAssoc1 = new FeatureAssociation(
            "SCAssoc1", "CtlrMPIncompatibility", featureAssoc1Components);
featureSpec_StandardComposition.add(featureAssoc1);

Vector featureAssoc2Components= new Vector();
featureAssoc2Components.add("MathProcessor");
featureAssoc2Components.add("Controller");
featureAssoc2Components.add("UserInterface");
FeatureAssociation featureAssoc2 = new FeatureAssociation(
            "SCAssoc2","TypeMatching", featureAssoc2Components);
featureSpec_StandardComposition.add(featureAssoc2);

standardCompositionPlan.setFeatureSpec(featureSpec_StandardComposition);

// Feature specification at the bottom-most level of the variability hierarchy
// for the OneWayController component (Figure 6)
BLUEPRINT_PLANS[7] = new BlueprintPlan(SatMotionApplication_Controller,
            new HashMap(), SatMotionApplication_OneWayController_BP){
      //.........
      HashMap featureSpec = new HashMap();
      featureSpec.put("OneWayType", new Feature("OneWayType",
            "TypeMatching", Feature.COMMON_CONSTRAINT));
      featureSpec.put("CtrlMPOWIncompatibility", new Feature("CtrlMPOWIncompatibility"
            "CtrlMPIncompatibility", Feature.UNIQUE_CONSTRAINT));
      BLUEPRINT_PLANS[7].setFeatureSpec(featureSpec);
};
```

**Figure 8. Generated source code for the architectural constraints.**

A vector stores the feature specification for a particular component in a composition. Corresponding to the figure 5 model, the `CtlrMPIncompatibility` feature associates with the components `Controller` and `MathProcessor`, while the `TypeMatching` feature also associates with `UserInterface`. We create a blueprint plan for each of the components at the lowest level of the variability hierarchy. The second half of figure 8 shows the corresponding source code. Our current middleware implementation requires associating predefined invariants with features only at the lowest level of the hierarchy. However, the modeling approach isn't limited to such restrictions.

### Runtime evaluation

Before we calculate the utility of a particular application configuration, we evaluate its architectural constraints by the middleware at runtime to ensure its feasibility. In figure 9, a code fragment from the middleware implementation gives a rough idea of the evaluation process.

```
public static boolean check(Configuration configuration){
 return check(configuration, new Vector(), new HashMap());
}
public static boolean check(Configuration configuration, Vector typeList,
   HashMap evalResults){

 IPlan plan = configuration.getPlan();

 if (plan instanceof AtomicPlan){
  AtomicPlan blueprint = (AtomicPlan) plan;
  Vector featureSpec = blueprint.getFeatureSpec();
  evaluateFeaturesForTypes(typeList, featureSepc, evalResults);
  return true;
 }else if (plan instanceof CompositionPlan){
  CompositionPlan compPlan = (CompositionPlan) plan;
  Vector featureSpec = compPlan.getFeatureSpec();

  ComponentTypes[] componentTypes = compPlan.getComponentTypes();
  HashMap resultsForComponentTypes = new HashMap();

  for(int i = 0; i < componentTypes.length; i++){

  Vector featureTypeList = buildFeatureTypeList(typeList, featureSpec,
    componentTypes[i]);
    HashMap compTypeEvalResults = new HashMap();

    if (!check(configuration.getSubConfiguration(componentTypes[i], featureTypeList,
      compTypeEvalResults))
     return false;
    else{
     resultsForComponentTypes.put(componentTypes[i].getName(), compTypeEvalResults);
    }
  }

  //Check all FeatureTypeAssociations
  for(int i = 0; i < featureSpec.size(); i++){

    if (!checkAllCommonConstraints(featureSpec.get(i), resultsForComponentTypes))
     return false;
    if (!checkAllUniqueConstraints(featureSepc.get(i), resultsForComponentTypes))
     return false;

  }

  //Evaluate all FeatureTypes in List
  evaluateFeaturesForTypesComposite(typeList, featureSpec, resultsForComponentTypes,
    evalResults);
     return true;
  }
  else
     return true;
}
```

**Figure 9. Middleware code for checking a configuration's architectural constraints.**

We implement the method recursively corresponding to the recursive structure of each component, which can be realized through either a composition or an atomic component. When we check a configuration, we first retrieve its plan specifying the realization details of a component and containing the feature specifications. Then we check whether the component is realized through an atomic realization or a composition. In the case of an atomic realization, just the feature realizations (and invariants) for the list of given features are stored in `evalResults`. It always returns "true," because we can't perform architectural constraint checking at the atomic level. In the case of a component composition, we create a list of features for each involved component, combining the list for the previous recursion level with the features contained in the feature specifications at the current level.

We then evaluate all feature realizations for this list and check the architectural constraints at the next recursion level by recursively calling the method for each of the subconfigurations. For each involved component realization, the evaluation results are stored in a HashMap. We use these results to perform the architectural constraints checking at this recursion level. If all architectural constraints are met, then `evalResults` stores the evaluation results for the given list of features. If an architectural constraint isn't met at any recursion level, the method returns "false." We reconfigure after selecting a particular configuration of the application to run. Therefore, the sequence of evaluation of the architectural constraints doesn't affect the reconfiguration procedure.

### Evaluation of the adaptation performance

Evaluating architectural constraints incurs additional computing cost. At the same time, it reduces adaptation time because we avoid checking resource requirements and calculating the utilities of infeasible configurations. We tested the approach with a pilot application, which would have 213 different configurations; only 54 of them must be considered when we apply appropriate architectural constraints. We measured adaptation time on a PDA and a PC (see table 1). From this table, it's evident that the overall adaptation process gets much faster, especially for resource-scarce devices.

**Table 1. Performance evaluation of the adaptation process.**

| Measured performance metric | Device | Without architectural constraint (sec) | With architectural constraint (sec) | Improvement (%) |
|---|---|---|---|---|
| Adaptation time | PDA | 5.74 | 4.71 | 20 |
| | PC | 1.898 | 1.838 | 3 |

## Related work

Many works address variability model specification.[7–9] For example, Marco Sinnema and his colleagues[7] address three types of dependencies constraining a variation point as well as the relationships among the dependencies. However, the dependencies must be modeled separately, which incurs big overhead, making the use of this concept in a model-driven development approach quite cumbersome.

Different Architectural Description Languages have been applied to specify architectural constraints. In ADLs such as Acme,[10] component- or subsystem-wide architectural constraints are expressed in a first-order predicate logic language. In ADLs such as C2, in which connectors are first-class modeling elements, constraints can be placed in connectors, enforcing a set of policies in the components attached to it.[11]

Thais Batista and colleagues[12] tackle the problem of software maintenance and reconfiguration, where architectural invariants constrain the reconfiguration to maintain consistency. In the Fractal Component Model of ConFract,[13] contracts capture assumptions about the functional and extra-functional properties of components that must be maintained upon invoking an interface method.

Detlef Streitferdt and colleagues define the feature concept as an important, distinguishable, user-visible aspect, quality, or characteristic of a software system.[3] Features are organized hierarchically to describe a system. Krzysztof Czarnecki and Chang Hwan Peter Kim[4] have done significant work on defining and using the feature model. They have developed an Eclipse plug-in to build feature models, and they can use OCL to specify constraints and verify them against well-formedness.

Unlike many other researchers,[10–13] we targeted the MDA-based development approach. Therefore, we use UML as the modeling language. The feature hierarchy concepts we mentioned earlier[3,4] fit well with the component variability model that considers crosscutting aspects of the architecture components as features. Our approach doesn't alter the component variability model; rather, we add features and constraints to it. In contrast to Czarnecki and Kim,[4] the feature hierarchy itself doesn't contain constraints; we apply those later on the basis of components' characteristics. It's even possible to introduce new variation points at runtime, so our approach—which can also evaluate features at runtime—is novel in that it facilitates adaptation, which is unanticipated at design time.

We've successfully applied our modeling technique for architectural constraints in the development of two adaptive applications in the MADAM project.

From a modeling point of view, we add a rather small amount of complexity to the architecture model of the application. From a performance point of view, we've demonstrated that the approach can effectively filter out infeasible application variants and significantly improve the performance of the adaptation process. It therefore improves scalability, which can severely limit the exploitation of variability concepts in adaptive applications, particularly for resource-scarce mobile devices.

We've already enhanced our modeling technique by permitting OCL constraints in addition to the presented predefined "common" and "unique" constraints. However, the current middleware prototype (available as open source from www.intermedia.uio.no/display/madam/Home) so far supports only the two predefined constraints. We're enhancing the middleware in another research project called MUSIC (Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments) to support these improved specifications.

**Acknowledgments**

We thank all partners of the MADAM and MUSIC projects for their contributions and the anonymous reviewers for their valuable advice.

**References**

1. K. Geihs et al., "Modeling of Component-Based Self-Adapting Context-Aware Applications for Mobile Devices," *IFIP Working Conf. Software Engineering Techniques*, Springer, 2006, pp. 85–96.
2. M. Alia et al., "A Component-Based Planning Framework for Adaptive Systems," *The 8th Int'l Symp. Distributed Objects and Applications* (DOA), Springer, 2006, pp. 1686–1704.
3. D. Streitferdt et al., "Configuring Embedded System Families Using Feature Models," *Proc. 6th Int'l Conf. Net.Objectdays,* TranSIT, 2005, pp. 339–362.
4. K. Czarnecki and C.H.P. Kim, "Cardinality-Based Feature Modeling and Constraints: A Progress Report," *OOPSLA 05 Int'l Workshop Software Factories* (online proceedings), 2005, http://softwarefactories.com/workshops/OOPSLA-2005/Papers/Czarnecki.pdf.

5. B. Ahlgren et al., "Invariants: A New Design Methodology for Network Architectures," *Proc. SIGCOMM 2004 Workshop Future Directions in Network Architecture* (FDNA 04), ACM Press, 2004, pp. 65–70.
6. R. Reichle, M.U. Khan, and K. Geihs, "How to Combine Parameter and Compositional Adaptation in the Modeling of Self-Adaptive Applications," *PIK (Praxis der Informationsverarbeitung und Kommunikation)*, special issue on modeling of self-organizing systems, vol. 31, no. 1, 2008, pp. 34–38.
7. M. Sinnema et al., "COVAMOF: A Framework for Modeling Variability in Software Product Families," *Proc. Third Software Product Line Conference*, Springer, 2004, pp. 197–213.
8. M. Clauss, "Generic Modeling using UML Extensions for Variability," *Proc. OOPSLA 2001, Workshop on Domain Specific Visual Languages*, Jyväskylä Univ. Printing House, 2001, pp. 11–18.
9. S. Thiel and A. Hein, "Systematic Integration of Variability into Product Line Architecture Design," *Proc. 2nd Int'l Conf. Software Product Lines* (SPLC-2), LNCS 2379, Springer, 2002, pp. 130–153.
10. D. Garlan, R.T. Monroe, and D. Wile, "Acme: Architectural Description of Component-Based Systems," *Foundations of Component-Based Systems,* G.T. Leavens and M. Sitaraman, eds., Cambridge Univ. Press, 2000, pp. 47–68.
11. P. Oreizy, D.S. Rosenblum, and R.N. Taylor, "On the Role of Connectors in Modeling and Implementing Software Architectures," Feb. 1998; www.isr.uci.edu/architecture/papers/TR-UCI-ICS-98-04.pdf.
12. T.V. Batista, A. Joolia, and G. Coulson, "Managing Dynamic Reconfiguration in Component-Based Systems," *2nd Int'l Workshop Software Architecture*, Springer, 2005, pp. 1–17.
13. P. Collet et al., "A Contracting System for Hierarchical Components," *8th Int'l Symp. Component-Based Software Engineering*, Springer, 2005, pp. 187–202.

**Mohammad Ullah Khan** is a PhD student and member of the Distributed Systems Research Group of the University of Kassel in Germany. His research interests include model-driven development and self-adaptive context-aware systems. Khan received an M.Sc. degree in information technology from the University of Stuttgart. Contact him at khan@vs.uni-kassel.de.

**Roland Reichle** is a PhD student and member of the Distributed Systems Research Group at the University of Kassel in Germany. His research interests are context-aware self-adaptive systems, model-driven development of collaborative behavior in heterogeneous teams of autonomous systems, and autonomous soccer robots. Reichle received a diplom-informatiker degree from the University of Ulm. Contact him at

reichle@vs.uni-kassel.de.

**Kurt Geihs** is a professor in the department of electrical engineering and computer science at the University of Kassel in Germany. His research and teaching interests include distributed systems, operating systems, networks, and software technology. His current research projects focus on self-adaptive distributed systems, self-management in service-oriented architectures, and autonomous mobile robots. He received a PhD in computer science from the Aachen University of Technology. Contact him at geihs@vs.uni-kassel.de.

**Cite this article:**
Mohammad Ullah Khan, Roland Reichle, and Kurt Geihs, "Architectural Constraints in the Model-Driven Development of Self-Adaptive Applications," *IEEE Distributed Systems Online*, vol. 9, no. 7, 2008, art. no. 0807-o7001.