COVER FEATURE

AUTONOMIC COMPUTING THROUGH REUSE OF VARIABILITY MODELS AT RUNTIME: THE CASE OF SMART HOMES

Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano,
Polytechnic University of Valencia, Spain

By automating tasks such as installation, healing, and updating, autonomic computing simplifies system operation at the cost of increasing internal complexity. A proposed approach for developing autonomic systems in smart homes reuses variability models at runtime to provide a richer semantic base for decision making.

Autonomic computing (AC) envisions computing environments that evolve without the need for human intervention. A system with autonomic capabilities installs, configures, tunes, and maintains its own components at runtime. Defining appropriate abstractions and models for understanding, controlling, and designing autonomic behavior is a challenge at the heart of AC.[1] Several research efforts have successfully produced major autonomic capabilities, such as self-configuration using reinforcement learning and self-adaptation using Petri-net-based[2] and adaptation models.[2,3] However, enhancement with these autonomic capabilities increases the resulting systems' complexity and production methods.

In contrast to, but complementing, previous studies, our work focuses on mass-production environments such as those used to create cars or houses, where production costs are a major constraint. Reducing production costs comes at the expense of limiting the level of detail in customization—for example, when buying a car, you can choose the color but only from a limited palette. The tradeoff is acceptable in these domains because in general the focus is on covering the average demand, not the needs of each individual.

Variability models have proven useful in mass-production domains to maximize reuse when developing a set of similar software systems (a system family).[4] Because common system parts—components, models, frameworks, documentation, and so on—are clearly identified, it is possible to detect reuse opportunities for each new development.

Our research shows that autonomic behavior can be achieved by leveraging variability models at runtime. In this way, the modeling effort made at design time is not only useful for producing the system but also provides a richer semantic base for autonomic behavior during execution. The use of variability models at runtime brings new opportunities for autonomic capabilities by reutilizing the efforts invested at design time.

Our proposed approach has two aspects:

 Published by the IEEE Computer Society

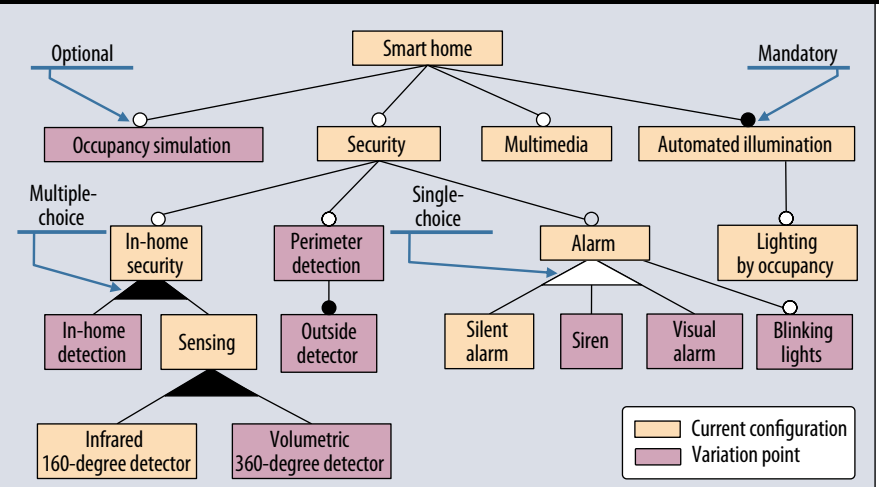

Figure 1. Smart home feature model. Features are hierarchically linked in a tree-like structure through variability relationships.

- *Reuse of design knowledge to achieve AC.* We reuse the knowledge previously captured in variability models to describe the variants in which a system can evolve. In response to changes in the context, the system itself can query these models to determine the necessary modifications to its architecture.
- *Reuse of existing model-management technologies at runtime.* We leverage models at runtime without modification—that is, we keep the same model representation at runtime that we use at design time: the XML Metadata Interchange standard. This avoids the need for technological bridges, making it possible to apply the same technologies used at design time to manipulate XMI models at runtime.

We developed the Model-Based Reconfiguration Engine (MoRE) to implement model-management operations. These operations determine how the system should evolve and the mechanisms for modifying the system architecture accordingly. Thus, systems use the knowledge captured by variability models as if they were the policies that drive the system's autonomic evolution at runtime.

We applied our approach to the case of smart homes (www.autonomic-homes.com). This domain is suited for variability modeling techniques because of the high degree of similarities among different systems; also, AC capabilities can address some of the domain's limitations such as minimal support for evolution as new technologies emerge or as an application type matures.[5] Our research demonstrates the approach's feasibility for smart homes, especially for self-healing and -configuring capabilities.

AUTONOMIC COMPUTING FOR SMART HOMES

People continuously reconfigure domestic spaces and the technologies involved to support their activities.[5] To reduce this configuration effort, smart homes can provide the following autonomic capabilities:

- *Self-configuration.* New kinds of devices can be incorporated into the system. For example, when a new movement or occupancy detector is added to a home location, the different smart home services such as security or lighting control should automatically use it without requiring configuration actions from the user.
- *Self-healing.* When a device is removed or fails, the system should adapt itself to offer its services using alternative components to reduce the impact of loss of the device. For example, if an alarm fails, the smart home can make the home lights blink as a replacement for the failed alarm.
- *Self-adaptation.* Users' needs differ and change over time. The system should adjust its services to fulfill user preferences. For example, when a user leaves, the smart home should reorganize services to give priority to security.

This autonomic behavior is closely related to *context adaptation*—a system's ability to gather information about the domain with which it shares an interface, evaluate this information, and change its behavior according to the current situation. Individual autonomic capabilities also require the system to infer knowledge from the current situation and trigger an appropriate response. However, AC emphasizes freeing system users from the details of system operation and maintenance and providing 24/7 operation.[1]

To achieve such behavior, our approach uses variability models[4] and a dynamic product-line architecture.[6] Variability models specify a smart home's possible configurations, while a dynamic product-line architecture can be rapidly retargeted to a specific configuration.

Variability modeling

From the different modeling techniques suited for variability analysis, we chose *feature modeling* because it has good tool support for variability reasoning.[7] Feature modeling is widely used to specify system functionality in a coarse-grained fashion by means of features that constitute increments in system functionality.

Features are hierarchically linked in a tree-like structure through variability relationships such as optional, mandatory, single-choice, and multiple-choice. Figure 1 shows a

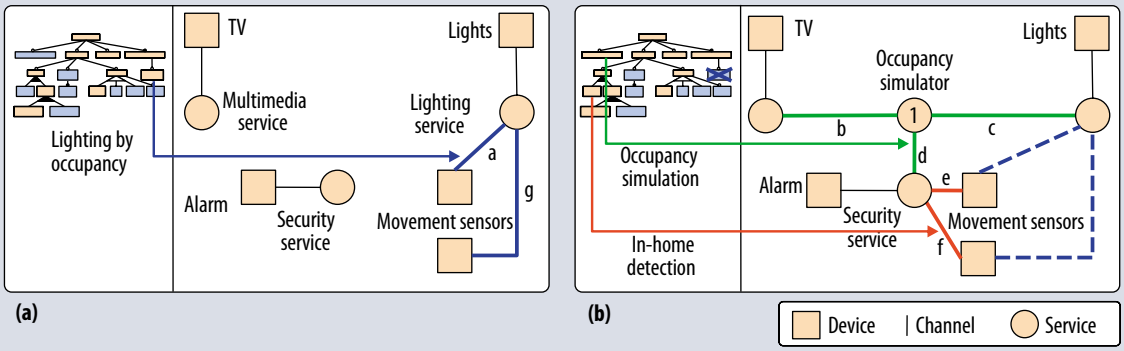

Figure 2. Impact of active features on system components for two scenarios: (a) the user is at home; (b) nobody is at home.

smart home feature model with automated illumination, multimedia, and security. The yellow boxes represent the smart home's current features, while the red boxes represent potential variants that may be activated in the future.

We let [FM] denote the set of all features, active and inactive, in a feature model. A system's current configuration (CC) is the set of all active features (F) in its feature model:

$$CC = \{F\} \mid F \in [FM] \wedge F.state = Active \wedge CC \subseteq FM$$

For example, for the feature model in Figure 1:

CC = {SmartHome,Security,InHomeSecuritySensing, Infrared160,Alarm,SilentAlarm,Multimedia,AutomatedIllumination,LightingByOccupancy}

Dynamic product-line architecture

We use a dynamic product-line architecture based on different components and their communication channels. We classify these components into two categories: *services* and *devices*. This architecture allows an easy reconfiguration since communication channels can be established dynamically between the components, and these components can dynamically appear or disappear from configurations.

Figure 2 shows this reconfigurable architecture according to the concrete syntax of PervML (www.pros.upv.es/labs/projects/pervml), a domain-specific language for developing smart homes. PervML provides a set of conceptual primitives for describing the system independently of the technology. PervML's structural model represents services by a circle, devices by a square, and the channels among services and devices by lines.

To specify which smart home components and channels support a certain feature, the model defines the *superimposition* operator (⊙). The superimposition takes a feature and returns the set of components and channels related to this feature, as the following examples show:

⊙(LightingByOccupancy) = {a,g}
⊙(OccupancySimulation) = {1,b,c,d}
⊙(InHomeDetection) = {e,f}

For example, the channels labeled a and g in Figure 2 support LightingByOccupancy.

RECONFIGURING SYSTEM ARCHITECTURE

In software product-line engineering, feature models focus on the efficient derivation of customized product variants that, once created, retain their properties throughout their lifetime. We argue that a system can activate or deactivate its own features dynamically at runtime by fulfilling certain *context conditions*. Examples of such conditions in smart homes are NewVolumetricSensor, AlarmFailure, and EmptyHome.

Because a given condition can trigger the activation/deactivation of several features, we define *resolution* (R) to represent the set of changes a condition triggers. A resolution is a list of pairs (F,S) in which F indicates a feature and S the feature's state. Each resolution is associated with a context condition and represents the change, in terms of feature activation/deactivation, produced in the system when the condition is fulfilled:

$$R = \{(F,S)\} \mid F \in [FM] \wedge S \in \{Active,Inactive\}$$

For instance, the condition EmptyHome is associated with the following resolution:

$$R_{EmptyHome} = \{(OccupancySimulation,Active),(InHomeDetection,Active),(LightingByOccupancy,Inactive)\}$$

This indicates that when the smart home senses that it is empty (according to the condition), it must reconfigure itself to deactivate LightingByOccupancy and to activate both OccupancySimulation and InHomeDetection.

The feature model at runtime enables the smart home to perform this reconfiguration. The smart home queries the models at runtime about the architectural components that support the features involved in the resolution. For example, taking the previous $R_{EmptyHome}$ as input, the smart home queries the feature model to determine the architecture for that specific context.

Architecture increments and decrements are calculated to determine the actions to modify the architecture. Specifically, we have defined two operations: ArchitectureIncrement ($A\Delta$) and ArchitectureDecrement ($A\nabla$). These operations take a resolution as input, and they calculate the modifications to the architecture in terms of components and channels. We define these operations by means of the superimposition operator and the relative complement operator (\), also known as the set-theoretic difference:

$$A\Delta = \odot((F,S) \in R \mid S = Active) \setminus \odot(CC)$$
$$A\nabla = \odot((F,S) \in R \mid S = Inactive)) \setminus \odot((F,S) \in R \mid S = Active)$$

For example, the results of these operations given $R_{EmptyHome}$ of the reconfiguration scenario shown in Figure 2 are as follows:

$$A\Delta_{EmptyHome} = \{1,b,c,d,e,f\}$$
$$A\nabla_{EmptyHome} = \{a,g\}$$

These operations indicate how to reorganize system components to move from a system configuration for when the user is at home (Figure 2a) to another when nobody is at home (Figure 2b). The movement sensors are no longer used for lighting (communication channels a and g are disabled, as indicated in $A\nabla_{EmptyHome}$); instead, they are used to provide information to the security service (communication channels e and f are enabled, as indicated in $A\Delta_{EmptyHome}$). In addition, the occupancy simulator (labeled as 1) is activated, and the communication channels required for this service to communicate with multimedia (channel b), lighting (channel c), and security (channel d) are established, as $A\Delta_{EmptyHome}$ indicates.

A resolution represents a partial configuration in which only a portion of the desired feature model state is defined. To check that the defined resolution set is actually within the variability model's constraints, developers can apply existing feature model analysis tools. For example, the Feature Model Analyzer (FAMA)[7] validates a partial configuration's consistency with a given feature model. Further, FAMA can combine multiple resolutions to ensure that there are no invalid configurations in a given situation.

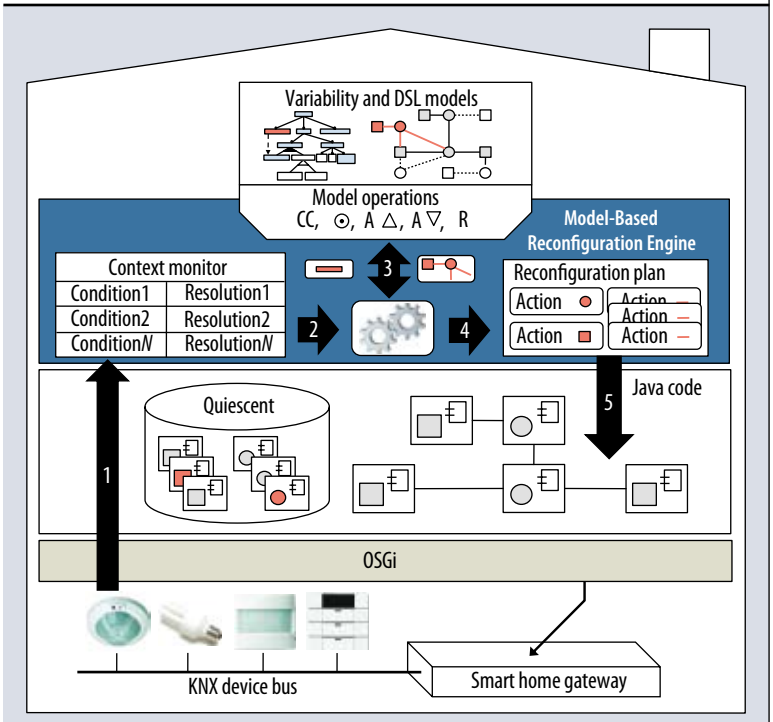

Figure 3. Model-based reconfiguration process. MoRE translates contextual changes into changes in the activation/deactivation of features.

MODEL-BASED RECONFIGURATION ENGINE

To achieve AC, a system must evolve from one configuration to another. Since our approach performs reconfiguration in terms of features, we developed MoRE to translate contextual changes into changes in the activation/deactivation of features.

Figure 3 shows the reconfiguration process. The *context monitor* uses the runtime state as input to check *context conditions* (step 1). If any of these conditions are fulfilled (for example, the home becomes empty), MoRE uses the associated resolution and previous model operations to query the runtime models about necessary architectural modifications (step 2). The engine uses the models' responses to generate a *reconfiguration plan* (step 3). This plan contains a set of *reconfiguration actions* that modify the system architecture and maintain consistency between the models and architecture (step 4). Execution of the plan modifies the architecture to activate/deactivate the features specified in the resolution (step 5).

MoRE uses the OSGi framework[8] to implement the reconfiguration actions. This framework contains a complete components model that extends Java's dynamic capabilities. We classify reconfiguration actions into three main categories:

- *Component actions.* Jeff Kramer and Jeff Magee[9] describe how a component must transit from an active

(operational) state to a quiescent (idle) state to perform system adaptation. We have implemented this pattern by means of the OSGi capabilities to install, start, restart, and uninstall components without restarting the entire system. All those components that are irrelevant for the current configuration are in a catalog of quiescent components that do not consume processor or memory resources but are ready to be started.
- *Channel actions.* Once a component transits to an active state, it must establish communication with other services. These communication channels, also called bindings, are implemented using the OSGi Wire class. An OSGI Wire is an enhanced implementation of the publish-subscribe pattern oriented to dynamic systems. In particular, an OSGi Wire implements the whiteboard pattern.

A feature model hides much of the complexity in the definition of an autonomic system's adaptation space.

- *Model actions.* After the system architecture has been modified, MoRE updates the feature model according to the new system functionality. It performs this update by means of a partial reflection of the architecture using *model introspection*. This powerful feature of existing modeling frameworks like the Eclipse Modeling Framework (www.eclipse.org/modeling) allows a program to work with any model by querying its structure dynamically at runtime. Model actions apply this technique to update the feature model's current configuration.

In example shown in Figure 2, when the user leaves home, MoRE is in charge of composing the suitable actions to reorganize the system architecture to give priority to security. To achieve $A\Delta$, the engine applies a component action to

- find the occupancy simulator's components in the catalog of quiescent components, and
- start these components.

MoRE thus moves the components from the catalog to the current architectural configuration. The occupancy simulator generates inputs for the multimedia and lighting services with the aim of deterring thieves by acting as if there were people at home, and channel actions are required to connect these services. Additional channel actions are needed to connect the movement sensors with the security service. To achieve $A\nabla$, MoRE breaks the channels between the movement sensors and lighting to deactivate the LightingByOccupancy feature.

Once the architecture has been successfully modified, MoRE must update the feature model accordingly. It sets the LightingByOccupancy feature to inactive and both OccupancySimulation and InHomeDetection to active to reflect the system's current state. Consequently, both the feature model and the system architecture are synchronized and support the desired behavior when nobody is at home.

EVALUATION

We evaluated our approach with respect to both autonomic-level achievement and performance.

Autonomic level achievement

To determine the level of autonomic behavior that can be achieved with our proposed approach, we used a state machine, which engineers employ to represent and check adaptation policies.[3]

Figure 4 illustrates how a simple feature model consisting of only four features (Figure 4a) defines eight possible system configurations, C1 to C8 (Figure 4c). When defining a condition for activation of a system feature by means of a resolution—$R_{ConditionX}$, $R_{ConditionY}$, or $R_{ConditionZ}$ (Figure 4b)—a designer is expressing the transitions between different system states (Figure 4d) in a declarative manner, without the need for an exhaustive definition of each state transition or the transitions derived from the composition of states. In this case, a single resolution such as $R_{ConditionY}$ results in eight transitions among system variants (represented as dashed lines).

A feature model hides much of the complexity in the definition of an autonomic system's adaptation space. In Figure 1, the feature model containing 18 features represents more than 200,000 states, and the three resolutions specified for this model—NewVolumetricSensor, AlarmFailure, and EmptyHome—define more than 600,000 possible transitions among system variants. Feature models provide an intensional rather than extensional description of each of the system's possible states.

We used real-life deployment examples for smart homes to evaluate the autonomic level our approach achieved. We collected these examples from previous studies of how users continuously reconfigure their homes and technologies within them to meet their demands.[5] Specifically, we cataloged adaptation scenarios to test the autonomic capabilities our approach introduces.[10]

Overall, we successfully reproduced 78 percent of the scenarios. Our approach covered 93 percent of self-configuring scenarios and 81 percent of self-healing scenarios, while providing more discrete results—65 percent coverage—for self-adapting scenarios. Many self-configuring and -healing scenarios involve simple conditions (a new device is plugged in or a device fails) for triggering activation of a single feature that controls the new device or

provides alternative functionality in case of failure. More detail is required to define autonomic behavior in self-adapting scenarios, which deal more directly with user preferences.

Because our approach defines autonomic behavior at the feature level, some specific requirements for system adaptation fall out of the variability scope. However, even though this lack of coverage could be complemented by developing specific components for the unsupported cases, it does not seem economically realistic to build individual features to suit each user. We intend to focus on commonalities and abstractions that are valid across a set of users, looking to trade off personalization and reusability. For those needs that fall out of the variability model's scope, designers must decide whether to provide a specific solution for a single system or update the variability model accordingly.

Scalability of model-handling technologies at runtime

Model manipulation at runtime, as opposed to design time, is subject to the same efficiency requirements as the rest of the system because the execution of model operations impacts overall system performance. In the adaptation scenario evaluations, model processing did not introduce significant performance penalization. However, to validate whether our proposed approach scales to large systems, we quantified this overhead for randomly generated large models.

These models started with one element and were populated with 200 new elements in each iteration. After model population, we applied and evaluated the different operations supporting the calculation of architecture modifications $A\Delta\nabla$. Even with 45,000 elements in each model, the response time was less than 500 milliseconds, which is acceptable compared to the typical performance of devices and communication networks in the smart home domain. Thus, we conclude that our reuse-based approach can be applied in other domains with similar temporal constraints.

RELATED WORK

David Garlan and Bradley Schmerl[11] pioneered the use of architectural-based models during runtime as the basis for system monitoring, problem detection, and repair. To achieve self-healing, they extended architectural models with property annotations on components to describe their quality of service (QoS). In contrast, we base autonomic behavior on variability models without modifications. Further, our resolutions scale to large systems because our approach expresses autonomic behavior in a declarative manner. Apart from self-healing scenarios, our approach also addresses self-adapting and -configuring scenarios.

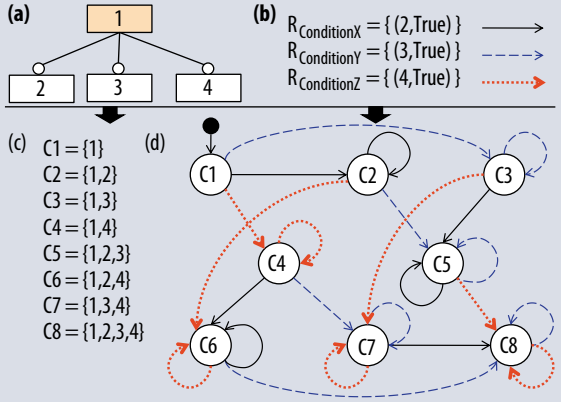

Figure 4. Visualizing variability as an adaptation space. A simple feature model consisting of only (a) four features defines (c) eight possible system configurations. When defining a condition for activation of a system feature by means of (b) a resolution, a designer is expressing (d) the transitions between different system states in a declarative manner, without the need for an exhaustive definition of each state transition or the transitions derived from the composition of states.

Brice Morin and colleagues[3] propose a combination of model-driven engineering and aspect-oriented modeling to support dynamic runtime variability. They dynamically compose aspects to produce a range of configuration models and then use these models to generate the scripts needed to adapt a running system from one runtime configuration to another. These adaptation scripts feature reconfiguration commands to modify the system architecture as our approach does, enabling the development of adaptive systems without having to enumerate all their possible configurations. Because costs represent a major constraint in mass-production environments, we achieve autonomic behavior by reusing at runtime the same variability models that were used at design time.

Jaejoon Lee and Kyo C. Kang[12] propose an approach for engineering dynamically reconfigurable products. In particular, they extend feature models with the notion of binding units to group system features. They also provide guidelines for building dynamically reconfigurable architectures based on QoS properties and for how a configurator should work. Nevertheless, they remain at the general level with respect to the models employed and do not provide techniques for superimposing features and system components as we do. Further, because their architectural components are grouped on indivisible binding units, the number of system variants to support reconfiguration is smaller than those of our feature-based approach. Since we use feature models without modifications, we can benefit from current work on feature model reasoning[7] instead of having to develop ad hoc reasoners.

With more devices being added to our surroundings, users increasingly seek simplicity. AC plays a key role in simplifying computing systems by reducing the need for maintenance. Historically, autonomic system developers focused on advanced capabilities. However, for mass-production environments, a tradeoff is necessary between customization and system development. The use of design models at runtime offers new opportunities for autonomic capabilities without increasing development costs. This is accomplished by means of a planned reutilization of the efforts invested at design time.

Whether for smart homes, mobile devices, or automotive systems, users require more autonomic functionality. We believe the techniques we have applied to the smart home domain can achieve similar results in other mass-production environments. In addition, the role of models at design time can be extensively exploited for validation and verification.

Because feature models, which determine autonomic behavior, are available at design time, we can thoroughly analyze specifications for the purpose of validation. We can guarantee deterministic reconfigurations at runtime, which is essential for reliable systems. Further, existing variability analysis techniques can detect unintended evolutional behavior in autonomic systems as a next step in providing systems that fulfill many user needs out of the box. C

Acknowledgments

Special thanks to Øystein Haugen for contributing valuable thoughts on variability and model-driven development issues. We also thank Nelly Bencomo and the anonymous reviewers for their helpful comments on late versions of this article. This work has been developed with the support of MEC under the project SESAMO TIN2007-62894 and cofinanced by FEDER.

References

1. J.O. Kephart and D.M. Chess, "The Vision of Autonomic Computing," *Computer*, Jan. 2003, pp. 41-50.
2. J. Zhang and B.H.C. Cheng, "Model-Based Development of Dynamically Adaptive Software," *Proc. 28th Int'l Conf. Software Eng.* (ICSE 06), ACM Press, 2006, pp. 371-380.
3. B. Morin et al., "An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability," *Proc. 11th Int'l Conf. Model Driven Eng. Languages and Systems* (MoDELS 08), LNCS 5301, Springer-Verlag, 2008, pp. 782-796.
4. J. Coplien, D. Hoffman, and D. Weiss, "Commonality and Variability in Software Engineering," *IEEE Software*, Nov. 1998, pp. 37-45.
5. J. O'Brien et al., "At Home with the Technology: An Ethnographic Study of a Set-Top-Box Trial," *ACM Trans. Computer-Human Interaction*, Sept. 1999, pp. 282-308.
6. S. Hallsteinsen et al., "Dynamic Software Product Lines," *Computer*, Apr. 2008, pp. 93-95.
7. D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated Reasoning on Feature Models," *Proc. 17th Int'l Conf. Advanced Information Systems Eng.* (CAiSE 05), LNCS 3520, Springer-Verlag, 2005, pp. 491-503.
8. D. Marples and P. Kriens, "The Open Services Gateway Initiative: An Introductory Overview," *IEEE Comm. Magazine*, Dec. 2001, pp. 110-114.
9. J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Trans. Software Eng.*, Nov. 1990, pp. 1293-1306.
10. C. Cetina, J. Fons, and V. Pelechano, "Applying Software Product Lines to Build Autonomic Pervasive Systems," *Proc. 12th Int'l Software Product Line Conf.* (SPLC 08), IEEE CS Press, 2008, pp. 117-126.
11. D. Garlan and B. Schmerl, "Model-Based Adaptation for Self-Healing Systems," *Proc. 1st Workshop Self-Healing Systems* (WOSS 02), ACM Press, 2002, pp. 27-32.
12. J. Lee and K.C. Kang, "A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering," *Proc. 10th Int'l Software Product Line Conf.* (SPLC 06), IEEE CS Press, 2006, pp. 131-140.

Carlos Cetina *is a researcher and PhD student in the Centro de Investigación en Métodos de Producción Software (Research Center on Software Production Methods), Universidad Politécnica de Valencia (Polytechnic University of Valencia), Spain. His research interests include model-driven development, software product lines, autonomic computing, and pervasive systems. Cetina received an MS in computer science from the Polytechnic University of Valencia. Contact him at ccetina@pros.upv.es.*

Pau Giner *is a researcher and PhD student in the Research Center on Software Production Methods, Polytechnic University of Valencia. His research interests include business process modeling, the Internet of Things, and mobile and ubiquitous computing. Giner received an MS in computer science from the Polytechnic University of Valencia. Contact him at pginer@pros.upv.es.*

Joan Fons *is a collaborator professor in the Research Center on Software Production Methods, Polytechnic University of Valencia. His research interests include model-driven development, Web engineering, pervasive computing, and user interface development. Fons received a PhD in computer science from the Polytechnic University of Valencia. Contact him at jfons@pros.upv.es.*

Vicente Pelechano *is an associate professor in the Research Center on Software Production Methods, Polytechnic University of Valencia. His research interests include model-driven development, Web engineering, mobile and ubiquitous computing, and business process modeling. Pelechano received a PhD in computer science from the Polytechnic University of Valencia. He is currently leading the technical supervision of the MOSKitt open source CASE tool (www.moskitt.org). Contact him at pele@pros.upv.es.*