Towards Model-Centric Engineering of a Dynamic Access Control Product Line

Mahdi Derakhshanmanesh
University of Koblenz-Landau
Koblenz, Germany
manesh@uni-koblenz.de

Mazeiar Salehie
Lero- The Irish Software Eng. Research Centre
Limerick, Ireland
mazeiar.salehie@lero.ie

Jürgen Ebert
University of Koblenz-Landau
Koblenz, Germany
ebert@uni-koblenz.de

ABSTRACT

Access control systems are deployed in organizations to protect critical cyber-physical assets. These systems need to be adjustable to cope with different contextual factors like changes in resources or requirements. Still, adaptation is often performed manually. In addition, different product variants of access control systems need to developed together systematically. These characteristics demand a product line engineering approach for enhanced reuse. Moreover, to cope with uncertainty at runtime, adaptivity, i.e., switching between variations in a cyber-physical domain (reconfiguration) and adjusting access policies (behavior adaptation), needs to be supported.

In this position paper, we sketch an approach for engineering dynamic access control systems based on core concepts from dynamic software product lines and executable runtime models. The proposed solution is presented and first experiences are discussed along a sample dynamic software product line in the role-based access control domain.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Management, Security

Keywords

Access control systems, adaptive software, dynamic software product lines, runtime models

1. INTRODUCTION

Access Control Systems (ACSs) can support security in physical, cyber or cyber-physical domains. No matter what approach is followed for realizing access control (role-based, attribute-based, etc.) in practice, especially in physical access control systems, managing changes is normally performed manually. Changes to access control policies are required though, e.g., to consider new employees, visitors with special needs, or varying business processes. Hence, Dynamic Access Control Systems (DACSs) are needed that can adjust themselves quickly to suit emerging situations.

Generally, we noticed that *adaptive security* and *self-protection* have not been widely addressed in the adaptive software community [8] and the access control domain is not an exception. Particularly, we are not aware of a solution that specifically combines Software Product Line (SPL) technology [7] and runtime adaptation in this domain. Since access control systems can be available in different variants depending on the needed security features, following the SPL approach seems natural, though. Furthermore, once a DACS was deployed, it needs to stay continuously and actively in place. Therefore, it is desirable to develop security systems that adapt themselves during operation.

In this position paper, we propose to follow a Dynamic Software Product Line (DSPL) [3] approach for realizing adaptive security. We believe DAC is an appealing sample domain for adaptation and DSPLs, because variants need to be switched (reconfiguration) and access policies are required to be adjusted (behaviour adaptation) at runtime. We scope the problem domain to *physical access control* in this paper and we consider the well-known Role-Based Access Control (RBAC) model [2]. Our main contributions are (i) to discuss the potential *application of DSPLs* in the dynamic access control domain and (ii) to sketch a flexible *model-centric solution* following our previous work [1] on engineering adaptive software that leverages query/transform/interpret operations on graph-based runtime models.

2. MOTIVATING EXAMPLE

Imagine a software company builds and maintains a product line of cyber-physical access control systems. Variability in the domain is captured using *feature models* [4] and a product variant is described by a *configuration*, i.e., by a selected set of features. Hence, a product line is represented by a feature model covering physical (e.g., access to buildings), cyber (e.g., access to file systems), and cyber-physical access control systems (e.g., access to physical and information assets in a company) with varying sensors and security control mechanisms.

This sample product line consists of two mutual exclusive access control models: Access Control List (ACL) and RBAC. Depending on the resources to be protected, con-

Published in the 6th International Workshop on Dynamic Software Product Lines (DSPL 2012) collocated with SPLC'12, Salvador, Brazil.

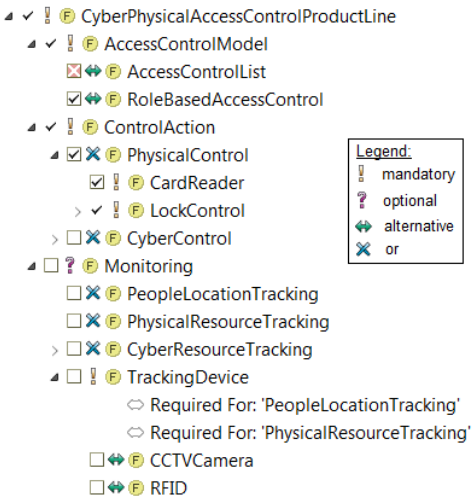

Figure 1: Sample Product Feature Configuration

trol actions and monitoring features can be selected. For a physical access control product variant, card reader and lock control as well as the tracking of people, physical or cyber resources can be chosen.

Subsequently, we assume a product is configured as shown in Figure 1. The illustrated excerpt does not include variation points and variations in implementing assets. The derived product variant is deployed in an environment with roles and rooms as described in Listing 1.

```
Roles = {Manager, Engineer, Admin Staff, IT staff,
         Guest, VIP}
Rooms = {Manager Room (R1), Main Office (R2),
         IT Room (R3), Engineering Rooms (R4,R5,R6),
         Seminar Room (R7)}
```

Listing 1: Physical Setting of Deployed Product

Moreover, the initial set of access control policies is formulated as given in Listing 2 to secure the building under the typical conditions assumed at the time of deployment.

```
IF role = "Manager" THEN grantAccess(R1,R7)
IF role = "Engineer" THEN grantAccess(R4,R5,R6,R7)
IF role = "Admin Staff" THEN grantAccess(R2,R1)
IF role = "IT Staff" THEN grantAccess(R3,R7)
IF role = "Guest" THEN grantAccess(R7)
```

Listing 2: Initial Set of Access Control Policies

Given this setup, imagine a VIP plans to visit the organization to participate in a seminar and to have a meeting with the manager. To protect the VIP, the *adaption strategy* is to revoke access permissions from specific roles such that only managers can be with the VIP in the same room. Additionally, existing business processes shall not be compromised, i.e., pre-booking and occupying rooms for the VIP is not an option, even though the person is not in that room. It is desired to adjust access control policies dynamically.

Adaptation strategies shall be implemented for the whole product line and shall be shipped with each product variant so it can react during operation. In the rest of this paper, we describe how a DSPL-based and model-centric approach can support the engineering of such a dynamic access control product line.

3. ARCHITECTURE

The dynamic access control system follows an architectural blueprint that we originally designed for adaptive software systems and which we implemented prototypically as the *Graph-based Runtime Adaptation Framework* (GRAF) [1]. To achieve adaptivity, this Java framework uses executable models at runtime and related techniques (especially: querying, transforming, and interpreting) as well as basic mechanisms from aspect-oriented-programming.

Subsequently, the core aspects of this solution concept, and its application in the context of DSPLs, are introduced via a *structural* and *behavioral view*.

3.1 Structural View

As sketched in Figure 2, we follow a layered architecture. Included components, repositories, and their roles are briefly introduced here as well as three essential interfaces. Note, that in a full-scale implementation, the presented architecture would be developed and used by two processes, i.e., by *domain engineering* and *application engineering* [7].

Access Control Core Functionality Layer. The lowest layer contains the product line's artifacts with *core functionality*, e.g., for granting and rejecting access to rooms as well as for reading data from RFID chips as needed for location monitoring. In general, performance critical functionality as well as driver software are located here.

Artifacts in this layer can expose sensed context changes (`StateVar`), e.g., data from tracking asset positions. Moreover, at predefined variation points in the artifacts, the control flow can be redirected to upper layers (`InterpretationPoint`) and a variation of functionality may be executed (`Action`) as described by models. The roles of these interfaces are described in Section 3.2.

Middleware Layer. Two model interpreters play a central role in this architecture. Both use the `Action` interface to call existing functionality or to change the state of artifacts and contained elements: The *access control model interpreter* executes the access control model by calling implemented behavior variations. The *configuration model interpreter* is invoked on changes to the configuration model and it enables/disables sensors in the assets or loads components that were not deployed but are needed to implement a new configuration of features.

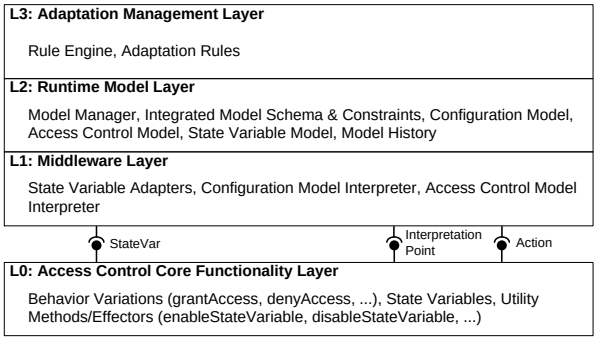

Figure 2: Layered Model-Centric Architecture for Dynamic Access Control Software (Following GRAF [1])

Runtime Model Layer. The product's runtime model [5] is interpreted during operation and conforms to the *runtime model schema* (metamodel) that is illustrated in Figure 3. It consists of three parts: (i) an *access control model* that encodes access control policies such as those illustrated in Listing 2, (ii) the *configuration model* that captures variability in features and implementing program elements (variables, methods) and (iii) a *state variables model* that represent the sensed subset of a product's context.

The runtime model schema for a specific product line is designed and developed during domain engineering. Then, an initial instance of the runtime model is created during application engineering and before product deployment. Maintaining the full schema and one concrete model instance, as an integrated part of each product at runtime, is at the heart of our approach to DSPLs.

Adaptation Management Layer. Given this setup, the product's behavior can be changed dynamically by adjusting the access control model during operation. This adaptation can be performed either (i) *autonomously* based on predefined *adaptation rules* and a generic *rule engine* as presented here, and (ii) *manually*, e.g., via a user interface or by calling the provided model-manipulation API. Designing and implementing adaptation rules are part of the product-line's domain engineering process. The full implementation of this architecture is ongoing work.

3.2 Behavioral View

The *runtime behavior* of a DACS, developed according to our approach, is strongly determined by the state and knowledge that is represented in the runtime model. When conditions in the environment change that can be sensed, the values of state variables embedded in the core functionality layer are propagated to the runtime model via the middleware layer's state variable adapter components. In effect, the runtime model is changed, i.e., it now contains the fresh sensor data, and the rule engine gets notified via the model manager. Additionally, model changes can be triggered by human administrators (not presented here) who manually feed new environment data into the state variables, add new roles, rooms or change the selection of features.

In reaction to a model change event, the rule engine component in the adaptation management layer plans appropriate steps using the set of available *adaptation rules*. Adaptation rules are Event Condition Action (ECA) rules where (i) the event event is usually a *model change event*, (ii) the condition is realized as a *model query* and (iii) the action

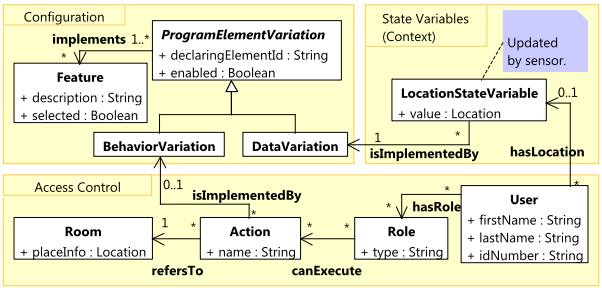

Figure 3: Excerpt of Runtime Model Schema (Visualized in Concrete Syntax of UML Class Diagrams)

is represented by a *model transformation* that is sent to the model manager for execution. A transformation of the runtime model results in an adapted behavior or configuration state of the DACS as described subsequently.

Adaptation of Access Control Policies. The responsible model interpreter is invoked at an interpretation point in the core functionality layer, e.g., when a user requests access and the respective code is executed. Then, the interpreter walks along a path from the user's role to the room's representation in the model and executes the behavior associated with the `Action` node. In this example, the associated `BehaviorVariation` can point to one of two methods: `grantAccess()` or `denyAccess()` by their `declaringElementId` (e.g., a fully qualified method name).

Adaptation of Configuration. In case that the configuration model changed, the responsible model interpreter can activate and deactivate variations, e.g., sensor variables. The activation state of all available program elements (that represent a variation) is synchronized between the runtime model and the implementing artifacts. To support adaptation, the configuration model captures the generally available variations in product line artifacts, even if they were not deployed in the given product variant.

In the subsequent section, we describe the application of this architecture along the introduced motivating example.

4. APPLICATION SCENARIO

In this section, we describe how the DACS reacts in one specific *application scenario* as partially introduced in Section 2. The hypothetical customer buys a subset of the product line that initially covers the features selected in Figure 1. In addition, adaptation rules are shipped with the product such that its capabilities can be adjusted automatically to (foreseen) context changes. The configuration model represents the initial feature selection and the access control model encodes the initial access control policies.

At some point, a VIP visits the company, which is an exceptional event from the customer's perspective. The new requirement is now to ensure that only managers can be in the same room as the VIP. To satisfy this goal, a *location-based* approach shall be applied where the VIP's location is continuously tracked. This feature was foreseen in the product line, i.e., in terms of the `Monitoring` features, but none of them was selected during initial shipping to respect privacy concerns of the organization's regular staff.

We assume that the VIP is given a card or gadget to be identified by the access control system. After detecting the VIP user in the system boundary, the user will be added, and the selection state of the `PeopleLocationTracking` feature is set to true in the configuration model part of the runtime model. All these modifications are done during operation of the DACS and at the model-level.

In result of the changed configuration, a state variable needs to be activated to provide the VIP's location information, e.g., via RFID, assuming the VIP is carrying an RFID-enabled card. In this sample scenario, each `DataVariation` in the runtime model has an `enabled` flag. Since the `LocationStateVariable` is required for implementing the `PersonTrackingFeature`, it must be activated. The configuration model interpreter enables the sensor via the `Action` interface once the feature selection changed. Whenever the VIP enters a new room, his location information is updated in the runtime model.

Afterwards, the rule engine is invoked and can react, e.g., by choosing the adaptation rule sketched in Listing 3, to update the access control policies represented by the access control model.

```
1 ON (VIPRoleIntroducedEvent)
2 IF (isSelected(PersonTrackingFeature) AND
3     exists(User.role == "VIP") AND
4     User.location != ModelHistory.getPrev(User.location))
5 DO {
6     // Revoke access from all users to room, except for managers
7     // Action->behaviorVariation.declaringElementID="denyAccess"
8 )
```

Listing 3: Adaptation Rule for Updating the Access Control Model (Pseudocode)

The next time a person attempts to enter any room, e.g., using an access chip-card, the control flow in the access control assets reaches the interpretation point. The model interpreter searches for a path between the `User` and the corresponding `Room` nodes in the runtime model and executes the `Action` node's associated `BehaviorVariation` to grant or deny access, e.g., by calling methods in the core functionality layer using Java's reflection mechanism.

5. RELATED WORK AND CONCLUSIONS

In this position paper, we presented *one* possible way to achieve reuse and to deal with dynamically changing customer requirements in the domain of access control systems by following a customized dynamic software product line approach. The solution concept is based on a flexible architecture that uses a central, executable runtime model. The presented approach can be implemented using GRAF [1].

Up to our knowledge, SPL and DSPL approaches have not been employed in building dynamic access control systems. Bertino et al. [9] developed a framework for *spatio-temporal RBAC* to adjust access policies based on the users' identity and environmental parameters (time, location). Zhang and Parashar [10] also presented a *context-aware access control* mechanism that dynamically adjusts access permissions. In a recent work, Morin et al. [6] proposed a model-based approach to adapt access control policies without considering possible reconfiguration in system features. In contrast to these related approaches, we include not only a configuration model at runtime, but also shift a part of the application's domain logic to the runtime model layer.

We believe that this architectural decision is beneficial, because (i) establishing the schema for the runtime model during domain engineering *supports understanding the domain*. Additionally, (ii) *separation of concerns* between the variable domain logic (access control model) and the reusable behavior variations (grant/deny access methods) in source code artifacts was achieved. Furthermore, (iii) there is *no redundancy between the runtime model and code*, as the access control policies are defined as a part of the flexible runtime model. Most of all, (iv) the model representation supports to *inspect and adapt* a part of the product state (feature configuration, access control policies, state variables) during operation by executing common querying/transform operations on the runtime model.

Summing up, our solution approach illustrates how to construct a dynamic software product line of adaptable access control systems based on (i) a loosely coupled set of artifacts including variations for implementing essential domain-specific operations and (ii) a model-representation of the feature configuration, domain logic, and state variables which can be permanently evolved at runtime. Even though our work is in a preliminary state, applying our previously published mechanisms for runtime adaptivity to the access control domain motivates us to continue our efforts.

In terms of future work, the presented approach needs to be implemented in a generic way and we plan to analyze if it is feasible to support product management by using information from a runtime model history. Moreover, the modeling of variation points and variations needs to be refined carefully as the related modeling part is based on a pragmatic solution. Adding and removing new hardware variations at runtime (as a part of reconfiguration) to implement existing features needs to be covered as well. Finally, we feel that integrating the *process-related knowledge* from traditional SPL engineering should be more in the focus of research towards engineering truly dynamic software product lines.

6. ACKNOWLEDGMENTS

The authors kindly thank Mehdi Amoui (Waterloo) for his feedback. This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero.

7. REFERENCES

[1] M. Derakhshanmanesh, M. Amoui, G. O'Grady, J. Ebert, and L. Tahvildari. GRAF: Graph-based Runtime Adaptation Framework. In *Proc. of the Int. Symp. on Software Eng. for Adaptive and Self-Managing Systems*, pages 128–137, 2011.

[2] D. F. Ferraiolo and D. R. Kuhn. Role-Based Access Controls. In *Proc. of National Computer Security Conf.*, pages 554 – 563, 1992.

[3] S. Hallsteinsen, M. Hinchey, S. P. S. Park, and K. Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, 2008.

[4] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. *Distribution*, 17(November):161, 1990.

[5] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@Run.time to Support Dynamic Adaptation. *Computer*, 42(10):44–51, Oct. 2009.

[6] B. Morin, T. Mouelhi, F. Fleurey, and Y. L. Traon. Security-Driven Model-Based Dynamic Adaptation. *Proc. of the the IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 205–214, 2010.

[7] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[8] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. on Autonomous and Autonomic Systems*, 4(2):1–42, May 2009.

[9] A. Samuel, A. Ghafoor, and E. Bertino. A Framework for Specification and Verification of Generalized Spatio-Temporal Role Based Access Control Model. Technical report, Purdue University, February 2007.

[10] G. Zhang and M. Parashar. Context-aware Dynamic Access Control for Pervasive Applications. In *Proc. of the Comm. Networks and Distributed Sys. Modeling and Simulation Conf.*, pages 21–30. Citeseer, 2004.