Addressing Data Model Variability and Data Integration within Software Product Lines

Joerg Bartholdt
Siemens AG
Corporate Technology
Architecture, CT SE 2
Otto-Hahn-Ring 6
81739 Munich, Germany
joerg.bartholdt@siemens.com

Roy Oberhauser
Aalen University
Computer Science Dept.
Beethovenstr. 1,
73430 Aalen, Germany
roy.oberhauser@htw-aalen.de

Andreas Rytina
itemis
Agnes-Pockels-Bogen 1
80992 Munich, Germany
andreas.rytina@itemis.de

Abstract

Software Product Line (SPL) engineering is one approach for addressing customization and variability for software products. However, current state-of-the-art often focuses on feature modeling and component variability while insufficiently addressing data model variability difficulties and their associated complexity. Various software qualities, such as correctness, reusability, maintainability, testability, and evolvability, are negatively impacted.

In this article the Approach for Data Model Variability (ADMV) is described which provides a unified and systematic methodology for providing a consistent view to capture data variability in data models. Adapter generation hides and decouples components from superfluous data elements and supports SPL data integration with the potentially multifarious external systems and devices that a SPL may need to consider. An eHealth SPL case study is presented supporting adapter generation with differential data conversion and data integration with medical devices. The results show that with this approach, data model variability and data integration can be effectively addressed and desirable software qualities preserved.

Keywords - Data Modeling; Data Integration; Variability; Software Product Lines; Unified Modeling Language; Model-Driven Software Development

1. Introduction

One approach that promotes the systematic reuse of software components for different but similar software products (typically products in the same domain) is SPL Engineering (SPLE). Typically the commonalities and variability of the products in the product line are captured and then the development is split into domain (commonalities) and application (additional individual features for the final product). Products are then built by integrating the common artifacts (usually a platform) and optionally configuring them with product-specific artifacts [11] [14].

Significant work and various methodologies for domain analysis and variability modeling for SPLs with a focus on features are, for instance, Feature-Oriented Domain Analysis (FODA) methodology [9], FeatuRSEB [8], PuLSE [2] and "the notion of variability" [25]. Typical feature models in SPLs allow for many ($\sim 10^x$) possible permutations. Considering that an artifact may influence the data model (e.g., adds new data or relations), all artifacts must be able to handle multiple data variants, although they themselves make no use of the available differences. Yet the aforementioned methodologies do not sufficiently support and address variability in the data models. The Orthogonal Variability Model (OVM) [14] does go beyond features to addressing variability in artifacts, but is an abstract approach missing a notation that can be used by automation for data models (also known as schemata). While the challenging issue of data model variability has been previously studied under schema integration [13], data conversion, data and metadata heterogeneity, schema evolution, enterprise application integration, etc., a holistic approach for SPLE is absent.

The Approach for Data Model Variability (ADMV) described in this paper provides a unified methodology for SPLE to consistently view and edit the data within the data model, capture the variability, as well as shield artifact developers from extraneous differences. Additionally, constraint checking support for data integration variability in SPLs via views and adapter

generation is considered, expanding on our previous work [1].

To motivate and demonstrate the features of the ADMV, a case study in the medical domain for an eHealth SPL derived from a third party served as the research basis. It is presented in simplified form for this article. In the following section the scenario and solution requirements are presented. In Section 3, the ADMV is presented and then applied in Section 4 to an eHealth SPL scenario to exemplify the approach. Section 5 considers alternative approaches and Section 6 evaluates the solution against qualities. Related work is then discussed in Section 7. A conclusion and future work discussion follows in Section 8.

2. Scenario and requirements

In eHealth, an increasing market demand for integrated medical information systems and solutions exists, with globalization in the market and customization demands spanning national boundaries. The difficulties for developing and supporting such systems become apparent in time-to-market, labor costs, and error-proneness when aligning and supporting the various data models and data integration needed for such systems. To support a variety of markets, an SPL approach allows the medical information platform customer to select arbitrary features as add-ons to the base product, e.g., date-definition, record repository, security, etc. This entails various challenges, among them that the overall product instance-specific data model will change depending on the features selected, and another challenge being the integration requirements with medical devices and other medical systems.

For example, a medical information system shall work in different hospital environments. Patient data are stored in folders representing a single hospital stay ("clinical record"). All documents created during a later hospital stay are stored in a different folder. In another environment (e.g., triggered by the electronic case record (eCR) specification in Germany [19]) a new folder level "case record" is introduced on the top level. Beneath, the structure follows the previously described "clinical record". All clinical records are sorted by a disease code into the different "case record" folders. That way, explicit access can be granted to medical personnel based on the medical issue (an orthopedic physician treating a broken leg would have no or restricted access to the psychological problems of the same patient). The product line shall be applicable in both types of domains.

Another requirement is the integration of various measurement devices for blood pressure, body temperature, etc., see Figure 2.1. The devices deliver semantically comparable values, but in different data formats, different scales (e.g., °C/°F) and different protocols. Nonetheless, the application must be able to manage that data in a consistent way, abstracting from the differences in detail.

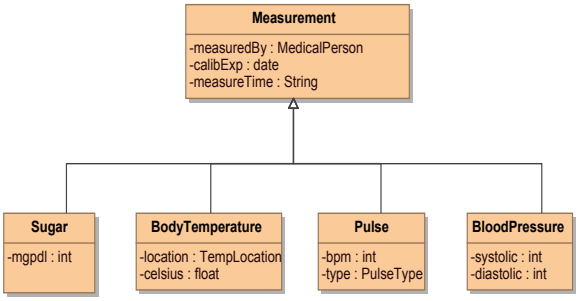

Figure 2.1. UML class diagram of the Measurement data model

A feature of the medical application includes, for instance, the calibration expiration management of the measurement devices. This requires each measurement to carry the information if the measurement was made beyond the calibration expiration and ideally, the expiration date itself (to leave the interpretation to the physician).

Optionally - depending on the environment (e.g., ambulatory vs. stationary), a history of data changes (measured values, patient demographics, etc.) must be recorded, which can be seen as a cross-cutting concern/requirement on domain objects.

Implementing these features and their variability has many effects on the data model of the product instances. E.g., modules for presentation of patient measurements should be programmed with a stable view on the relevant data, ignoring various formats of data delivery (date in `long` or `String` format), data interpretation (°C/°F), and additions like history. This reduces the dependency of such modules on the data model and other components that can vary in the product line instances, thus relieving developers from dealing with this (from their perspective impertinent) variability.

2.1. Requirements

The deficiencies in the examples above illustrate the following requirements that are imposed on the solution to cope with variability in data models:

1) Modeling of the data objects in the solution space must be consistent and provided in a central view (analogous to the feature tree in the problem space that

shows a central view of the variants of the product line). This allows developers and engineers to keep the overview and consistency of the possible product instances and the corresponding data models. The individual products must be derived from this model.

2) Developers of artifacts shall be shielded from the effects of the many possible variants on their code (API and structure of the domain objects) while retaining the compile-time safety that getter/setter navigation in the domain object model guarantees. This includes the demand for loose coupling not only for the functionality of components, but also for the data exchanged between those services.

3) Interoperability of artifacts shall be supported automatically over the SPL lifetime even if the development takes place at different times and disparate locations, thus implying support of multiple versions of the artifacts.

4) In support of correctness, data integrity, data security, and other data-related requirements across the multitude of possible SPL variations, constraints on model consistency and runtime checks shall be supported. Examples are dependency checks of the resulting instance data model (consistency) and authorization constraints for accessing data elements (runtime).

5) Desirable qualities, motivated by SPLE in general, should be supported including consistency, correctness, comprehension, maintainability, usability, efficiency, portability, integration, interoperability, reusability, testability, and traceability.

Although this case study comes from the eHealth domain, the issues are representative and applicable to data variability in SPLs in general.

3. Solution

This section provides a general description of the ADMV process and details on the utilization of fundamental concepts. The approach will then be illustrated by applying the ADVM to an eHealth SPL in Section 4.

3.1 ADMV-Process

The ADMV Process is an UML standards-based approach for SPL data modeling and data integration usable with common Model-Driven Software Development (MDSD) tooling, integrated with feature modeling, and supporting desirable software qualities during SPL development. Unified Modeling Language 2.x (UML2) class diagrams were selected for modeling due to the extensibility via stereotypes (in contrast, e.g., to the entity-relationship diagram) and the plethora of tools available to process the UML model further.

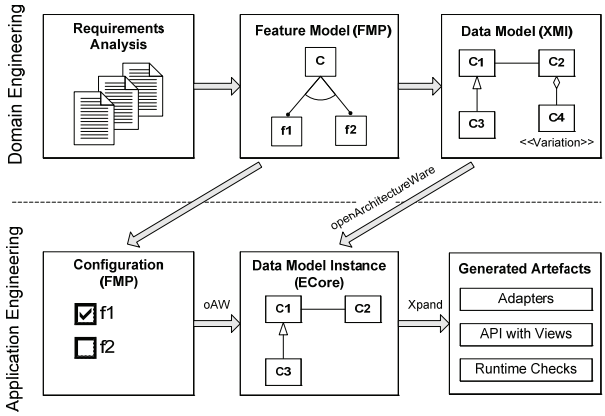

Figure 3.1. ADMV Process

The ADMV process (Figure 3.1) defines several steps in domain engineering and application engineering. These steps are:

1. **Requirements Analysis.** The ADMV starts in the Domain Engineering phase with requirements analysis. Through the analysis of the problem domain, common and variable requirements are collected.
2. **Feature Modeling.** Each variable requirement results in a String which is used as feature name. Dependencies of the features are analyzed and structured in a Feature Model (e.g., using FMP [23]).
3. **Data Modeling.** A Data Model is created in UML2 XMI (XML Metadata Interchange) [30] that includes variations. The first step before integrating variability is the definition of all the common parts. Then, for each feature, the variation points and variants are identified. Eventually the variants are associated with the variation points in connection with an adequate variability type. The ADMV addresses three types of variability: positive - adding new fields, data or relations to the core model; negative - eliminating fields, data, or relations from the core model; and structural - varying the type, cardinality, or naming of elements.
4. **Configuration.** At the start of the Application Engineering phase, a product configuration is created, e.g., in FMP.
5. **Artifact Generation.** Product artifacts are generated such as adapters, converters, views and runtime checks. To accomplish this, the current ADMV Generator implementation uses the

configuration in FMP as well as the Data Model as inputs (e.g., using openArchitectureWare (oAW) [12]) to create a Data Model Instance based on the ADMV metamodel (e.g., an Ecore metamodel [26]), from which the required code artifacts are generated (e.g., using Xpand, oAW's template language for code generation).

6. **Generated Artifact Customization.** Complex Conversions which the ADMV Generator cannot automatically resolve are implemented manually. In addition, the exceptions for the generated runtime checks are implemented manually to fulfill certain tasks when a runtime check fails.
7. **Artifact Integration.** Artifacts are integrated into the build of the data layer and other components.

3.2 Variability types

Negative variability. Negative variability starts from a maximal description (e.g., a UML (Unified Modeling Language) model containing all possible elements of the product line) and deletes the elements that are not connected to selected features. By this reduction, the final model of the selected product instance will be the result. Thus the complete model can be viewed, which may be advantageous if a product instance usually consists of mostly selected features such that the resulting model is close to the complete model (the delta to the complete model is small). On the other hand, it might result in information overload - especially if the product instances consist of only a few selected features such that the resulting models are small (the delta to the complete model is large).

Depending on the selected features, model elements can be removed to derive different product instances. This is reflected in the data model by tagging the different types with the stereotype <<Variation>>. The condition for which it is generated for the product instance is defined by the tagged value {feature = "any feature condition"}. This indicates to the generation process that the elements associated with the feature condition are only generated if the condition evaluates to true, otherwise they are removed.

This is called negative variability since the starting point is a superset of the data model definition and the unnecessary elements are stripped away according to the features selected.

Positive variability. In contrast to negative variability, positive variability starts from a minimal description (a core model, containing only the common parts) and, depending on selected features, additional elements (classes/members/associations) are added to the core model. The parts to merge are described in different places, which may make comprehension of the overall model difficult. This is especially true if there are many additional parts, which is often the case in non-trivial product lines.

Positive variability is useful if cross-cutting concerns should be modeled that cannot be effectively modeled by common base classes and negative variability. As this approach separates the data definition (class plus cross-cutting concerns described outside the class), it contradicts Requirement 1 in Section 2.1. The necessity and benefits in certain circumstances may be reasonable, but we recommend the technique be applied rarely, e.g., due to its potential negative effect on understandability. One technique for applying positive variability in an efficient way is described in [18].

Structural variability. Structural variability describes a change in the model dependent on some feature selection. The element is already contained in the model, but its structure (type, cardinality, association) may vary. Structurally changing the data model is achieved by adding the stereotype <<modify>> to the elements that should be structurally changed and by setting predefined tagged values. Possible tagged values are, e.g., feature, type, cardinality, name and initialValue.

In the resulting data model, the corresponding property is changed. This can also be used to redirect associations by changing the type of the association. An example is given in Section 4 regarding the introduction of additional folder structures due to the electronic case record (eCR) feature.

3.3 Check-Constraints

Constraints are a common concept in modeling and many approaches exist, for instance the Object Constraint Language (OCL). Constraints are used in many different ways: for consistency checks, such as the model itself (e.g., cardinality); for runtime checks (valid references, consistent instantiations); or for optimization [28].

Constraint checking and their languages are a known and powerful capability in assuring modeling correctness, which is especially important when supporting data model variability in a SPL. The ADMV encourages the application of constraint capabilities at the most appropriate points across the tools used in the process. For instance, feature modeling constraints can be utilized to determine the validity of a certain combination of features; data modeling constraints can be applied using active validation (e.g., via OCL or binaries as available in some UML 2 modeling tools) before transformation; transformation constraints can be applied to check conditions (e.g., ensuring that the domain and feature

model are not inconsistent with each other) before or during the generation process; and runtime checks can be automatically included by generators. Thus preconditions, invariants, and postconditions can be specified and carried through the process and applied at the appropriate points.

Consistency checks. The ADMV applies a variety of checks at different phases of the process to ensure the consistency of the models. At modeling time a UML2 tool applies constraints to check the validity of the data model. The uniqueness of members within a class is an example.

To ensure the consistency between the feature model and the data model, additional modeling time checks can be defined. When associating model elements to features, the checks can ensure that the feature model also contains these features.

Aside from ensuring model consistency, the model transformation for deriving product line instances must also be checked. This is done during the generation phase by applying, e.g., oAW-Checks. To ensure that the transformation was correct, oAW-Checks can test if the respective variation points are bound to variants and if the resulting data model is still valid after transformation.

Accessor constraints. To support the verification of certain conditions at run time, the ADMV additionally extends the support of the definition of constraint checks for accessor methods (also known as getters and setters). These constraints can be expressed with a constraint language such as oAW Check. The ADMV Generator transforms these constraints into methods which implement the oAW Check constraints. Every time a getter or setter is called, the associated constraints are successively executed. If one check fails, a runtime exception will be thrown. If all constraints are evaluated to true, the accessor method will be executed.

The analysis of the constraint-string is currently done in the ADMV implementation by the Xtend Parser which is part of the oAW framework. The Xtend Parser returns an abstract syntax tree (AST) which is the input for the ADMV Generator.

3.4 Views

View concepts are known from database systems, model-driven approaches, etc. The way views are considered in the ADMV is from the perspective of the view that a product-line component has on the data model. Certain components may be interested in viewing only parts of entities and shall be shielded from their further development because those components are considered stable and should not have to be adjusted just because the product-line data model changes.

A view is defined as a variant of an entity, which might be shared among several product line instances, or is specific to only one of these instances. An entity can have many views, each of them defining a set of child elements. All child elements have several attributes such as name, type, cardinality, etc. The definition of the view is done in the data model. For each entity there is exactly one complete view (which is the only one potentially persisted in a database) and an unlimited number of projected views. The complete view can be converted to any other projected view and vice versa. The child elements of the projected view can be arbitrarily filled with the source data. In this way it is possible to distribute the content of the source element over multiple elements of the projected view. Vice versa, it is possible to join the source element's data and assign it to a single element of the projected view. In addition, the datatype or properties of the target element can be different from the source element. Thus the structure and content of a projected view and the complete view can be disparate.

Functional components should be shielded from any differences in the data models, which can be achieved with adapters. However, manually written adapters place an additional burden on the developer: besides the initial development, they must be kept consistent with the changes in the data model over time. The ADMV models those adapters together with the data model and generates the code normally automatically – at least for members with the same name. For more difficult conversions, only the getter and setter are generated – the implementation must be added manually. To preserve manual code upon a later update of the data model with subsequent re-generation of the source-code, the Generation Gap pattern [27] may be applied. This is a step towards a consistent view on the data model over the whole SPL over time and it allows the exchange of data between components with different views on those artifacts.

Introducing "Views" gives those types of components a stable, reduced view on the data model. The actual designers and programmers need not be concerned about a variation; they are shielded by their view of an entity.

Note that if modules execute write access to the data, a reverse mapping from the projected view to the complete view must be defined.

Adapters. Adapters are based on the original data object of the product instance and provide a more stable view on the data for components that only require a subset. The adapters provide multiple data views to components and utilize a common data model,

thus conversions are required at runtime. In Figure 3.2a the conversion relationships between views to support the differing projected views desired by components are illustrated, with a maximum number of unidirectional converters required being n(n -1) where n is the number of views required. The maximum number of converters required can be reduced to a linear 2(n-1) if the direct conversions between projected views are avoided and only conversions to and from a complete view are utilized (a star topology) (see Fig 3.2b). This incurs a higher runtime cost of two conversions (once to the common view and then to the desired view) vs. only one, but benefits maintenance and evolution due to the reduced number of conversion methods. The runtime impact is dependent on the number of elements and complexity of conversion.

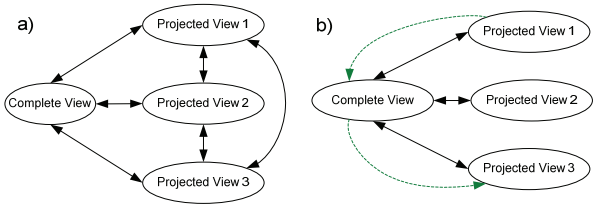

Figure 3.2. Projected view conversions

Data integration via adapters. The view concept supports integration in that it makes different formats and semantics explicit and generates the different adapters.

Advantages of this approach versus manual adapter implementation include the management of data structures from a single unified model, the retention and utilization of the core data model and its variability information by generators when conversion code is generated (the generators use the variability), and the automatic generation of conversion skeleton code and trivial body code (for simple conversions).

3.5 Artifact generation

The process of artifact generation is shown in the Figure 3.3. The data model and the configuration model are the input of the ADMV Generator. While any realization could be used, the current implementation is now described.

The two models are transformed by the template "models2Ecore" to a new data model which is based on an Ecore metamodel. The Ecore-based metamodel is less complex than the UML2 metamodel, making it is easier to define templates for transformation and generation. Initially the variability is not bound in the Ecore data model. It will be bound by the template "toProductModel". This is a model-to-model transformation where all variation points are bound to the configured variants, creating the data model for the configuration. The derived data model is the input for code generation. The views, adapters, and runtime checks are generated by Xpand.

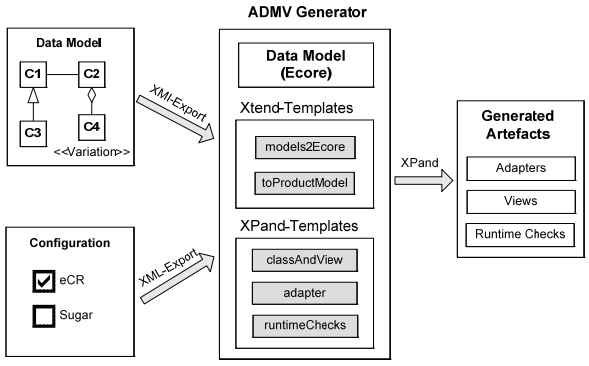

Figure 3.3. ADMV generation process

4. eHealth SPL example

Based on the scenario described in Section 2, the ADMV will now be illustrated.

After requirements analysis (Step 1), a feature model will be defined from the collected features (Step 2). This is the foundation for the product Configuration, defining how features can be combined during the configuration. Figure 4.1 shows the (reduced) Feature Model (FM) for the example domain using the Czarnecki-Eisenecker notation [4]. Hollow circles describe optional features, hollow arcs describe alternative features and filled arcs describe an "or"-relation (select one or more of associated features). A simplified form is used here, e.g., containing functional and non-functional features without explicit constraints, to show the possibilities of the ADMV.

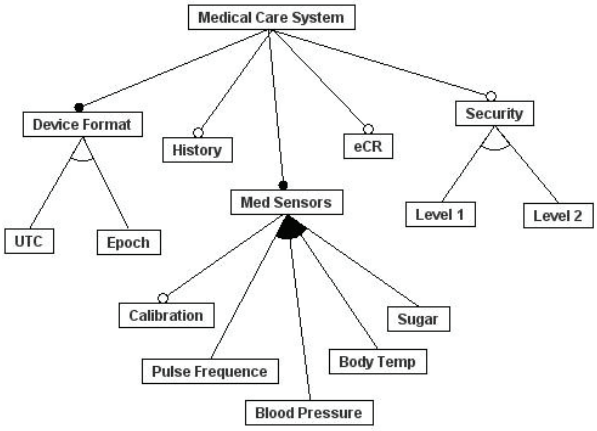

Figure 4.1. Feature model

Four topics were chosen to illustrate the approach: the change in the folder structure due to support of the eCR capability; support for various medical sensors and their different reporting formats (e.g. temperature in °C or °F); security in the sense of authorization of access to data; and a history of data changes. Different variability types can now be chosen to translate the related features into the solution space.

Figure 4.2 shows the FMP representation of the feature model from Figure 4.1 with an example product configuration (Step 4).

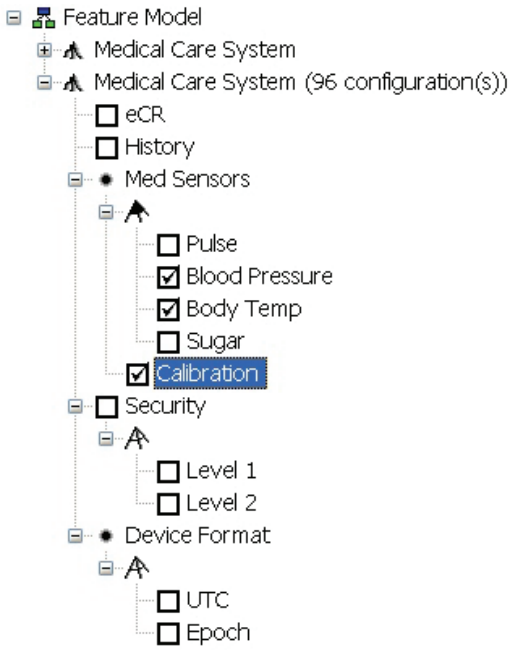

Figure 4.2. Feature model instance

Negative variability (Step 3)**.** Variation points that will be bound through negative variability are marked with the stereotype `<<Variation>>` on class, association, or member level. An appended condition (using bracket: {}) describes which features in the feature model must be selected in order for this part to appear in the data model of the product instance. Note that Boolean expressions are allowed, e.g., `Feature1 AND NOT Feature2`.

Negative variability is shown here to adapt the maximized data model to the resulting instance data model. Figure 4.3 shows the reduced data model.

The presented variation points are the four subclasses and the member `calibExp` (calibration expiration) of the superclass `Measurement`. Because Pulse and Sugar were not selected, the resulting data model shown in Figure 4.4 only contains the measurement types `BloodPressure` and `BodyTemperature`. Because the feature Calibration is selected, the resulting data model contains the member `calibExp`.

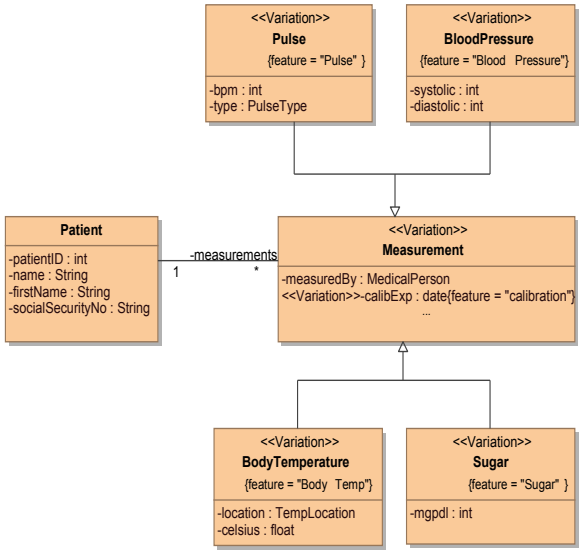

Figure 4.3. Data model with negative variability

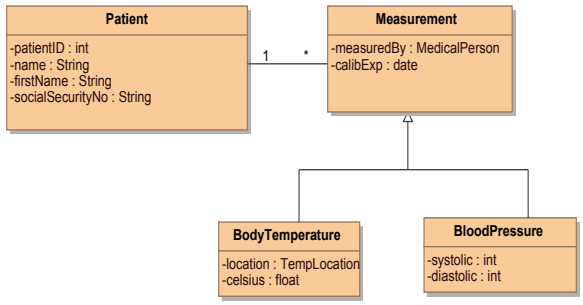

Figure 4.4. UML diagram of the example data model instance

Positive variability (Step 3)**.** Positive variable is demonstrated by the history feature, where the changes to each domain object over time should be tracked. Each domain object receives an additional member variable "history" containing previous entries and several operations. In ADMV, positive variability is realized by the stereotype <<add>> and the feature condition in brackets. Figure 4.5 shows an example.

The elements which will be added to the variation points by positive variability are composed in the class `HistoryElements`. To implement these elements via positive variability, the owner class is assigned with the stereotype <<add>> and the feature condition "History". The example reveals a scenario when positive variability is appropriate. Using negative variability to achieve the same behavior is more complex, especially the more members depend on the

cross-cutting feature: it would have to be repeated in each class that could potentially have that feature enabled and would have to be tagged with <<Variation>>. By choosing positive variability, the elements have to be modeled only once.

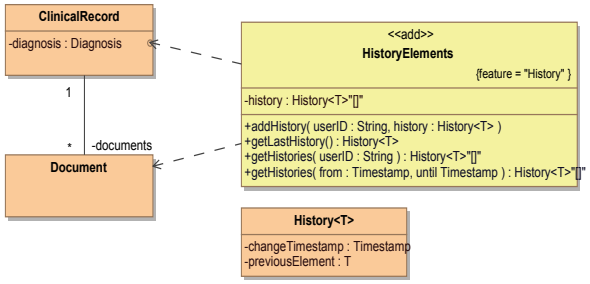

Figure 4.5. Positive variability

Structural variability (Step 3)**.** Structural variability is tagged using the sterotype <<modify>>, introducing the condition and type modification (again, in brackets: {}). For instance, see the association `records` from class `Patient` to `ClinicalRecord` in Figure 4.6.

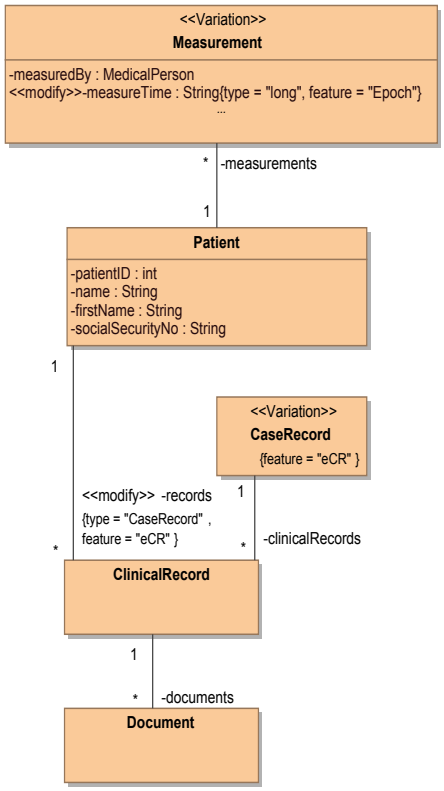

Figure 4.6. Data model with structural variability

In this example the eCR feature introduces an additional level for structuring the medical records, which is reflected by tagging the association from patient to clinical record as <<modify>> and redirecting the association to `CaseRecord` if feature "eCR" is selected. A second structural variation point is tagged to the member `measureTime` of the class `Measurement`. The date format is usually a `String`, but if "Epoch" is selected in the feature model, the date format will be a `long` (seconds since epoch).

Generated views (Step 5). The example of hierarchically differently structured patient information is defined by the structural variation point assigned to the reference `records` between `Patient` and the `ClinicalRecord` (see Figure 4.6). By default ("eCR" is not selected) records directly reference clinical records. Once the feature "eCR" is selected, the patient member `records` references `CaseRecord` which in turn references the `ClinicalRecord`. The two instances of the data model are shown for comparison in Figure 4.7.

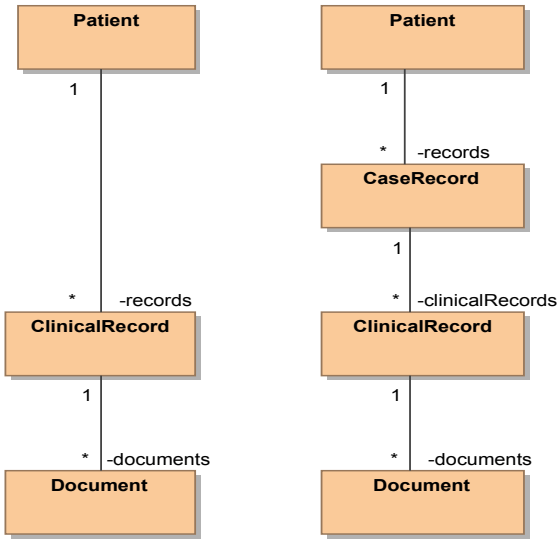

Figure 4.7. Data model results of structural variability

Generated data views for the structural variability example are shown in Listing 4.1 and different usage examples of generated data types are shown in Listing 4.2. The actual differences are written in **bold** font.

Listing 4.1

```
public interface Patient {
 List<ClinicalRecord> getRecords();
 void addRecords(ClinicalRecord value);
 void addRecords(List<ClinicalRecord>
       valueList);
    ...
}
```

```
public interface Patient {
  List<CaseRecord> getRecords();
  void addRecords(CaseRecord value);
  void addRecords(List<CaseRecord> valueList);
    ...
}
```

Listing 4.2

```
//default hierarchy
public void test1(Patient p) {
    p.getRecords().get(0).getDocuments();
}

//eCR 3-level hierarchy
public void test2(Patient p) {
   p.getRecords().get(0).getClinicalRecords()
     .get(0).getDocuments();
}
```

Consistency checks. To avoid inconsistencies in the generated artifacts, multiple checks which are defined in OCL and in the oAW Check language are executed. Listing 4.3 shows a simple oAW Check to verify the uniqueness of members. The check is applied to all attributes in the data model. If there are more than one equally named attributes within the same class, an error message informs the developer.

Listing 4.3

```
context Attribute ERROR "name not unique" :
((Class)eContainer).attributes
  .select(a|a.name == name).size == 1;
```

To verify the consistency between the feature model and the data model the following oAW check in Listing 4.4 is applied to all variation points.

Listing 4.4

```
context VariationPoint ERROR
"feature does not exist in feature model":
  getAllFeatures(featureModelUri())
  .contains( feature );
```

The function `getAllFeatures()` expects the path to the feature model as an input parameter which is resolved by `featureModelUri()`. It then returns the list of features of the feature model. Eventually the function `contains(..)` checks if the feature of the variation point in the data model is also contained in the feature model. If this evaluates to false it results in an error message.

Accessor constraints. The capability of defining runtime checks for accessors is illustrated with a security feature. The two security alternatives in the feature tree are bound to different access policies to measurements. Level 1 allows all the medical team personnel (usually, the staff in the same ward who cares for a patient) to access the measurements, Level 2 is stricter in the sense that each user may see only measurements made by themselves). Figure 4.8 shows the two examples of accessor constraints. Furthermore, accessor constraints can be associated with features as shown in Figure 4.8, again defined in brackets: {}.

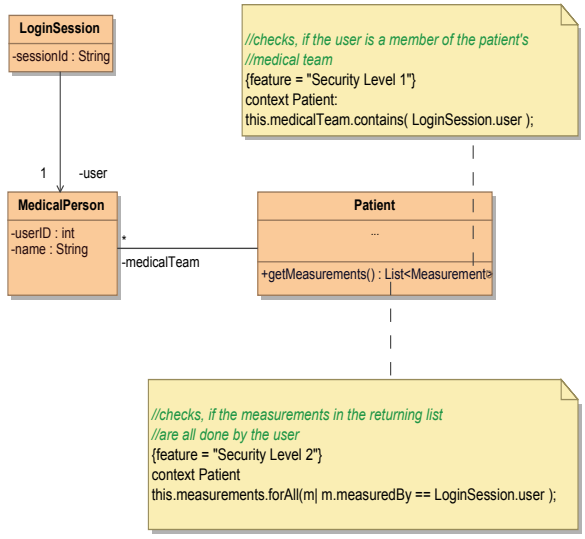

Figure 4.8. Accessor constraints

Run-time checks. Listing 4.5 shows the result of the generation process if "Security Level 1" is selected.

Listing 4.5

```
public void check1_getMeasurements()
   throws ConditionExceptionCheck1{
 if ( ! this.getMedicalTeam()
     .contains( LoginSession.getUser()) )
  throw new ConditionExceptionCheck1();
}

public List<Measurement> getMeasurements() {
 try {
  check1_getMeasurements();
 } catch (ConditionExceptionCheck1 e) {
  // resolve in an error message
 }
 return this.measurements;
}
```

The getter method calls `check1_getMeasurements()`, which checks if the logged-in user is a member of the patient's medical team. If this is not the case, then an exception will be thrown. The content of the catch block must be manually coded (Step 6) to perform further actions

such as logging, informing the user that he is not allowed to access the measurements, and returning `null`.

The authorization requirements may even be stricter. This is presented by the constraint which is associated to the feature "Security Level 2". It checks if the measurements in the returning list are all done by the logged-in user. The generated and manually added code for this check is depicted in Listing 4.6. Again the content of the catch block must be manually coded to the specific needs (Step 6), e.g., by filtering the returning list to fulfill the security rule. The manually added filtering is formatted in **bold**.

Listing 4.6

```
public void check1_getMeasurements()
   throws ConditionExceptionCheck1 {
 boolean cond = true;
 for (Measurement m : this.getMeasurements())
 {
  cond &= m.getMeasuredBy()
    == LoginSession.getUser();
 }
 if (! cond )
  throw new ConditionExceptionCheck1  ();
}

public List<Measurement> getMeasurements() {
 try {
  check1_getMeasurements();
 } catch (ConditionExceptionCheck1 e) {
  List<Measurement> filteredList
      = new ArrayList<Measurement>();
  for (Measurement m : this.getMeasurements())
  {
   if (m.getMeasuredBy()
         == LoginSession.getUser())
    filteredList.add(m);
  }
  return filteredList;
 }
 return this.measurements;
}
```

Adapters and views. The eHealth SPL contains components that operate on the clinical records independent of the context being an electronic case record infrastructure or a standard hospital. Instantiating the eCR data model would invalidate all code that uses the (simple) patient API. This illustrates the need for adapters.

The ADMV approach uses adapters to shield the actual designers and programmers from the differences in the instantiated data model. They need not be concerned about the "eCR" variation; the view – in this case – flattens the hierarchy of case records and resorts the clinical records together. Figure 4.9 shows the complete view to the left and the projected view to the right. The common elements are omitted for clarity.

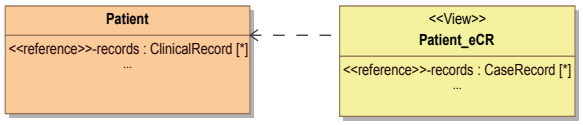

Figure 4.9. View of Patient

The ADMV Generator creates an adapter for the patient and two convert methods for each view to support bidirectional conversion (Step 5). Listing 4.7 shows a generated adapter and manually added code (Step 6) **in bold.**

Listing 4.7

```
public class Patient_eCR_Adapter
implements IPatient_eCR
{
  private IPatientView srcView;
  private IPatient_eCR adaptedView
           = (IPatient_eCR) new Patient_eCR();

  public Patient_eCR_Adapter
            (IPatientView srcView)
  {
    this.srcView = srcView;
    PatientViewConverter.convert(srcView,
                    adaptedView);
  }

  public List<CaseRecord> getRecords()
  {
    return adaptedView.getRecords();
  }
}

// there can be many convert methods
// depending on the no. of views
// the right method will be called via
// multi-method dispatching

public static void convert(Patient srcView,
   Patient_eCR targetView)
{
  // Bold code must be manually added
  CaseRecord cr = new CaseRecord();
  cr.setClinicalRecords(srcView.getRecords());
  targetView.setCaseRecords(cr);
  ...
}

public static void convert(...)
  ...
}
```

Using adapters for data integration. In case multiple devices deliver measurement data slightly differently, these must be converted to a specified core data structure. E.g., body temperature may be delivered as `value: int; scale: enum, celsius: float`, or `fahrenheit: int` from the different devices. The systems normative data structure assumes `celsius: float`, so all others need to be converted. New formats may arise at run-time too, e.g.,

when the hospital buys new devices or hospitals with different devices are merged.

The example in Figure 4.10 models a view containing a measurement type with a float representing the temperature in degrees Fahrenheit (Step 3). The resulting conversion types are generated (Step 5) and the bodies of the methods have to be added (Step 6).

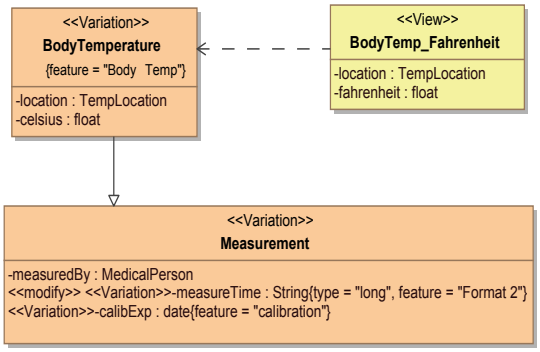

Figure 4.10. View of BodyTemperature

Listing 4.8 shows the conversion adapter for integrating measurement devices.

Listing 4.8

```
public static void convert(BodyTemp_Fahrenheit
    srcView, BodyTemperature targetView)
{
  targetView.setCelcius((
    srcView.getFahrenheit()-32)/1.8);
  …
}
```

Adapter generation. Listing 4.9 shows a simplified illustration of the Xpand-Template for the adapter generation. The input parameter for the adapter template can be any view. The French quotation marks « and » serve to distinguish between the static output and escaped control code that is interpreted. The instructions which fill the adapter template with model data are formatted here in bold.

Listing 4.9

```
«DEFINE adapterTmpl FOR View»
«FILE name + "_Adapter.java"»
public class «name»_Adapter
implements I«name»
{
  private I«entityName»View srcView;
  private I«name» adaptedView
           = (I«name») new «name»();

  public «name»_Adapter
            (I«entityName»View srcView)
  {
    this.srcView = srcView;
    «entityName»ViewConverter.convert(srcView,
                    adaptedView);
  }
  «FOREACH attributes AS a»
  public «a.type» get«a.name»()
  {
    return adaptedView.get«a.name»();
  }
  «ENDFOREACH»
  ...
«ENDDEFINE»
```

In order to convert the source view to the adapted view, the converter methods are generated (Step 5). This is done by the template in Listing 4.10. The converter template expects a complete view as an input parameter. The converter methods are generated in two steps: first all conversions from the complete view to the projected views are generated followed by all conversions in the reverse direction. When generating a convert method, it checks if the target attribute is also contained in the source view. If so, the conversion is a simple pass-through of data and can be generated automatically. Otherwise, it has to be implemented manually (Step 6).

Listing 4.10

```
«DEFINE converterTmpl FOR CompleteView»
«FILE name + "ViewConverter.java"»

public class «name + "ViewConverter"»{
«FOREACH views AS target»
  private void convert(«name
+ " src, " + target.name + " target" »){

  «FOREACH target.attributes AS attrib»
    target.set«attrib.name»(
    «IF attributes
       .select(e|e.name == attrib.name
        && e.type == attrib.type).size > 0 -»
      src.get«attrib.name»() );
    «ELSE-»
      null );
    «ENDIF-»
  «ENDFOREACH»
  }
«ENDFOREACH»
«FOREACH views AS src»
  private void convert(«src.name
  + " src, " + name + " target" »){
  ...
  }
«ENDFOREACH»
...
«ENDDEFINE»
```

5. Alternatives

This section considers various alternatives for dealing with data variability within the constraints set forth in Section 2.

UML2 package merge. The most viable alternative for data model variability is the "package merge" feature [29] introduced in UML2, and its usage for

SPLs has been evaluated [17][10]. Class extensions (e.g., additional members) can be modeled in a separate package that must have a merge association with the base package. The mapping is done on class name equality. Package merge is not suitable for the requirements described in Section 2 because it scatters the variation point over multiple packages. Thus the number of packages explodes and does not scale well with the number of features. Package merge cannot model negative or structural variability that is needed for requirement 2.

To compare the ability of both approaches, the number of classes to model was counted. The Measurement class and its subtypes result in five classes, four of which are tagged to be variable, two members are also tagged.

Using package merge, a core model containing only the base class Measurement without the members `calibExp` and `measuredTime` was used. For each subtype, a package was created since each of the subtypes is selectable separately. Within the package, the base class is repeated to add the child class and the association between them. For the optional calibration management, a package is added, repeating the base class with the member `calibExp`. For the structural variability of the `measuredTime`, two packages are needed, one repeating the base class with the `long` type and one with the `String` type. This requires up to seven additional packages and 12 classes. Additionally, data model comprehension becomes difficult since the information is spread across many packages (see Figure 5.1).

A general comparison of the effort involved in the two approaches is shown in Table 5.1, where:

F = number of features influencing the data model

$V(f)$ = number of variation elements for a single feature f

$C(f)$ = Number of classes created or modified by feature f (might be less then $V(f)$ in case a feature controls more than one member of a class)

Table 5.1. UML Package Merge vs. ADMV for packages, attributes, and classes

Approach	Packages	Attributes created or modified	Repeated Classes
UML package merge	F	$\sum_{f=1}^{F} V(f)$	$\sum_{f=1}^{F} C(f)$
ADMV	0	$\sum_{f=1}^{F} V(f)$	0

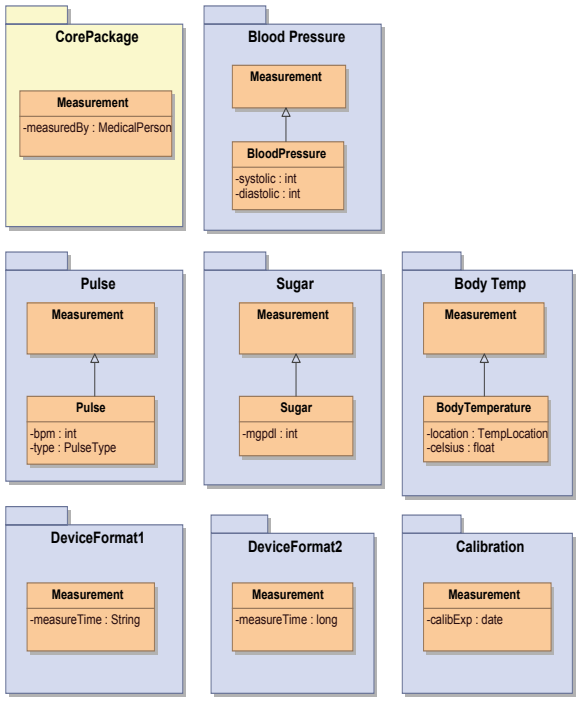

Figure 5.1. UML Package Merge

Optional members. Negative variability could be modeled by using a full-blown data model for each instance and returning "null" in case a non-selected member or association is requested. Alternatively, hashmaps could be used to carry (single-valued) optional members. Shortcomings of this approach include:

- Members cannot be declared to be not null, in case the feature is selected and null is an inappropriate value (especially if the data model is persisted in databases).
- The development of all components could accidentally use members that are not necessarily selected. Auto-completion and compile-time checks are not possible.
- Using hashmaps gives developers no indication about available members.
- Structural variability is not possible.

Explicit dependencies. Each extension to the data model could be presented by a separate data component and explicitly used by a functional component (see Figure 5.2). The data components retrieve the necessary elements to form their view on a domain data. Communicating with other components introduces the obligation for the receiving component to retrieve their view of the data again.

Drawbacks include the numerous calls for database retrievals per component due to a lack of sharing, as

well as interpretation difficulties when components transmit or receive data via references or value objects. If transactions are considered or the services are remote, this solution is infeasible.

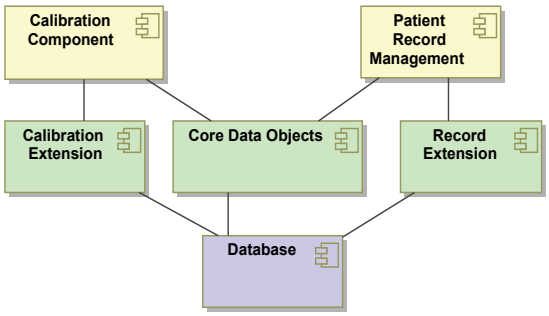

Figure 5.2. Data model with explicit dependencies

Layering. Similar to the Decorator design pattern [6], components are grouped in layers that correspond to the level of enrichment of the data (see Figure 5.3). Per layer, one definition of each data element exists. On the lowest level this will be a core element, on the next level a slightly enriched element (some more attributes or associations) that can even be extended on higher levels. If a component of a higher layer needs data, it asks the persistence component of that layer to retrieve it. The persistence component routes this request down the layers and extends ("enriches") the data, converting the data into its own layer data model.

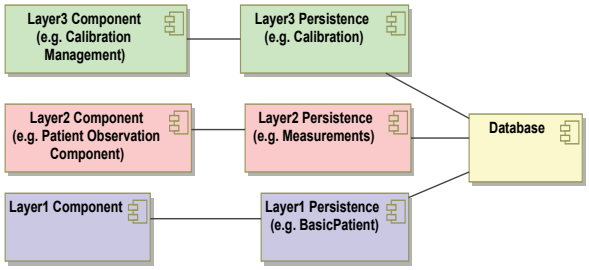

Figure 5.3. Data model with layered extensions

Multiple calls have to be executed to provide the Layer 3 component with the data and there is insufficient support for transaction handling.

In summary, the aforementioned alternatives have the disadvantage that the data model is not presented consistently and that knowledge is spread over at least some layers or even individual components. This assumes that a strict layering is even possible. The ADMV does not create separate information in separate packages nor does it need to repeat classes as UML package merge does. Feature-dependent classes are defined once in both approaches (see the `Measurement` children).

6. Evaluation

For an evaluation of the ADMV, an appraisal of its support for desirable qualities is considered. Additionally, any practical limitations of the approach with regard to performance and scalability with current implementations are also assessed.

6.1 Quality properties

Consistency and correctness. Correctness is supported via constraint checks and the generation of adapters and projected data views appropriate for a component in its current version. Via the validation of the data model at usage time via OCL constraints (e.g., MagicDraw Active validation), various modeling errors can be detected sooner and thus avoided in later phases. Consistency checks can assure the consistency of the models, e.g., between the feature model and data model. Support for the correctness of the generated artifacts is thus enhanced.

Comprehension. ADMV reduces the number of classes and locations where (redundant) information is stored, which furthers comprehension. Code generation is based on a metamodel specialized for modeling data variability. Code generation templates can thus be more simply created compared to UML metamodel generative approaches such as OMG's Model-Driven Architecture (MDA).

Maintainability. Maintenance and evolvability are supported by both shielding component developers from changes via adapters as well as the application of constraints throughout development. By programming templates against a common ADMV metamodel, an unlimited number of future templates and template changes support any necessary extensibility.

Usability. Usability is fostered by the integration of ADMV in standard modeling (FM and UML) and with tool frameworks that support customization (e.g., oAW). The usage of constraint languages at the appropriate levels also furthers usability.

Efficiency. The enhanced support for code generation techniques has the potential to improve efficiency for larger SPLs. Runtime efficiencies are also achievable since variation decisions are typically made at generation time. The reduction in the number of classes required to deal with variability also promotes efficiency.

Portability. Modeling variability with UML-based stereotypes, coupled with the ADMV metamodel as a

basis for generation, supports the portability and exchangeability of MDSD implementations for modeling, model-to-model transformations, and artifact generation.

Data integration and interoperability. These qualities are supported via the adapters and projected component views that support independent conversions. The complete view also supports interoperability across the SPL and product instance life cycle.

Reusability. Component reuse is supported since no direct tight coupling to other components via data elements occurs. Enhanced comprehension enhances reusability opportunities. Templates reuse common code.

Testability. Constraints can be readily defined via very capable languages such as OCL and oAW Check, which supports the testability of the models. The reduced number of classes also simplifies component testing, since knowledge of other existing components in the individual product instances is not required.

Traceability. The modeling of variability and data in a central model makes the effects of the variability more traceable. By using UML tools with stereotypes and tagged-value-based search capabilities (as in MagicDraw), the traceability of variation points and features is improved. Certain variation points can be localized by simple string searches.

6.2 Performance and scalability

Due to the use of code generation techniques, the impact of the variations and the use of the adapters at runtime is relatively inconsequential. View conversion of data where necessary, e.g., from one format to another, is currently a manual programming task and thus the runtime impact is dependent on the conversion complexity. However, due to the large set of possible permutations and the reliance on MDD, variation scalability measurements were made to determine the impacts of the variations for development time usage of the ADMV.

The measurements were performed on an AMD Athlon XP 2400+ (2GHz) PC with 3GB RAM running Microsoft Windows XP Pro SP2, Java JDK 1.6, Eclipse 3.3, openArchitectureWare 4.2, and the Eclipse Modeling Framework 2.3. All measurements were performed 3 times and the averages presented.

For the first set of measurements, the transformation time using oAW from an XMI Data Model file containing variations to a Data Model Instance XMI (all variation points applied based on features) was measured as shown in Table 6.1 and Figure 6.1. A nearly linear correlation between a change in the number of variation points and the generation time was measured as the number of features was held constant, and an increase in the number of features also showed a nearly linear increase in the generation time. This result is explained by the iterations in the generator code implementation for each variation point and for each feature. Varying the number of Boolean conjunctions up to 20 for a variation point made no perceptible difference due to other inherent overheads.

Table 6.1. Data model instance transformation time (ms) for features and variation points

Number of variation points	Total number of features		
	300	600	900
50	2771	5281	9141
100	4429	9696	17781
150	6416	14219	26078

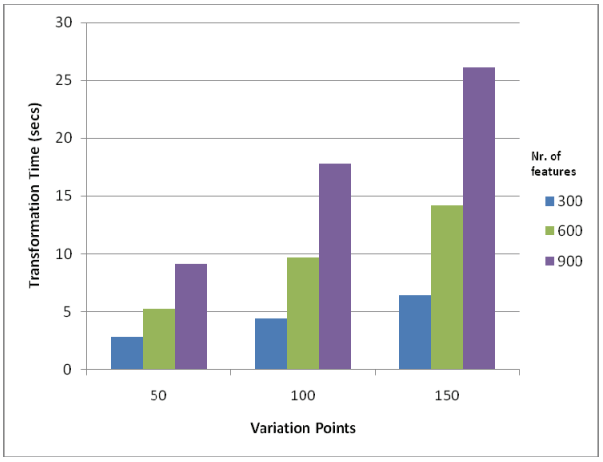

Figure 6.1. Data model instance transformation time vs. variation points and features

A second set of measurements concerned the generation of adapters. Each of the different variability types was tested and, as expected, no noticeable difference in generation time occurred based on the negative, positive, or structural variability types. In the ADMV, each adapter for an entity can support multiple projected views. The Lines of Code (LOC) generated in support of the conversion betweens views increased in the same percentage to the number of views, as expected due to the 2n relation resulting from the complete view basis for all conversions. The maintainability of the conversions is thereby supported. The generation time required for adapters with multiple views is shown in Figure 6.2, showing a nearly linear increase as the number of adapters or views increase. The generation time for this scale

appears reasonable for development usage in current SPLs.

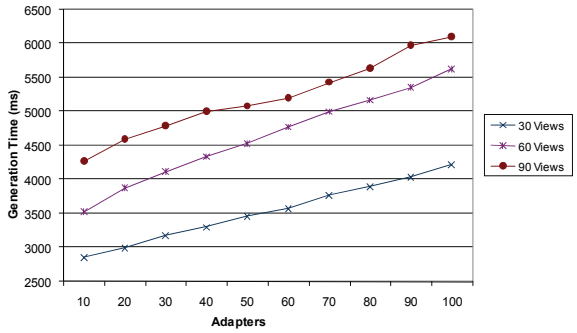

Figure 6.2. Generation time vs. number of adapters

In summary, the development-time variation scalability and performance of the ADMV with current tooling for industrial use is shown to be practicable.

7. Related work

Other approaches for SPL variability in data models include the conceptual framework SPLIT [3], where additional UML stereotypes, e.g., <<variabilityMechanism>> and <<variationPoint>>, are used for specifying variable elements. SPLIT does not, however, integrate an abstract feature view as does the ADMV, and each variation point and the corresponding variants requires a separate class which can cause issues in lucidity for large SPLs.

Clauß presents in [24] a generic modeling approach which uses additional stereotypes to express variability. The Stereotype <<optional>> is used for optional variants which do not stand in a relationship with other variants (variation point with one variant). Variation points which group multiple variants together are tagged with the stereotype <<variationPoint>> and the associated variants with <<variant>>. Furthermore, the variation points and variants can be assigned with tagged values to define certain properties. Some of these properties are the binding time of variants, the multiplicity of associable variants, and the condition of binding. However, this approach doesn't offer a concept to handle data independently from the corresponding product instance, nor does it address the derivation of product line instances.

In [7], PLUS (Product Line UML-Based Software Engineering) extends UML to model variability and commonality using stereotypes and primarily subclassing. While entities are mentioned, the wrappers described are intended for database access and do not support all variation types and multiple view and data versions for components as addressed in the ADMV. The extension of PLUS with the ADMV would provide a more comprehensive solution for SPL UML techniques.

In MDD-AO-PLE [15][16][18] and similar related aspect-oriented SPLE work, the application of techniques to SPLs are investigated for addressing cross-cutting variability. While this work has not specifically addressed the difficulties described in this paper for data models, the combination of these techniques with ADMV could be synergistic, e.g., to address positive variability or for common data view format conversions in adapters.

The following comparison matrix shows a assessment of related SPLE approaches in regard to a selection of requirements.

Table 7.1. Comparison matrix

	SPLIT	PLUS	MDD-AO-PLE	UML ext. [24]	ADMV
requirement analysis	+++	+++	++	+	++
FM[1] integration		✔	✔	✔	✔
positive variability	✔	✔	✔	✔	✔
negative variability				✔	✔
structural variability	✔	✔		✔	✔
UML2	✔	✔		✔	✔
data conversion[2]					✔
checks (modeling)	✔	✔	✔	✔	✔
checks (config.)[3]			✔	✔	✔
checks (generator)			✔	✔	✔
checks (runtime)					✔
product instantiation[4]	+++	+	+++	+	+
code generation			✔		✔
trace variability[5]	✔	✔	✔	✔	✔

([1]) FM = feature model.

([2]) Ability to convert data to different formats.

([3]) Checks at configuration time.

([4]) The process of creating a specific software product using a software product line is referred to as product instantiation [25].

([5]) Ability to trace variability between solution space and problem space.

Work with regard to SPL component evolution support includes [5], where a multi-team decentralized SPL variability modeling approach is described, supporting the merging of model fragments. However, it does not address versioning of entities and component usage and lacks UML support. [22] addresses multi-context component reusability using UML extensions views (functional, static, and dynamic), but does not consider data modeling, constraints, or code generation issues.

Work on model-based data integration, mapping, and transformation in the eHealth domain includes [21] and the AutoMed project [20]. To our knowledge the usage of such an approach for an eHealth SPL for modeling data variability has not been explored.

8. Conclusion and future work

Given the inadequate integration and specific support for MDSD data modeling and variability in current SPL approaches and research, the ADMV contributes a UML standards-based method for data modeling that can be utilized by common MDSD tooling, is integrated with feature modeling, and supports desirable software qualities during SPL development. UML diagrams are augmented with variability information including constraints, from which artifacts for particular configurations can be generated automatically. The approach for adapter generation supports SPL data integration with the potentially multifarious external systems and devices, which may represent the same kind of information in different formats

An eHealth case study that motivated the work was used to illustrate the application of the ADMV to a SPL. Scalability of the ADMV with regard to features and variation points is linear and likely to be sufficient for typical current SPL development. The unification of concepts and mechanisms in ADMV promote support for desirable SPLE qualities, including consistency, correctness, comprehension, maintainability, usability, efficiency, portability, integration, interoperability, reusability, testability, and traceability. These and other benefits can be realized for SPLs in conjunction with the ADMV.

Future work includes the addition of a conversion language for somewhat complex conversions in adapters (e.g., concatenation and regex-split). Support of dynamic runtime variation including adaptation and binding of component views with database migration is another area to be investigated. Additionally, optimization for object tree transfers and greater automatic adapter data conversion code generation are promising.

9. References

[1] Bartholdt, J., Oberhauser, R., and Andreas Rytina, "An Approach to Addressing Entity Model Variability within Software Product Lines". In *Proceedings of the Third International Conference on Software Engineering Advances (ICSEA 2008)*, IEEE Computer Society Press, 2008.

[2] Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., and DeBaud, J., "PuLSE: a methodology to develop software product lines," In *Proceedings of the 1999 Symposium on Software Reusability* (SSR '99). ACM, pp. 122-131.

[3] Coriat, M., Jourdan, J., and Boisbourdin, F., "The SPLIT method: building product lines for software-intensive systems," In *Proceedings of the First Conference on Software Product Lines: Experience and Research Directions* (Denver, Colorado, United States). P. Donohoe, Ed. Kluwer Academic Publishers, Norwell, MA, 2000, pp. 147-166.

[4] Czarnecki, K. and Eisenecker, U.W., *Generative Programming: Methods, Techniques, and Applications.* Addison–Wesley, May 2000, ISBN 0201309777.

[5] Dhungana, D., Neumayer, T., Gruenbacher, P., and Rabiser, R., "Supporting the Evolution of Product Line Architectures with Variability Model Fragments," In *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008) (February 18 - 21, 2008)*. WICSA. IEEE Computer Society, Washington, DC, 2008, pp. 327-330.

[6] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995, ISBN 0-201-63361-2.

[7] Gomaa, H., *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures,* Addison-Wesley, 2005, ISBN 0201775956.

[8] Griss, M. L., Favaro, J., and Alessandro, M. d. "Integrating Feature Modeling with the RSEB," In *Proceedings of the 5th international Conference on Software Reuse (June 02 - 05, 1998)*. ICSR. IEEE Computer Society, Washington, DC, 1998, p. 76-85.

[9] Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E. & Peterson, A. S. "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie-Mellon University Software Engineering Institute, 1990.

[10] Laguna, M. A., González-Baixauli, B., and Marqués, J. M., "Seamless development of software product lines," In *Proceedings of the 6th international Conference on*

Generative Programming and Component Engineering (GPCE 2007). ACM, New York, NY, pp. 85-94.

[11] Linden, F.J. v.d., Schmid, K., and Rommes, E., *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*, Springer, Berlin, ISBN 3540714367, 2007.

[12] Voelter, M., Haase, A., Kolb, B., and Efftinge, S., "Introduction to openArchitectureWare 4.1.2," In *Proceedings of the Model-Driven Development Tool Implementers Forum* (MDD-TIF07), TOOLS EUROPE 2007.

[13] Batini, C., Lenzerini, M., and Navathe, S. B., "A comparative analysis of methodologies for database schema integration," ACM Computing Surveys (CSUR), Vol. 18, Issue 4, (Dec. 1986), 323-364.

[14] Pohl, K., Böckle, G., Linden, F.J. v.d., *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer-Verlag, 2005, ISBN 3540243720.

[15] Voelter, M. and Groher, I., "Handling Variability in Model Transformations and Generators", in *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling* (DSM'07), Sprinkle, J., Gray, J., Rossi, M., Tolvanen, J.-P., (eds.), Computer Science and Information System Reports, Technical Reports, TR-38, University of Jyväskylä, Finland 2007, ISBN 978-951-39-2915-2.

[16] Voelter, M. and Groher, I., "Product Line Implementation using Aspect-Oriented and Model-Driven Software Development," In *Proceedings of the 11th international Software Product Line Conference* (September 10 - 14, 2007). International Conference on Software Product Line. IEEE Computer Society, Washington, DC, 2007, pp. 233-242.

[17] Laguna, M. A., González-Baixauli, B., and Marqués, J. M., "Seamless development of software product lines," In *Proceedings of the 6th international Conference on Generative Programming and Component Engineering* (Salzburg, Austria, October 01 - 03, 2007). GPCE '07. ACM, New York, NY, 2007, pp. 85-94.

[18] Groher, I., "Aspect-Oriented Feature Definitions in Model-Driven Product Line Engineering", Dissertation, Johannes Kepler Universität, Linz, April 2008.

[19] Boehm, O., "eCR Application Architecture v1.2 Services and Interfaces", Fraunhofer Institute for Software and Systems Engineering (ISST), www.fallakte.de 2008

[20] Smith, A. and Mcbrien, P., "A Generic Data Level Implementation of ModelGen," In *Proceedings of the 25th British National Conference on Databases: Sharing Data, information and Knowledge* (Cardiff, Wales, UK, July 07 - 10, 2008). A. Gray, K. Jeffery, and J. Shao, Eds. Lecture Notes In Computer Science, vol. 5071. Springer-Verlag, Berlin, Heidelberg, 2008, pp. 63-74.

[21] Ying, B., Rong, Z., and Xiao, J., "A Data Integration Approach to E-Healthcare System". In *Proceedings of the 1st International Conference on Bioinformatics and Biomedical Engineering, 2007 (ICBBE 2007)*, pp. 1129 – 1132.

[22] Saidi, R., Front, A., Rieu, D., Fredj, M., Mouline, S., "From a Business Component to a Functional Component using a Multi-View Variability Modelling," In *Proceedings of the International Workshop on Model Driven Information Systems Engineering: Enterprise, User and System Models (MoDISE-EUS'08)* held in conjunction with the CAiSE'08 Conference, Montpellier, France, June 16-17, 2008, ISSN 1613-0073, pp. 34-45.

[23] Antkiewicz, M. and Czarnecki, K., "FeaturePlugin: feature modeling plug-in for Eclipse," In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Exchange* (Vancouver, British Columbia, Canada, October 24 - 24, 2004), eclipse '04, ACM, 2004, pp. 67-72.

[24] Clauss M., "Generic modeling using UML extensions for variability", In *Proceedings of the Workshop on Domain Specific Visual Languages*, OOPSLA 2001, Jyväskylä University Printing House, Jyväskylä, Finland, 2001, ISBN 951-39-1056-3, pp. 11-18.

[25] Van Gurp, J., Bosch, J., and Svahnberg, M. "On the Notion of Variability in Software Product Lines," In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (August 28 - 31, 2001)*. WICSA. IEEE Computer Society, Washington, DC, 2001, pp. 45-54.

[26] Eclipse Modeling Framework Project, http://www.eclipse.org/modeling/emf/ May 14, 2009.

[27] Vlissides, J., "Generation Gap," C++ Report Volume 8, Number 10, November / December, 1996, pp. 12-18.

[28] Generic Eclipse Modeling System (GEMS), http://www.eclipse.org/gmt/gems/ May 14, 2009.

[29] OMG, "UML 2.1.2 Superstructure Specification", OMG doc# formal/07-11-02, 2007.

[30] OMG, "Meta Object Facility(MOF) 2.0 XMI Mapping Specification, v2.1.1", OMG doc# formal/07-12-01.