Formalizing Distributed Evolution of Variability in Information System Ecosystems

Hendrik Brummermann\textsuperscript{1}, Markus Keunecke\textsuperscript{2}, Klaus Schmid\textsuperscript{2}

\textsuperscript{1}Hochschul-Informations-System GmbH, Goseriede 9, 30159 Hannover, Germany
brummermann@sse.uni-hildesheim.de

\textsuperscript{2}Software Systems Engineering, Institute of Computer Science, University of Hildesheim, Germany
\{keunecke, schmid\}@sse.uni-hildesheim.de

Please cite this publication as follows:

The corresponding \texttt{BibTeX}-entry is:
\begin{verbatim}
@INPROCEEDINGS{BrummermannKeuneckeSchmid12a,
    author = {Hendrik Brummermann and Markus Keunecke and Klaus Schmid},
    title = {Formalizing Distributed Evolution of Variability in Information System Ecosystems},
    year = {2012},
    pages = {11--19},
    publisher = {ACM},
    doi = {10.1145/2110147.2110149},
}
\end{verbatim}

ACM, 2012. This is the authors version of the work. It is posted here by permission of the ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS ’12), DOI: 10.1145/2110147.2110149.
Formalizing Distributed Evolution of Variability in Information System Ecosystems

Hendrik Brummermann
Hochschul-Information-System GmbH
Goseriede 9
30159 Hannover, Germany
brummermann@sse.uni-hildesheim.de

Markus Keunecke
University Hildesheim
Institut fuer Informatik
Marienburger Platz 2
31141 Hildesheim, Germany
keunecke@sse.uni-hildesheim.de

Klaus Schmid
University Hildesheim
Institut fuer Informatik
Marienburger Platz 2
31141 Hildesheim, Germany
schmid@sse.uni-hildesheim.de

ABSTRACT
The open variability of software product line ecosystems allows customers and third party organizations to create extensions to a system which may refine the variability model. In this paper we will describe an approach to evolution support, which was developed in the context of one specific company, HIS GmbH. However, the approach is much more generic than this. In particular, it is based on the formalization of modifications to configuration values and constraints on both the model and the data in the context of the evolution of multi-level configurations. Our approach supports the identification of inconsistencies in evolution.

Categories and Subject Descriptors
D.2.7 [Distribution, Maintenance, and Enhancement]; D.2.13 [Reusable Software]: Reuse Models; H.3.5 [Online Information Services]: Web-based services

General Terms
Design, Management, Experimentation

Keywords
Variability Modeling, Evolution, Software Product Lines, Software Ecosystem

1. INTRODUCTION
Evolution in software ecosystems poses a number of challenges [5], which we are going to address by formalizing configuration values and constraints.

While traditional product line approaches focus on the development within a single company [6][19], more recent work extended the product line approach to product populations [20] and even further to software ecosystems [4]. In software ecosystems the situation is more complex than in traditional product line engineering, because software can be developed by multiple organizations semi-independently. Therefore the development happens to some extent without explicit synchronization. In particular, a customizable software platform may be built by one organization, while another organization builds software on top of it without explicit information flow back to the first one [15].

Figure 1 describes a simplified situation focusing on only two organizations. Based on an original version ($V_n$), the development organization releases a new version of the platform ($V_n'$) after some time. The changes done by third parties and customers ($V_{n+1}$) need to be merged with the evolved version $V_n'$. This may lead to inconsistencies as there is no cross-organizational coordination. It is important to predict the consequences of such evolution steps in order to automatically or manually resolve conflicts before the new version is taken into production.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VaMoS ’12, January 25-27, 2012 Leipzig, Germany
Copyright 2012 ACM 978-1-4503-1058-1 ...$10.00.
Our configuration modeling approach is based on decision modeling, but modifies it to address concepts and particular issues of this situation.

As an important step towards predicting the consequences of an update and assisting in the conflict resolution, we define a formalism for partial configurations and their composition. It is usable for both composing partial configurations of extensions and new partial configurations provided by updates. Furthermore, our approach supports the prediction of update consequences. It can detect inconsistencies before the new system is taken into production and assign blame to the appropriate parts of a configuration.

The main challenge, we address in this paper, is the development of a formal approach to support multi-level configuration of products in a software product line ecosystem. An additional requirement is that each intermediate level of configuration should already result in a fully configured and thus usable system.

The remainder of this paper is structured as follows: Section 2 describes the context and gives a summary of the arising issues. In the following section, we provide a formal definition of partial configurations and their composition before we look at evolution scenarios in Section 4. In Section 5 we will discuss related work, while Section 6 provides the conclusion and discusses further work.

2. CONTEXT

While our paper provides a novel and generic solution, it is inspired by a specific company context, which we describe in this section. We introduce the company and its open development model before we provide a formalized abstraction of real world configuration types in the following section. We conclude this section with a typical example, that we will use in further sections of this paper to illustrate our formalism. This example should be seen only as a reference example of an ecosystem. Our approach explicitly focuses on the generic situation.

2.1 Case Study

Our context is a German university management system called HISinOne, which is the core of its own software ecosystem. The open development model of the development organization HIS GmbH enables universities to adapt and extend HISinOne independently to fit their needs. Often customers contract the solution department of HIS or independent service providers to aid with the configuration. Third party vendors develop extensions or alternatives for parts of the HISinOne system with which HISinOne has to work together at the customers’ sites.

The software needs to be very flexible because requirements among customers are quite different: Customers range from small universities of applied science to huge universities focusing on research. Furthermore Germany is divided into 16 regional states with different laws regarding the regulation of universities. The different levels of autonomy require flexibility that reaches deeply into the core of the system. For example validation and calculation rules and business processes need to be adaptable.

HIS provides a base configuration that is common to all installations. Furthermore it provides different sets of partial configurations specific to segments of customers. The example in Figure 2 shows a typical situation with a third party vendor adding its own configurations parameters and partly redefining existing configurations values. The customers typically do many changes to different areas of the system and may add their own parameters as well. Sometimes they define a second partial configuration that they enable in special situations or use for experiments. The effective configuration is a composition of the parts in the order they are defined.

2.2 Configuration types

HISinOne has a number of different configuration types that define the variability of various aspects of the system. According to Bosch this is common for real world systems [3].

Table 1 lists the important configuration types in HISinOne and the number of entries in the base system. The numbers are lower bounds of the solution space variability because not all variation points are easily identifiable. The current system consists of configurations that are partly based in the problem space and partly based in the solution space. Furthermore the complete problem space is not known to HIS as third party organizations may extend or modify the system in unforeseen ways. HIS therefore tries to provide a very flexible basis for the solution space.

HIS distinguishes the following configuration types for practical reasons:

Explicit global variability points are defined in a decision tree and affect the system as a whole such as the name of the university.

Decentralized variability points allow different effective values at the same time, depending on the area they are used in. For example, the application form looks different depending on requirements defined by the course of study. This kind of variability was omitted from Table 1 because there is no easy way to count it.

Layout and Design are mostly implemented by using the web technology Cascading Style Sheets. Customers often adapt it to their cooperate design by changing the color schema for example. Changes can be very complex and completely redefine the layout.

User interface texts are managed outside the user interface definitions because they need to be translated into different languages and adapted to specific vocabulary of the customer. User interface texts are used for both labels of input fields and long help texts.

<table>
<thead>
<tr>
<th>Configuration type</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit global variability points</td>
<td>872</td>
</tr>
<tr>
<td>Layout and Design</td>
<td>2905</td>
</tr>
<tr>
<td>User interface texts</td>
<td>10117</td>
</tr>
<tr>
<td>Explicit default constraints</td>
<td>689</td>
</tr>
<tr>
<td>Explicit required constraints</td>
<td>185</td>
</tr>
<tr>
<td>Implicit constraints</td>
<td>8749</td>
</tr>
<tr>
<td>Tables</td>
<td>669</td>
</tr>
<tr>
<td>Columns</td>
<td>5124</td>
</tr>
<tr>
<td>Rows of default reference data</td>
<td>162135</td>
</tr>
<tr>
<td>Sum</td>
<td>191445</td>
</tr>
</tbody>
</table>

Table 1: Number of variation points in HISinOne as of 16th October 2011
Explicit default constraints are applied to both operational data and configuration values, but can be overridden by customers. For example, many universities use lecture numbers which are configured as mandatory fields by default.

Explicit required constraints are useful to enforce technical and law-based restrictions such as requiring a registration number for students. Unlike default constraints, customers may not remove them.

Implicit constraints are derived from the data model and applied automatically in the service and user interface layers. For example, an attribute that is defined as integer will not accept a text literal and a field with a not-null database constraint is mandatory.

Tables and columns are used to store the data and customers can add their own tables and columns. Default and required constraints can be defined for any column.

Default reference data consists of reference data such as a list of all universities or configuration information for algorithms.

Using these specialized configuration types seems reasonable from an administrator’s point of view. On a more abstract level they show huge similarities concerning the composition and merging of partial configurations on updates. They can be grouped into configuration values and constraints.

HIS distinguishes between constraints that are required by all installations and default constraints. An example for a required constraint is: “If it is a German university, the registration number of students is a mandatory field” (because it is required by law). Customers are not able to disable this constraint. The number of a lecture, however, is treated differently for domain reasons: Many universities want their lectures to have short visible numbers in order to be able to refer to them easily. But a small number of universities do not use this number. HIS provides a default constraint “the lecture number is a mandatory field”. Unlike the constraint on registration number, customers may remove this constraint on the lecture number.

In this paper, we focus on explicitly defined variability including user interface texts and layout definitions. Our approach will work for decentralized variability points, too, if they are identifiable by some kind of name.

2.3 Example from the applicant management domain

In this section, we will describe an example from the applicant management domain which we will use as a running example throughout this paper. Applicant Management is a part of the system that customers commonly change. First, we describe the base configuration of the system as delivered by HIS. Then we give an example of a customer modification. Our scenario ends with a realistic update of the base configuration.

E1 In a base configuration the system provides a wizard for applicants to enter the data which is required for their application such as contact information and qualifications. The dialogues contain a number of input fields and help texts. For example, there is an input field for the grade of the school certificate, which is visible (app.grade.visible) and has a mandatory default constraint (app.grade.mandatory).

E2 Now a customer implements the university management software at his site and adopts it to fit his needs. Here we assume that the customer is an Art School. It does not need a grade of the school certificate and disables the constraint which forced the grade to be mandatory. The customer even goes a step further and hides the input field from the user interface (app.grade.visibility).

To make the dialogue better understandable for the applicants the customer also changes the help text (app.help.text) to explain the application requirements for the Art School.

E1' After some time the customer wants to update to the newest version released by the development organization. But in the meantime changes to law occurred requiring the grade of school certificates for statistical reasons. The development organization has therefore evolved the software: The constraint making the grade of school certificates mandatory was taken from a default constraint to a required one.

This results in a conflict: The Art School configuration is no longer allowed to set the grade to not mandatory.

In the next section, we will describe our approach to dealing evolution problems like this. As a basis for this, we will first define in a strictly formal manner our concept of a configuration.

3. COMPOSING CONFIGURATIONS

In this section, we will introduce configuration modeling, which is similar to decision modeling but allows meta-variability [16] in the sense that the variation model itself is variable, as constraints may be redefined. Partial configuration consists of values and constraints that can be defined and redefined on every level. After defining a composition of partial configurations we conclude this section with an approach for detecting inconsistencies.

3.1 Value Configurations

In this section, we formally define partial value configurations, which have been described informally in Section 2.
Value configurations assign values to configuration variables. Therefore we define:

**Definition 1.** The set of all configuration variables is called \( \mathcal{V} \).

Due to the distributed nature of software ecosystems the subset of all used configuration variables is not known to the development company, because customers may add their own configuration variables.

**Definition 2.** The set of all possible configuration values is called \( \mathcal{W} \).

For simplicity, we do not distinguish multiple types. An extension could be made easily.

**Definition 3.** Let \( V \subseteq \mathcal{V} \) be a subset of configuration variables, then \( C_{\text{value}}(V, c : V \rightarrow \mathcal{W}) \) is called a partial value configuration, if \( c : V \rightarrow \mathcal{W} \) is a total function assigning values to configuration variables.

We refer to values assigned in a configuration as default values, because in a composition later configurations may override them. After all partial configurations are composed, the remaining set of default values is effective.

**Definition 4.** A partial value configuration \( C_{\text{value}}(V, c : V \rightarrow \mathcal{W}) \) is called a full value configuration over a variable set \( V' \) if \( V = V' \).

The example in Section 2.3 defines that the dialog for the application has the help text “Welcome to the application wizard” and a visible input field for grades with the help text “Grade of school certificate”. In our formalism this can be described as

\[
C_{\text{value}} \left\{ \text{app.grade.visible} \right\} \rightarrow \left\{ \begin{array}{l}
\text{app.helptext = "Welcome to the application wizard"}, \\
\text{c(app.grade.helpertext) = "Grade of school certificate"}, \\
\text{c(app.grade.visible) = true}
\end{array} \right.
\]

**3.2 Constraint Configurations**

Unlike traditional variability modeling approaches, in our approach customers are able to modify existing constraints and add new ones. Derived projects are not restricted to monotonous extensions. Thus, constraints may not be considered a fixed part of a configuration, resulting in a kind of meta-variability.

Another extension is that constraints are applied to both configuration values and data in the database. For example many universities assign visible numbers to lectures for easy reference. Therefore the development company provides a base configuration in which the lecture number is mandatory. But a customer may decide not to use lecture numbers and remove this constraint.

In addition to these default constraints, there are required constraints that may not be modified because they are required for technical or domain-specific reasons. For example, the grade of the school certificate is required by statistic law in Germany, so a German university may not remove this constraint.

Default and required constraints can exist on all levels of partial configurations, not just the base configuration. This requires that we must be able to redefine constraints. We use names to identify constraints as a later defined default constraint may override an earlier one.

**Definition 5.** The set of all names is called \( \mathcal{N} \).

\( \mathcal{N} \) is analogous to the set of all variables \( \mathcal{V} \) in value configurations. Analogous to the set of all values \( \mathcal{W} \), we define a set of all possible constraints.

**Definition 6.** The set of all possible constraints is called \( \mathcal{P} \).

For example HISinOne currently supports the following constraint types:

- value must be provided
- ranges defined by minimum and maximum numbers
- length limit of text fields
- formats defined by regular expressions
- additional datatypes such as “academic-term” and “date-in-the-past”
- programmed special purpose and multi-field constraints such as “the number of years between the day of birth and the day of application must be below a configured value” that can be plugged in

Multiple constraints are combined using an AND-semantics. Our concept, however, is not limited to the above types. A partial constraint configuration consists of default constraints and required constraints. Default constraints are uniquely identified by names, so that replacement or modification of constraints can be identified. Required constraints, however, cannot be replaced but only added. Therefore the same identifier may refer to multiple constraints. The identifiers are still useful, however, because they allow to keep track of constraints.

**Definition 7.** Let \( N_d \subseteq \mathcal{N} \) and \( N_r \subseteq \mathcal{N} \) be two sets of names, then \( C_{\text{constraint}}(N_d, N_r, d : N_d \rightarrow \mathcal{P}, r : N_r \rightarrow 2^\mathcal{P}) \) is called a partial constraint configuration, if \( d \) is a total function assigning default constraints to names and \( r \) is a total function assigning a set of required constraints to names.
In example E1, the grade is a mandatory field because of a default constraint:
\[ C_{E1} = (\)
\[ N_d = \text{\{app.grade.mandatory\}}, \]
\[ N_r = \{\}, \]
\[ d(\text{app.grade.mandatory}) = \text{grade is mandatory} \)
\]
For simplicity, we assume that partial constraint configurations are consistent within themselves.

### 3.3 Configurations

A partial configuration consists of both the value and the constraint components:

**Definition 8.** Let \( C_{\text{value}}(V, c : V \rightarrow W) \) be a partial value configuration and \( C_{\text{constraint}}(N_d, N_r, d : N_d \rightarrow P, r : N_r \rightarrow 2^P) \), then we call \( C(V, N_d, N_r, c : V \rightarrow W, d : N_d \rightarrow P, r : N_r \rightarrow 2^P) \) a partial configuration. The set of all partial configurations is named \( \mathbb{C} \).

Note, that constraints can force a specific value for a given configuration variable, so there is no need for required values as an explicit component of configurations.

Applying our formalism to the examples given in Section 3.2 results in the configurations shown in Table 2:

- The initial base configuration (E1) created by the development organization defines some help texts and a visible input field for grades.
- The customer’s local customization (E2) hides the input field for grades and removes the constraint. It changes the help text.
- The evolved base configuration (E1’) defines some help texts and a visible input field for grades. The constraint making the grade mandatory is required and may not be removed.

### 3.4 Consistency Analysis

In this subsection, we define the requirements for consistent configurations. Before we look at the detection of inconsistencies, we define how constraints are evaluated.

#### 3.4.1 Evaluation of Constraints

As explained in Section 2.2 constraints can be applied to configuration values and data in a database. A database consists of records, therefore we define:

**Definition 9.** We call \( \text{Rec} \) the set of all possible data records.

For simplicity and without loss of generality, we look at a database as the product of all tables (\( \text{Table}_1 \times \text{Table}_2 \times \text{Table}_3 \ldots \)).

**Definition 10.** A database is a non empty subset of \( \text{Rec} \) consisting of homogeneous records with the same cardinality and type-structure.

An installation consists of a configuration and a database.

**Definition 11.** Let \( C \) be a configuration and \( \text{Data} \) be a database, then \( I = (C, \text{Data}) \) is called an Installation. The set of all installations is named \( \mathbb{I} \).

For the enforcement of constraints in an installation, we do not need to distinguish between default and required constraints. Therefore we can define a function \( \pi \), which extracts all constraints from a given configuration and puts them into one joint set.

**Definition 12.** Let \( C(V, N_d, N_r, c : V \rightarrow W, d : N_d \rightarrow P, r : N_r \rightarrow 2^P) \) be a partial configuration. Then we define a predicate extraction function \( \pi : \mathbb{C} \rightarrow 2^P \) with \( \pi(C) = \{d(n_d)|n_d \in N_d\} \cup \{p|p \in r(n_r), n_r \in N_r\} \).

In order to detect certain inconsistencies, we have to apply predicates to installations and configurations:

**Definition 13.** Let \( I(C, \text{Data}) \) be an installation with a configuration \( C(V, N_d, N_r, c : V \rightarrow W, d : N_d \rightarrow P, r : N_r \rightarrow 2^P) \) and \( \text{Data} = \{rec, |i \in \mathbb{N}\} \), then we define the evaluation of a predicate \( p \) as predicate function \( p(c, rec) \). If \( p \) is independent of data, we may use a boolean function \( p(c) \) instead.

Using the above definitions, we can now define what consistency means for our kinds of constraints.

#### 3.4.2 Inconsistency Detection

The inconsistencies, we found in practice, can be categorized as follows:

- **CR1** configurations values must comply with constraints
- **CR2** data must comply with constraints
- **CR3** the set of constraints must not be inconsistent

We will look at each of these requirements in turn:

- **Configurations values must comply with constraints**
  
  An example for the inconsistency of values (CR1) is hiding the input field for grades, although there is a constraint making the grade mandatory.

- **Data must comply with constraints**
  
  For the second category (CR2), we assume there is an applicant record in the database that does not have a grade.

- **The set of constraints must not be inconsistent**
  
  Constraints may contradict each other in a configuration, thus it is impossible to fulfill them all at once (CR3).
∀( inconsistent constraints, if: 
check we call a function 
fine:

Summary
To summarize the three categories of inconsistency we define:

Definition 16. Let $C$ be a configuration, then there are inconsistent constraints, if:
\[
\forall (c: v \to W): 3p \in \pi(C): p(c) = false \text{ or } \forall rec \in Rec, V(c: v \to W): 3p \in \pi(C): p(c, rec) = false
\]

Table 2: A formal representation of configuration examples

<table>
<thead>
<tr>
<th>Effective configuration</th>
<th>Evolved effective configuration</th>
</tr>
</thead>
</table>
| $C_{E1}$: Original configuration by development org.  
$C_{E1} = (V = \{app.helptext, app.grade.helptext, app.grade.visible\}, N_d = \{app.grade.mandatory\}, N_r = \{\}, c(app.helptext) = \text{“Welcome to the application wizard\”,} c(app.grade.visible) = \text{false,} d(app.grade.mandatory) = \text{true} \})$ | $C_{E1'}$: Evolved configuration by development org.  
$C_{E1'} = (V = \{app.helptext, app.grade.helptext, app.grade.visible\}, N_d = \{\}, N_r = \{app.grade.mandatory\}, c(app.helptext) = \text{“Welcome\”,} c(app.grade.helptext) = \text{“Grade of school certificate\”,} c(app.grade.visible) = \text{true,} r(app.grade.mandatory) = \{true\} \})$ |

| Partial configuration by the customers  
$C_{E2} = (V = \{app.helptext, app.grade.visible\}, N_d = \{app.grade.mandatory\}, N_r = \{\}, c(app.helptext) = \text{“Welcome to Aster Art School\”,} c(app.grade.visible) = \text{false,} d(app.grade.mandatory) = \text{false} \})$ | unchanged, see left |

Table 3: Effective configurations after composition of the configurations shown in Table 2

$C_{E1} \circ C_{E2} = (V = \{app.helptext, app.grade.helptext, app.grade.visible\}, N_d = \{app.grade.mandatory\}, N_r = \{\}, c(app.helptext) = \text{“Welcome to Aster Art School\”,} c(app.grade.helptext) = \text{“Grade of school certificate\”,} c(app.grade.visible) = \text{false,} d(app.grade.mandatory) = \text{false} \})$

$C_{E1'} \circ C_{E2} = (V = \{app.helptext, app.grade.helptext, app.grade.visible\}, N_d = \{\}, N_r = \{\}, c(app.helptext) = \text{“Welcome\”,} c(app.grade.helptext) = \text{“Grade of school certificate\”,} c(app.grade.visible) = \text{true,} r(app.grade.mandatory) = \{true\} \})$

Definition 17. Let $I(C, Data)$ be an installation, then we call a function check : $I \to \{true, false\}$ consistency check defined as follows:
check(I) = false, if I has at least one of the 3 inconsistencies
check(I) = true, otherwise

The interesting entries of an inconsistent installation are those that cause the inconsistency:

For example there may be a complex partial configuration which defines many values and constraints. The following minimized configuration already shows an inconsistency without having to take other values and constraints into account:
\[
C(V = \{\}, N_d = \{app.grade.length, app.grade.smaller\}, N_r = \{\})
\]

A minimal inconsistent installation is the smallest installation with a problem. Therefore removing any entry from this installation makes it consistent. In other words: There can be no other sub installation which is still inconsistent.

Definition 18. Let $I(C, Data)$ be an installation with $C(V, N_d, N_r, c : V \to W, d : N_d \to P, r : N_r \to 2^P)$. Then we call I minimal inconsistent installation, if:
check(I) = false and $\forall I': \text{check}(I') = true$ and $I'(C', Data')$ with $C'(V', N'_d, N'_r, c' : V' \to W, d' : N'_d \to P, r' : N'_r \to 2^P)$ is an installation with $V' \subseteq V, N'_d \subseteq N_d, N'_r \subseteq N_r, c' \subseteq c, d' \subseteq d, r' \subseteq r, I \neq I'$

Please note, that the condition $I \neq I'$ forces $I'$ to be a truly smaller installation in at least one of the components. The minimal inconsistent installation is not uniquely specified. Nevertheless it can be a very useful tool in practice, because it reduces the amount of work for administrators significantly:

In case of an inconsistency an administrator has to resolve the problem. A typical configuration consists of hundreds of configuration variables and constraints as shown in the table.
in Section 2, thus minimizing the number of entries, that have to be examined manually.

3.5 Composition

As discussed in Section 2 there is a sequence of partial configurations starting with a base configuration provided by the development company and ending with the last configuration by the customer. There may be multiple partial configurations in between to configure the system for different kinds of customers.

Later configurations in the sequence can redefine default values and default constraints and add their own ones as shown in Figure 2. They can also add required constraints, but may not remove or modify them as shown in Figure 3.

**Definition 19.** Let C1 and C2 be partial configurations defined as follows:

\[ C_1(V_1, N_{d1}, N_{r1}, c_1 : V_1 \rightarrow W_1, d_1 : N_{d1} \rightarrow P, r_1 : N_{r1} \rightarrow 2^p) \]

\[ C_2(V_2, N_{d2}, N_{r2}, c_2 : V_2 \rightarrow W_2, d_2 : N_{d2} \rightarrow P, r_2 : N_{r2} \rightarrow 2^p) \]

Then a composition \( \circ \) is defined as: \( \circ (C_1, C_2) = C(V, N_d, N_r, c : V \rightarrow W, d : N_d \rightarrow P, r : N_r \rightarrow 2^p) \)

with:

- \( V = V_1 \cup V_2 \)
- \( N_d = N_{d1} \cup N_{d2} \)
- \( N_r = N_{r1} \cup N_{r2} \)
- \( \forall v \in V : c(v) = c_2(v), \text{ if } v \in V_2, c_1(v) \text{ otherwise} \)
- \( \forall n \in N_r : d(n) = d_2(n), \text{ if } n \in N_{r2}, d_1(n) \text{ otherwise} \)
- \( r(n) = r_1(n) \cup r_2(n) \)

\( C_1 \circ C_2 \) is said as \( C_1 \) before \( C_2 \).

Table 3 shows the result of composing our examples E1 with E2 and E1’ with E2:

Composing the examples E1 and E2 results in the input field for grades being hidden and help texts being changed according to the customer configuration. The composition of E1’ and E2 is very similar, but has an additional required mandatory constraint on the grade. This example shows that composing two consistent configurations may result in an inconsistent configuration.

**Definition 20.** Let \( S = (C_1, C_2, \ldots, C_n) \) be a sequence of partial configurations, then we call \( C_E \) the effective configuration over \( S \), if \( C_E = C_1 \circ C_2 \circ \ldots \circ C_n \). For simplicity, we may call \( C_E \) the effective configuration.

4. EVOLUTION

This section describes how our approach introduced in Section 3 helps in the actual evolution situations between two configurations.

First, we will introduce differences between configurations and how they help in the analysis of error situations. Furthermore, we will explain how the inconsistency detection from Section 3, helps predicting the consequences of an update.

4.1 Difference between two configurations

The evolution of configurations may lead to problems as described at the end of Section 3. In such a case, it is important to identify the effective changes that occurred as a basis for determining the problem source. Here, we will formalize the notion of the difference of two configurations.

**Definition 21.** Let \( C_1 \) and \( C_2 \) be partial configurations defined as:

\[ C_1(V_1, N_{d1}, N_{r1}, c_1 : V_1 \rightarrow W_1, d_1 : N_{d1} \rightarrow P, r_1 : N_{r1} \rightarrow 2^p) \]

\[ C_2(V_2, N_{d2}, N_{r2}, c_2 : V_2 \rightarrow W_2, d_2 : N_{d2} \rightarrow P, r_2 : N_{r2} \rightarrow 2^p) \]

Then the difference \( \Delta : C_2 \times C_1 \) is defined as:

\[ \Delta(C_2, C_1) = ( \]

\[ V_{\text{added}} = V_2 \setminus V_1 \]

\[ V_{\text{modified}} = \{ v | v \in V_1 \cap V_2, v \in c_1(v) \} \setminus c_2(v) \}

\[ V_{\text{deleted}} = V_1 \setminus V_2 \]

\[ N_{d_{\text{added}}} = N_{d2} \setminus N_{d1} \]

\[ N_{d_{\text{modified}}}(n) = \{ n | n \in N_{d1} \cap N_{d2}, N_{d1}(n) \neq N_{d2}(n) \} \]

\[ N_{d_{\text{deleted}}} = N_{d1} \setminus N_{d2} \]

\[ r_{\text{added}} = N_{r1} \setminus N_{r2} \]

\[ r_{\text{deleted}} = \{ n | n \in N_{r1} \cap N_{r2}, N_{r1}(n) \neq N_{r2}(n) \} \]

The difference can be used to explore all changes in a new version of a partial configuration. This is especially useful as a diagnostic tool because it helps the administrator to find the root cause of problems quickly, that may show up in the production system after a configuration change or extension. Note, that the difference operator is aware of the structure of configurations. Unlike line-based difference tools it abstracts from syntactic changes as discussed in 2.

On updates, it is important to identify effective changes on an installation. These can be calculated by applying the difference operator to the original and updated effective configuration. This will hide changes to variables and constraints that have been overridden, as they are not part of the effective configuration.

In the example shown in Table 2, the development company changes a default constraint to a mandatory constraint. Applying the \( \Delta \) operator to these partial configurations will result in:

\[ \Delta(C_{E1}, C_1) = ( \]

\[ V_{\text{added}} = \{ \}

\[ V_{\text{modified}} = \{ \}

\[ V_{\text{deleted}} = \{ \}

\[ N_{d_{\text{added}}} = \{ \}

\[ N_{d_{\text{modified}}}(\text{app.grade.mandatory}) \]

\[ N_{d_{\text{deleted}}} = \{ \}

\[ r_{\text{added}} = \{ \}

\[ r_{\text{deleted}} = \{ \}

This identifies the key changes, which the customer’s administrator has to look at in detail.

4.2 Consistency

Consistency of installations is especially important in the context of evolution. While the composition of consistent partial configurations can result in an inconsistent effective configuration, this is usually not an issue during the initial deployment because the local customizations are developed with the base configuration in mind. Therefore issues are resolved during the customer’s development and a lengthy testing process ensures that the system fulfills the customer’s needs before it is taken into production.

Updates, however, are merge situations: Although an old partial configuration is simply replaced by a new one, the partial configurations following in the sequence are now applied to the new configuration resulting in a different effective configuration. It is desirable to spend less time on qual-
ity assurance compared to the test during the development of the initial customization. The new base configuration and the old customer configuration may be in conflict on some parts.

Table 3 shows the effective configurations of our example. The original customer configuration \( E_1 \) leading to the effective configuration \( E_1 \land E_2 \) was developed with the base configuration in mind. Therefore the customer was able to set the undesired default constraint \( \delta(app.grade.mandatory) \) to \( false \). After some time the development company released version \( E_1' \).

In general there are three possible outcomes of changes:

**Conflict:** \( E_1' \) contains a required constraint \( r(app.grade.mandatory) \) set to \{true\}. The composition of \( E_1' \land E_2' \) results in a new effective configuration which requires the grade to be both mandatory and not mandatory at the same time. This is an inconsistency as defined in Section 3.3 because the constraints are contradicting each other.

**Overridden:** Another change done by the development company in \( E_1' \) is the value “Welcome” for the configuration variable \( app.help.text \) which used to be “Welcome to the application wizard”. This change is not part of the effective configuration because the university defined its own title “Welcome to Aster Art School”.

**Effective:** In practice most changes, however, make it into the effective configuration because they are related to new functionalities or customers have not modified the affected variables or constraints. Therefore the minimal inconsistent installation as defined in Section 3.4 is a valuable tool to focus on the problematic parts.

As a customer is not able to modify the partial configuration provided by the development company, it is helpful to highlight those parts of the minimal inconsistent installation that the customer may change. Note that for conflicts, in which default constraints are involved, overriding that constraint is a valid option.

The same mechanism can be used by the development company to identify changes before a release that will have a negative impact on customer installations, if the development company has access to (anonymized) copies of customer installations.

5. RELATED WORK

The context of our work is a software ecosystem according to the definition given by Bosch [4] or Schmid [15].

Van Ommering [29] introduced the notion of product populations. This is related to our work as we include an open variability model which is defined in a distributed way. In contrast to the situation discussed in this paper, Van Ommering assumes a central authority controlling the product lines.

Krueger describes an approach to partial instantiation of software product lines with updates, feedback and cross-merges in [11]. In our situation feedback from customers is not complete and cross-merges are the common case. In [12] Krueger describes mass customization and automatic derivation of products. This is related to our situation because customers basically create their own products based on a configuration infrastructure.

Czarnecki et al. [9] introduce the concept of staged configuration with multi-level configuration as a special form with separate feature models for each stage. Classen et al. [7] formalize the process of staged multi-level configuration. Staged configuration is described as a process that prunes down the number of possible decisions with each configuration stage, while our approach allows to extend and redefine the variability model at each level. Stoiber et al. [18] also refers to staged configuration and discusses the impact of constraints in a stepwise configuration process. Although Stoiber considers the consistency of products, the aspects considered in his paper do not cover evolution in a distributed scenario as we do here.

Mazo et al. [13] discuss the validation of variability represented as decision models. This is also important in our context, but their setting is not distributed and with central coordination. Furthermore, we go one step further by validating existing data with the variability model.

Ganev et al. [10], Cavalcanti et al. [6] and Anastasopoulos et al. [1] discuss the evolution of variability models. They focus on centralized situations. Apel et al. [2] discuss distributed evolution with explicit synchronization. In our context, however, evolution occurs distributed across multiple organizations with little or no synchronization.

Similar issues exist outside the area of information systems: The approaches of the Eclipse and Debian communities are discussed by Schmid et al. in [15]. Both the Eclipse and Debian community are good examples for current industrial practice, although their focus is not on information systems.

At the database level Siegmund et al. in [17] and Schaeler in [14] describe approaches for modularization and merging of fragments. Nonetheless their findings on database modularization do also apply in a distributed context. In contrast to our situation both papers do not consider possible issues for evolution.

6. CONCLUSION

We defined a formalism for the composition of variability in software ecosystems and showed how this helps to address the distributed product line evolution.

Our approach is based on partial configurations, which may be provided by different organizations such as the development company, third party vendors and customers. A partial configuration consists of modifiable configuration values, modifiable constraints and required constraints. Constraints may be defined on both configuration values and data in the database of an installation.

Later configurations in the sequence are not limited to “fill in the gaps”, but may extend the system and redefine parts that have been provided by previous partial configurations. This allows the development company to provide default constraints, that are useful for many customers, while enabling customers to override them to achieve their desired behavior. This goes well beyond existing approaches to product line variability, which often do not provide characteristics such as default values, composition of partial configurations or constraints on database information. However, all those characteristics are a must for the kind of products and the development environments, we are concerned with.

Our approach describes in a strictly formal manner how partial configurations can be composed and how inconsistencies are detected. Our formalism supports structured
difference calculations, which is especially helpful to fix issues very quickly in a production system that occurred after a configuration change.

After an update provided by the development company or a third party vendor, those changes have to be merged with the local customization done by the customer. A full installation consists of a huge amount of data and a large number of configuration values and constraints. Therefore we showed how a minimal inconsistency set is calculated, which lets administrators focus on the important parts of the configuration in order to resolve the conflicts.

While we provided a formal basis, much work still needs to be done: To validate our formalism we are going to implement it in tools. We have already implemented the composition of configurations and the difference calculations for some configuration types such as user interface texts and CSS layout rules. We will continue our work by implementing the detection of inconsistencies in installations before they go into production and support administrators by identifying the minimal changes required to repair the system.

7. REFERENCES


