2015 41st Euromicro Conference on Software Engineering and Advanced Applications

Bridging the Gap between Software Variability and System Variant Management: Experiences from an Industrial Machinery Product Line

Stefan Fischer*, Lukas Linsbauer*, Roberto E. Lopez-Herrejon*, Alexander Egyed* and Rudolf Ramler†
* Johannes Kepler University Linz, Austria
{stefan.fischer, lukas.linsbauer, roberto.lopez, alexander.egyed}@jku.at
† Software Competence Center Hagenberg GmbH, Austria
rudolf.ramler@scch.at

***Abstract*—Companies that develop complex systems often do so in the form of product lines, where each product variant can be configured to a certain degree to fit a customer's specific requirements. Features cannot be combined arbitrarily in a product line. The knowledge which features require or exclude each other is represented in form of variability models. Unfortunately, in practice, such variability models do not exist or they are oriented towards the needs and viewpoints of specific organizational units, e.g. sales, manufacturing, hardware engineering, or software development. In this paper we present our experiences in building a variability model for the highly configurable software part of a complex mechatronic system produced by one of our industrial partner companies. The company already had support and processes for product variant management in place for sales and hardware manufacturing. However, the corresponding variability model was at the level of the overall system and excluded the variability of the software part. The paper discusses the resulting problems and challenges and describes the approach we selected to bridge the gap that existed between product variants and software configurations. The goal and driving motivation for our work was the improvement of the software development process and specifically the testing of software variants. The paper also shows how software configuration and testing activities can benefit from an appropriate variability model.**

I. Introduction

The concept of product lines has been used in traditional manufacturing industry for a long time. The same concept is now also applied in software development in the form of Software Product Lines (SPLs), which are a systematic software reuse approach [1], [2]. SPLs have been shown to provide benefits like reduced time-to-market, reduced costs or increased quality [2]. A core concept of SPLs is the composition of product variants from a set of given features (i.e. increments in functionality) in order to fit different customers' requirements. This ability to adapt is called *variability* and is represented in the form of variability models (most commonly feature models) that express which features can be combined to form a product variant and which ones cannot. Conceptually a variability model is a set of constraints formulated on the features of a product line that must hold for a valid product variant.

Software is becoming more important now in every domain, also in the traditional manufacturing industry whose focus used to be entirely on hardware not too long ago. We observed that a significant portion of the know-how is now in the software of the built machinery, and not just in the mechanical and electrical components anymore, as can for instance clearly be observed in the automotive domain. Due to this fact, the product line concepts of traditional manufacturing and software development in such companies fuse together into an overall product line concept. A product variant of such a product line has both, significant portions of hardware as well as software that both need to be able to consistently adapt and implement the desired variability of the product line. Achieving this is far from trivial as it embodies several challenges which we will discuss.

This paper summarizes our experiences in working with an industrial partner, a internationally leading machine manufacturing company with locations in different countries and continents. While our initial perspective of this work centered around the needs of the software development team (around 30 software engineers), we realized that no discussion on a system's variability could do without also understanding the needs of the hardware development team working on the mechanical and electrical parts (mechatronics engineers in this case) and the members from the sales team (which better than others understand the customers' requirements). For confidentiality issues, we cannot name our industrial partner or any details about it. Therefore all examples used in this paper are synthetic but have been chosen carefully to still reflect the actual situation we experienced.

The motivation for our work was to support the company in establishing a variability model for the software system and to improve the testing process for software variants. There are several SPL testing approaches out there that require variability information to be available [3], [4]. Our goal therefore was to gather (variability) information scattered across various organizational units, relate and unify this information and make it available and useful for the software development team and - more specifically - for testing software variants. Ultimately a consistent concept of a product line and its variability should be established across the whole company.

The remainder of this paper is structured as follows: Section II illustrates the initial situation that we found at the company along with the problems that we identified. Section III explains the approach that we took to tackle the problems by integrating the existing models and concepts of variability. Section IV shows the results we achieved, describes

978-1-4673-7585-6/15 $31.00 © 2015 IEEE
DOI 10.1109/SEAA.2015.57

applications based on tools we developed, and discusses some lessons learned from the project. Section V provides an overview of related work on testing SPLs in general and in an industry context. Finally Section VI summarizes our experiences, highlights our conclusions and gives an outlook on our future work.

II. Industry Context & Initial Situation

The case company is an international machinery manufacturer building high-quality mechatronic systems. It is an international company with several development and production sites all over the globe, employing several thousand people. The local site has specialized on engineering and manufacturing a range of heavy-duty industrial machineries organized in terms of a product line. The company has a long and outstanding history in mechanical and electrical engineering, which has led to highly elaborated systems engineering and manufacturing processes.

Although software plays an important role in mechatronic systems such as industrial machinery, the whole domain is still dominated by expertise and innovations in mechanical and electrical engineering. The awareness for software engineering is slowly but steadily increasing, mainly due to the constantly increasing costs and effort involved in software development, software maintenance and software defects. Over the last decade the number of employees in departments concerned with software development has grown substantially while the size of other engineering teams remained more or less constant. This development increases the need to improve the integration of software engineering in the overall engineering and manufacturing processes. However, this context also explains the current situation of software engineering and, in particular, the current approach for managing software variability.

When we started working with the company, we found that the software development team had no documented variability model. Knowledge about their products' variability did exist in the collective memory of the team. The first step was to identify possible sources of information in the company. It took some time and effort to find out where to gather necessary information. Through discussions with the software development team we identified the sales department as a primary source of information, since that is where the lifetime of any product variant begins.

A. *Product Variant Management at System Level*

The sales department maintains a model for managing product variants, which we would call a variability model in software engineering. This model is comprised of a hierarchically structured set of system properties, each with a set of possible values and a related set of constraints defining the dependencies between the properties. The model is implemented in the company's Enterprise Resource Planning (ERP) system by SAP. The SAP business suite offers a dedicated module for variant management [5] that integrates and exchanges information with various ERP functions such as sales, production planning, material management, costing, and procurement.

The model is also the basis for the product configurator used by the sales personnel to negotiate and specify offers for customers. The model including the constraints ensures that only allowed combinations of features are selected and that the machinery can be produced from these specifications. Furthermore it maps high-level sales options to specific system parts and materials. For example, the sales option "destination country" triggers the selection of suitable voltage converters and power connectors. In the end, whenever a new product is sold, the ERP system automatically outputs the bill of materials and task lists for production planning and control.

Figure 1 illustrates the properties and the possible values of the configuration options in the ERP system, and it also depicts the constraints that exists for selecting different property values. The model contains 71 properties and 589 corresponding values. In the center the figure depicts a root node of the model, which combines all possible configurations and represents the starting point for configuring a system. From the center there are connections to the properties (shown as first, inner circle of nodes), and from these properties there are connections to their possible values (outer circle of nodes). We can observe that for most properties there exists a great number of possible values, which means that there are many configuration options one can select for these properties. Furthermore, the constraints are depicted in Figure 1 as the connections between the nodes representing the property values on the outer layer. Overall this figure should give a good understanding of the complexity of the system and the large number of configurations that are possible.

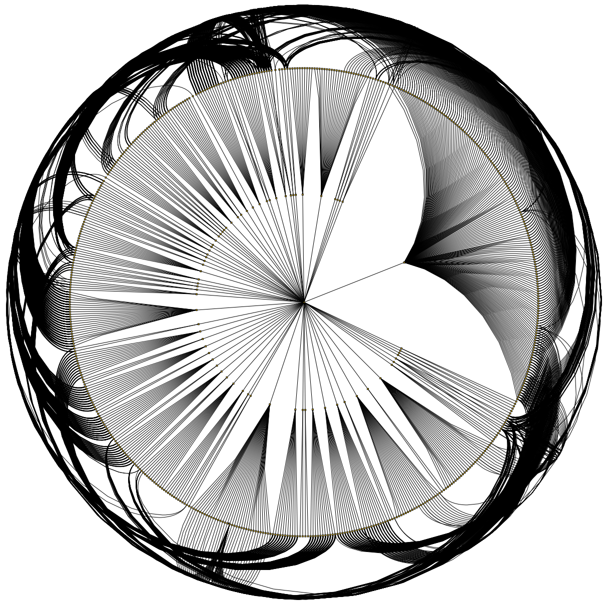


Fig. 1. Sales properties and property values with constraints.

At that point in the product line the software system is only considered as "yet another" system part that has to be integrated. The variant management considers the software system as a single, self-contained entity and does not distinguish between different software components or configuration settings.

B. *Software Architecture and Variability*

Of course, the software has to be configured according to the integrated hardware features and properties of the overall system. However, there existed no automatic support that translates the selected configuration based on the sales model into a configuration of the software system. In fact, the knowledge of how to configure the software to match the configuration of the machinery is part of the expert know-how of the responsible employees. The interaction between hardware and software can be highly complex, in particular since machines have to be calibrated and adjusted for specific production processes. Thus, software configuration tasks range from setting flags in configuration files to the optimization of process parameters stored in the software system.

The overall software system has two main sub-systems, the human machine interface (HMI) and the control software. The HMI provides a graphical representation of all parts and internal states of the system for machine operators and service engineers as well as an interface for programming the machine for the production process. The control software is a real-time software system for actually controlling the hardware of the machinery including custom hardware features and optional equipment ranging from laser sensors to loading robots.

The software system includes a central database for storing configuration parameters and user specific settings. This database contains several hundred parameters related to the configuration of various different software modules. A graphical configuration interface is provided for setting frequently used configuration options. However, since the configuration of the machinery usually goes beyond these options, the experts do not rely on the configuration interface but directly set the values in the database. Furthermore, although the database is the central point for storing configuration data, the various software modules may also maintain individual settings and configuration files. For example, software modules usually provide default values for configuration parameters in case these parameters are not present in the central database. The constant evolution of the system, i.e., the software system as well as the mechanical and electrical system, increases the complexity as new configuration options have to be added and others become obsolete but need to be maintained for reasons of backward compatibility. Finally, there are several more configuration files and settings from the underlying run-time system, the system libraries, hardware drivers and the operating system itself (e.g., installed service packs and supported languages) that affect the software configuration.

Currently the approach for persisting software configuration data is redesigned with the goal to replace the central database. Several members of the software development team (e.g., software architect, product manager, and developers of various software modules) are involved in specifying and implementing the new configuration mechanism.

C. Problems and Challanges

Currently the size and complexity of the software system is constantly increasing but software variability is still disconnected from the variant management of the overall mechatronic system. In consequence, software development faces several problems and challenges.

- *Department specific understanding of variability:* The model capturing the variability of the overall system is mainly used by the sales department and not made available for the members of the software development teams. Thus, the understanding of what is meant by variability is different in the sales process, in mechanical and electrical engineering, and in software development. This goes to the point where the software team still develops and maintains features that are not part of a valid, sellable product configuration anymore.
- *No automatic software configuration:* The bill of materials used in manufacturing is generated as part of the product configuration process. However, there is no equivalent for configuring the software system. The configuration of the software system is completely informal. The knowledge of how to configure the software to correctly interact with a certain hardware configuration is tacit knowledge possessed by a small group of employees.
- *Lack of communication over organizational boundaries:* Exchanging information across the boundaries of the different organizational units (i.e., sales, mechanical engineering, electrical engineering, software engineering) is far from trivial due to unrelated perspectives, tasks and responsibilities. Not even the terminology used in the model of the sales team is consistent with terminology used during the configuration of the software system.
- *Models from different domain perspectives:* The mapping between the different models used in sales, mechatronics and software is not trivial. For example, not every property in the sales model has an impact on the hardware or the software, e.g., the packaging requirements to protect the machine when it is shipped to the customer site. Similarly, not every property that affects the hardware necessarily has an impact on the software, e.g., the color selected for painting the machine. And we also found that there are sales properties that affect the software only but not the hardware, e.g., interfaces to production planning and control systems. The problem is further intensified by the many different settings and configuration methods spread out across the software system. In some cases the same configuration can be achieved by setting different parameters or making changes to different property files.
- *Extremely large number of configurations:* The impact the configuration problem has on software testing and debugging is enormous [6]. Considering only the main options listed in the configuration interface for the central database, we calculated that there are more than 110 million possible configurations. In any case it is estimated that there are several hundred different configurations of the software product in the field. There is no distinct general or standard configuration. More than hundred of these configurations have been frequently applied to numerous shipped machines, thus, each can therefore be considered as a ”standard” configuration. Without taking the configuration constraints from the sales model into account, a regression test would take several weeks, even if it is fully automated [7].

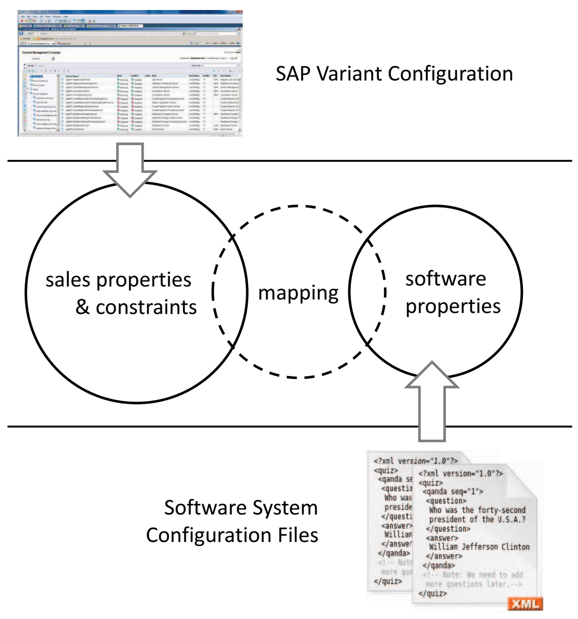
Naturally, all these issues are a problem because of the important (and still rising) role of the software. A significant portion of the product's functionality is already implemented in software. However, traditional manufacturing companies often underestimate the consequences and risk that software development as well as software variability is getting out of sync with the variant management used in sales and manufacturing.

III. Variability Model Integration

This section describes our approach for bridging the gap between the product variant management used by the sales department and the software configuration options implemented in the software system.

A. Overview of the Approach

Figure 2 illustrates the integration approach. It is based on a distinct model layer introduced between the high-level model maintained in the ERP system and the low-level model represented by the software configuration options. Since there is no leading system that can be appointed as "reference", the layer helps to synchronize the different abstraction levels of the models (i.e., top-down view of variability in product management and configurations emerging bottom-up in software development) as well as their evolution (i.e., product version versus software releases). The mapping itself is maintained in a separate model that can also be evolved individually. The integration is supported by a set of tools that facilitate and implement the illustrated approach.


SAP Variant Configuration
sales properties & constraints
mapping
software properties
Software System Configuration Files
XML

```
<?xml version="1.0"?>
<quiz>
 <qanda seq="1">
  <question>
   Who was the forty-second
   president of the U.S.A.?
  </question>
  <answer>
   William Jefferson Clinton
  </answer>
 </qanda>
 <!-- Note: We need to add
  more questions later.-->
</quiz>
```

Fig. 2. Mapping of sales properties and software configuration options.

B. Data Extraction and Mapping

In a first step we analyzed the model maintained in the ERP system. The SAP variant configuration [5] provides a comprehensive modeling and configuration environment including programming with logical expressions for specifying preconditions, selection criteria, etc. By parsing the results from SAP's export mechanisms we were able to extract following data:

- The set of properties (i.e. features) and the values each property can take, e.g., property *Country* can take the values *France*, *Spain* etc.
- The set of constraints on these properties, e.g., $(Country = UK \vee Country = US \vee ...) => Language = English.$
- A set of default values for the properties, e.g., the default value for property *Language* is *English* if no value is selected during product configuration.

We received data on software configuration options from the software development team. Furthermore we extracted configuration options from the user interface of the configuration database. This also resulted in a set of properties with possible values but without any constraints. In fact the software system currently allows any combination of configuration values to be set without checking for validity. In case of a serious misconfiguration the system would not be able to recover without manual intervention.

In total, the number of identified software properties used in software configurations was significantly smaller than the sales features extracted from the ERP system (71 properties with in sum 589 possible property values in the ERP system versus 16 properties and 60 values for software configurations). Nevertheless, the software is highly configurable. But it is not obvious and also not documented what machine configuration requires what software configuration. Some configuration terms in the software match sales and machine feature names, others do not match by name but might still have an equivalent sales or machine feature, and some are connected through more complex constraints. We related the sales and software properties by a set of manually created constraints. These constraints were derived from similarities in the names of certain properties or their values or via discussions with employees from the sales and software departments.

C. Implementation

As mentioned before, the software configuration options are expressed differently than the sales properties. Therefore the mapping was not a simple 1:1 mapping, but some sales properties were expressed as multiple properties in the software and vice versa. We implemented this mapping through a set of constraints linking the properties to one another, inspired by a syntax introduced by Microsoft in their PICT tool for pairwise testing [8]. The syntax allows expressing the model in a human readable textual format. Constraints can be specified using a simple yet flexible constraint language. The extracted sales properties and software configuration options were also both modeled in this syntax.

We ended up with three models (represented by the circles in Figure 2): one model with the sales properties and constraints, one for the mapping constraints, and one model containing the software properties. The three models were stored in separate files. This separation supports modularization as well as an independent evolution path and version control for

Sales Properties:

```
1    MachineNumber: "M1", "M2"
2    Size: "M", "X"
3    Protection: "YES", "NO"
4    Simulate: "Y", "N"
5
6    IF [Size] = "X" THEN [Protection] = "YES";
7    IF [Protection] = "YES" THEN [Simulate] = "Y";
```

Software Properties:

```
8    MachineType: "M1", "M1X", "M2", "M2X"
9    Simulation: "YES", "NO"
```

Mapping:

```
10   [MachineType]."M1" = [MachineNumber]."M1" AND
         [Size]."M";
11   [MachineType]."M1X" = [MachineNumber]."M1" AND
         [Size]."X";
12   [MachineType]."M2" = [MachineNumber]."M2" AND
         [Size]."M";
13   [MachineType]."M2X" = [MachineNumber]."M2" AND
         [Size]."X";
14   [Simulation]."YES" = [Simulate]."Y";
15   [Simulation]."NO" = [Simulate]."N";
```

Fig. 3. Example Properties

each individual model. For example, once additional information is provided by the development team, the next version of the model representing the software properties will also contain constraints for further restricting the selection of these properties.

Figure 3 shows a small example with a few selected properties, in the used syntax. The lines from line 1 to line 4 represent the sales properties. Lines 6 and 7 are the constraints for the sales properties. They are formulated in an if-then-else manner, for instance, if `Size` is `X` then `Protection` has to be `YES` as can be seen in line 7. Next, line 8 and line 9 represent the properties as used in the software configuration. Finally line 10 to line 15 are the mapping between the software properties and the sales properties.

Line 10 to line 13 in Figure 3 represent the mapping between `MachineType` in the software properties and properties `MachineNumber` and `Size` in the sales properties. For example line 11 means, if `MachineType` is `M1X` then the sales properties `MachineNumber` and `Size` must be set to `M1` and `X` respectively. These constraints go both ways, so also if `MachineNumber` is `M1` and `Size` is `X`, then `MachineType` has to be `M1X` for the constraint to be satisfied.

The implementation resulted in a suite of small tools and utilities that provide support for following tasks. The application of these tools is described in more detail in the next section.

- Parsing and analyzing the model files containing properties, possible values and constraints formulated on these properties.
- Checking the validity of a specified configuration (i.e., a product variant within the product line) and showing violated constraints in case of a misconfiguration.
- Generating a (partial) software configuration from a given sales configuration (similar to the bill of materials in manufacturing).
- Computing of a minimal set of combinations of software configuration properties for software testing that satisfy t-wise coverage criteria.

IV. Results and Applications

This section discusses the results of our work in terms of the effect for the case company and it provides further details about the application of the tools we developed and the lessons we learned.

A. Awareness for Software Variability Issues

A very important result of our work for our industrial partner was that it did raise awareness about internal problems in the overall engineering process. The problems were mainly caused by a lack of information exchange between the different organizational units within the company. For instance software and hardware development had a very different understanding of the product features that resulted in a mismatch of what has to be maintained. In the collaboration with representatives from the sales department and from software engineering we found out that the software development team still maintained features that had been removed from the sales configuration and no machines were produced that would require these features.

The software department had a notion of features, which however were mainly a subset of the sales features. Furthermore the software features had some different characteristics than the sales features, which makes a mapping and information exchange difficult and prone to inconsistencies.

In general we observed that the hardware side, i.e., the mechanical and electrical engineering products, received more attention and had a higher priority than the software side. Sadly this is often the case in industry, partly due to historical reasons and partly due to the ability to change the software more easily in late stages of the engineering process. Even if this additional flexibility is considered an advantage of software, late changes in the software development also cause additional costs, increase the risk of introducing critical defects, and require additional communication and synchronization effort in the concurrent engineering of hardware and software.

B. t-Wise Covering Arrays

An important goal was to provide support for software configuration testing. Testing all possible configurations is usually impossible; in our case this would mean testing more than 110 million configurations. To reduce the number of combinations we developed tool support for combinatorial interaction testing that generates a t-wise covering array on the software features. The tool has to take the constraints into account in order to make sure that only valid combinations are generated for testing.

With the support of the tool we generated 1.232 valid software configurations satisfying 3-way interactions (*t=3*) between the configuration parameters. The initial exorbitant amount of possible combinations was reduced to a small (0.001%) but significant number of configurations relevant for testing. By including additional constraints as suggested by the development team, this number can be reduced even further.

Definition 1: A t-set ts is a 2-tuple [sel,$\overline{sel}$] representing a partially configured product, defining the selection of t properties, i.e. $ts.sel \cap ts.\overline{sel} = \emptyset \wedge |ts.sel \cup ts.\overline{sel}| = t$.

For generating the t-wise covering arrays our tool first selects t-wise combinations (i.e. *t-sets* see Definition 1) based on the software properties. Definition 1 is usually used for *t-sets* for the selection of binary features (i.e. features that are either selected, in sel or not selected, in $\overline{sel}$). To use *t-sets* for our purpose we use sets of properties instead. Meaning the set sel represents properties that have to be assigned with one of its predefined values and set $\overline{sel}$ represents properties that do not get a value assigned to them (i.e., are not selected in the current configuration). Therefore our *t-sets* cover all possible t-wise combinations.

For each of the properties in our *t-sets* that get a value assign (i.e. properties in sel), the algorithm checks all possible predefined values for them and subsequently tries to satisfy the constraints in the mapping file, by selecting the sales property-values that are mapped the selected software property-values. This allows us to make use of the constraints in the sales model. Subsequently the tool tries to satisfy all these constraints in the sales model, by selecting the appropriate property-values. If there are constraints left that cannot be satisfied, because of another software property that was already set to a value, the configuration is invalid and will not be considered in the covering array. Otherwise, if the selection is valid, i.e. no constraints are violated, the mapping is utilized once more to check if there are software property-values that have to be selected now, due to the selection of sales property-values while satisfying constraints. If this was successful, and there are no constraints violated in the software properties, the configuration will be considered in generating the covering array.

To calculate the covering array we used an algorithm similar to the one used by Johansen et al. in [9]. The first step is to exclude all the *t-sets* that are not satisfiable, i.e., combinations of property-values that already violate at least one constraint and can not be made valid by selecting any other properties that do not have a value yet, as described above. This leaves us with a list of *t-sets* that have to be covered by the resulting covering array. Next the algorithm merges these *t-sets* into valid configurations. By this merging we get products that cover more, best case as many as possible, t-wise combinations. The algorithm iterates over the *t-sets* and adds them to a current configuration. For each *t-set* it checks if the configuration is still satisfiable with the *t-set* added. If so the *t-set* is added to the configuration, otherwise it is skipped for the current configuration. After all the *t-sets* have been checked, for the current configuration, the already added *t-sets* are removed from the list and the configuration is made valid by selecting property-values that still may violate satisfiable constraints. The algorithm repeats this process, with new current configurations and the remaining *t-sets*, until the list of *t-sets* is empty. The configurations that have been merged in this way represent the resulting covering array.

The advantage of this approach is that it can consider transitive dependencies. For instance property `MachineType` is set to `M1X` in the software properties in Figure 3. Because of the mapping in line 11, this means property `Size` in the sales properties is set to `X` then property `Protection` has to be `YES`, due to the constraint in line 6. From this and the constraint in line 7 property `Simulate` has to be set to `Y`. For the software properties this then means property `Simulation` has to be `YES`, due to the mapping constrain in line 14. By using all the sales properties we can use these constraints directly and therefore get the behavior as if we had the constraint if `MachineType` is `M1X` then `Simulation` has to be `YES` directly in the software properties.

Table I shows the combinations generated by the tool for the example software properties and the mapped constraints in Figure 3.

```
MachineType = M1, Simulation = YES
MachineType = M1, Simulation = NO
MachineType = M1X, Simulation = YES
MachineType = M2, Simulation = YES
MachineType = M2, Simulation = NO
MachineType = M2X, Simulation = YES
```

TABLE I. GENERATED COMBINATIONS

C. Adding Constraints to the Configuration Interface

Another application leveraged the ability of the tool support to check if a given software configuration satisfies basic constraints such as those from the sales model. The company showed interest in integrating our tool as an add-on for the configuration interface of the central configuration database. The interface allowed selecting values for a set of frequently used configuration properties. Yet the interface did not provide any means for checking configuration constraints so far.

To add constraint checking to the interface they decided to utilize the implementation provided by our tool. The idea was not to restrict the user in making specific selections of property values but to issue a warning in case of a potential misconfiguration. Potential misconfigurations are detected by checking if a configuration is valid, which works exactly as described above, based on the constraints specified in the sales model. If a configuration is invalid, i.e., at least one constraint is violated, the tool will report an error. However, the user is still able to proceed even though the last selection resulted in an invalid configuration, which of course can also be understood as a partially valid configuration [10]. Thus the tool points the user to the constraints that have been violated, so that the user can change the selection and repair the configuration. Suggestions for repairing are important since the constraint model is highly-complex and different repairs may be possible to get back into a valid state.

D. Lessons Learned

One of the most important observations we made in our work was that software variability can be influenced by very different organizational units. In the case of our industrial partner, the sales and the software departments as well as the mechanical and electrical engineering departments influence what configurations of the software system are possible and valid. All these departments can introduce constraints which make existing configurations in the software pointless, because their exists no machinery that can be built in that way. Excluding invalid configurations from testing is important since they would lead to numerous hours of wasted testing time (per configuration) and a possibly large amount of false positive test results requiring further manual analysis. Therefore all the constraints that can influence the software in any way should be considered in an attempt to reduce the test effort.

We observed that the mapping between the models is critical for reducing the test effort. Therefore we performed an experiment where we used only mappings from one software property at a time, to see how the constraints for these properties affect the resulting number of configurations that would have to be tested. For the experiment we use 3-way interactions as the criteria for configuration coverage in testing. When including all our manually defined mapping constraints, our algorithm computes 1.232 configurations that would have to be tested for 3-way coverage. Without these mapping constraints (i.e., only unconstrained software properties) our tool computes 1.579 configurations, which includes 347 configurations that are invalid according to the sales model.

Group of Constraints	Reduction 3-way
A	2
B	2
C	162
D	214
E	2
F	0
G	2
H	0
I	0

TABLE II. EFFECT OF DIFFERENT CONSTRAINT GROUPS ON THE NUMBER OF CONFIGURATIONS TO TEST.

Table II shows how groups of mapping constraints for different software properties affect the resulting number of configurations. In particular the table illustrates the reduction of configurations calculated with 3-way covering arrays that we achieve when using only a specific group of mapping constraints and excluding all other mapping constraints. We can observe that the groups C and D have the biggest impact for reducing the number of configurations. All other constraints have very little or no impact, because these properties are not much or not at all constrained in the ERP system either. Our observations leads us to believe that we can improve the results further if we refine the data in our future work.

As a final lesson learned we recognized that it is extremely important for different organizational units to work together by following a defined process that supports information exchange and communication between each other. Otherwise it can happen that in developing and evolving a complex system, new features are developed incorrectly or maintained unnecessarily. The exchange of information regarding variability and configurations can be implemented for example in form of a company wide understood model that integrates the different views and needs of all involved organizational units.

V. Related Work

Johansen et al. [3] developed an optimized approach for generating covering arrays. As basis for their optimizations they used an algorithm similar to the one used in this paper for generating the covering array. It would be interesting if their optimization could also be used in our scenario. However the difference to our work is, that in our scenario we did not have a feature model but rather properties that can take on predefined values. Furthermore Haslinger et al. [4] use feature model knowledge to reduce the number of t-wise combinations in the calculation of covering arrays. In our work we used the knowledge that a property can only take on one value at a time as the base of generating the covering array. The difference is that we do not have binary features, which are either selected or not selected, but predefined values for each property. Moreover, the main contribution of our work was preparation of data from different models to be able to apply a combinatorial interaction approach on the company specific variability models.

A comprehensive overview of combinatorial interaction testing for software product lines can be found in [11]. Examples for applying combinatorial interaction testing to software product lines from industry are the works of Marijan et al. [12] and Steffens et al. [13] who evaluate their approaches on industrial case studies. The difference to our work is that they rely on feature models with binary features, and they also focus less on the challenges involved in industry applications. However the evaluations in these publications could be useful for our future work and might give us results to compare against when moving forward in the collaboration with our industry partner.

Wang et al. [14] describe their research on applying combinatorial interaction testing to an industrial SPL and also illustrate the companies initial problems and how they improved their testing. However, their work is concerned with prioritization of test cases and, again, based on a feature model with binary features. Nonetheless they also observed similar problems in the processes and workflows of their industry partners. For instance they found many tasks that had to be performed manually and for which there was no model or documentation available. The knowledge about performing these tasks existed solely in the heads of some employees. The work of Wang et al. confirms that the situation we experienced at our industry partner is not an isolated case and that the proposed integration approach may be useful to a wider audience.

VI. Conclusions and Future Work

In this paper we presented the experiences we gained from building a variability model for the highly configurable software part of a complex mechatronic system produced by one of our industry partners. The goal of the collaboration has been to support the company in establishing a variability model for the software system and to optimize testing of the software configurations. In the process of getting there, we discovered some shortcomings in overall engineering process and a loss of information across the boundaries of organizational units. We were able to highlight these problems and to raise the awareness for software variability issues.

During our work together with company representatives we found out that there is already a model that expresses the product variability at system level in form of properties and constraints. This model is used by the sales department to configure products and by manufacturing for production planning and control.

From this model we extracted the information about product variants and constraints. We mapped this information to the software variability model derived form software configuration options. The combined model was finally used as basis for computing test configurations with t-wise interaction coverage. Throughout our work we provided the company with a set of tools that can be used to *i)* compute t-wise coverage, *ii)* verify the validity of a given software configuration, and if it is found to be invalid, *iii)* show which constraints are violated and suggest appropriate repairs.

With our work we bridged the gap between product variant management and software configurations. We were able to make information from the high-level variant management available to the software development team. The provided information is used to apply advanced testing techniques like, e.g., combinatorial interaction testing. Furthermore, the tool support for checking the validity of a software configuration is considered for integration in the configuration interface of the software system.

This paper describes the first results we were able to achieve in our collaboration with the industry partner. The success and the positive feedback encourage us to continue our work. Our plans for future work are manifold.

First of all we plan to continue in the direction of integrating the different variability models by further refining the mapping between product variants and software configurations. Currently the mapping uses only properties and values that are directly associated. By digging deeper into the knowledge of the experts, we want to extract and implement the rules that are used to resolve complex, multi-level dependencies in the software configuration process. In the end we want to provide the ability to automatically generate a full software configuration from the high-level product configuration, similar to the automated generation of the bill of materials from the sales configuration in the ERP system. By automating the process the software configuration of new machinery could be speed up significantly.

The effort required for configuration testing is still a major issue for the company. Thus, we plan to use the default values of the properties stored in the sales model to determine which product variants are more likely and to prioritize the configurations for testing according to this likelihood. Another possibility would be to include the sales data about which configurations were sold in the past in order to determine the most used property values for deriving a prioritization schema.

In the long term it will be necessary to establish a company-wide, integrated variability model and configuration process that is used and understood across all departments. To achieve this we plan to start with the development of a company-wide knowledge base that can be viewed and maintained by all departments. It should integrate the heterogeneous models from sales, hardware, software etc., map the commonly used feature terms, and contain a database for recording all configurations in the field.

ACKNOWLEDGMENT

The work described in this paper was supported by the COMET program of the Austrian Research Promotion Agency (FFG). Furthermore we want to thank our industry partner for the cooperation and support, and we gratefully acknowledge the permission to report our experience.

REFERENCES

[1] F. van der Linden, K. Schmid, and E. Rommes, *Software product lines in action - the best industrial practice in product line engineering*. Springer, 2007.

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

[3] M. F. Johansen, Ø. Haugen, and F. Fleurey, “An algorithm for generating t-wise covering arrays from large feature models,” in *16th International Software Product Line Conference, SPLC ’12, Salvador, Brazil - September 2-7, 2012, Volume 1*, E. S. de Almeida, C. Schwanninger, and D. Benavides, Eds. ACM, 2012, pp. 46–55.

[4] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed, “Using feature model knowledge to speed up the generation of covering arrays,” in *The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS ’13, Pisa , Italy, January 23 - 25, 2013*, 2013, p. 16.

[5] U. Blumohr, M. Munch, and M. Ukalovic, *Variant Configuration with SAP*, 2nd ed. SAP PRESS, 2011.

[6] D. Jin, X. Qu, M. B. Cohen, and B. Robinson, “Configurations everywhere: Implications for testing and debugging in practice,” in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 215–224.

[7] R. Ramler and W. Putschogl, “Reusing automated regression tests for multiple variants of a software product line,” in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 122–123.

[8] J. Czerwonka, “Pairwise testing in the real world: Practical extensions to test-case scenarios,” 2008. [Online]. Available: https://msdn.microsoft.com/en-us/library/cc150619.aspx

[9] M. F. Johansen, Ø. Haugen, and F. Fleurey, “Properties of realistic feature models make combinatorial testing of product lines feasible,” in *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, 2011, pp. 638–652.

[10] A. Nöhrer, A. Biere, and A. Egyed, “A comparison of strategies for tolerating inconsistencies during decision-making,” in *16th International Software Product Line Conference, SPLC ’12, Salvador, Brazil - September 2-7, 2012, Volume 1*, 2012, pp. 11–20.

[11] R. E. Lopez-Herrejon, S. Fischer, R. Ramler, and A. Egyed, “A first systematic mapping study on combinatorial interaction testing for software product lines,” in *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops Proceedings, April 13-17, 2015, Graz, Austria*, 2015, pp. 1–10.

[12] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu, “Practical pairwise testing for software product lines,” in *17th International Software Product Line Conference, SPLC 2013, Tokyo, Japan - August 26 - 30, 2013*, 2013, pp. 227–235.

[13] M. Steffens, S. Oster, M. Lochau, and T. Fogdal, “Industrial evaluation of pairwise SPL testing with moso-polite,” in *Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings*, 2012, pp. 55–62.

[14] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, and M. Liaaen, “Multi-objective test prioritization in software product line testing: an industrial case study,” in *18th International Software Product Line Conference, SPLC ’14, Florence, Italy, September 15-19, 2014*, 2014, pp. 32–41.