Light-Weight Tool Support for Staged Product Derivation

Christoph Elsner
Siemens Corporate Research and Technologies, Erlangen, Germany
christoph.elsner@siemens.com

ABSTRACT

Tool support that checks for configuration errors and generates product parts from configurations can significantly improve on product derivation in product line engineering. Up to now, however, derivation tools commonly disregard the staged derivation process. They do not restrict configuration consistency checks to process entities such as configuration stages, stakeholders, or build tasks. As a result, constraints that are only valid for certain process entities must either be checked permanently, leading to false positive errors, or one must refrain from defining them at all.

This paper contributes a light-weight approach to provide tailored tool support for staged product derivation. Compared to previous approaches, it is not tied to a single configuration mechanism (e.g., feature modeling), and also accounts for the stakeholders involved and the build tasks that generate product parts. First, the product line engineer describes the derivation process in a concise model. Then, based on constraint checks on the configuration (e.g., a feature model configuration) that are linked to the modeled entities, comprehensive tool support can be provided: Configuration actions can be guided and restricted depending on the configuring stakeholder in a fine-grained manner, and constraints attached to a build task will only be checked if it actually shall be executed. Finally, in combination with previous work, the paper provides evidence that the approach is applicable to legacy product lines in a light-weight manner and that it technically scales to thousands of constraint checks.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Software Architectures—*Reusable Software*

General Terms

Design, Languages

Keywords

Product Line, Staged Product Derivation, Tool Support

SPLC '12, September 02 - 07, 2012, Salvador, Brazil

1. INTRODUCTION

Software product line engineering aims at reducing software development cost by systematic reuse from early on when developing similar products. One promising product line technique is the automation of application engineering, also often called automated product derivation. Its core is the automated mapping of a declarative configuration to a specifically tailored product using product generators, such as preprocessors or code generator templates. Two facets of automated tool support need to be considered: automated *configuration* support, such as configuration consistency checks and automated fixes, and automated *generation* of products from the configuration, via such generative technologies.

Despite the great potential of automation in product derivation for reducing development cost, reports point out the immense manual effort and difficulty of industrial product derivation [6, 9, 21, 24]. Why is this the case? Figure 1 illustrates the derivation process of the software product line of a magnetic resonance tomograph. Its *staged product derivation process*, which involves multiple different, heterogeneous entities, considerably complicates the provision of comprehensive automated support.

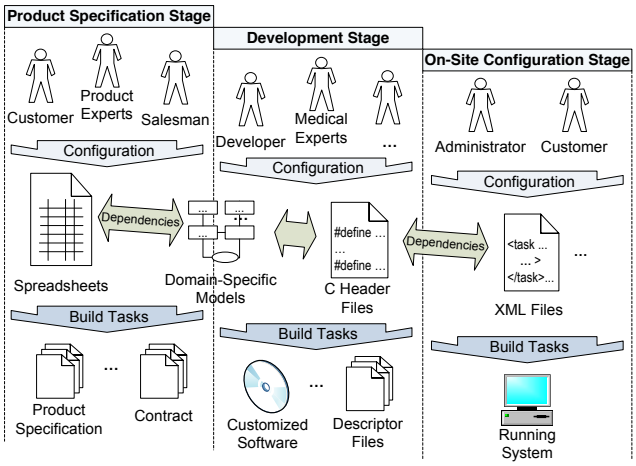

Figure 1: Staged product derivation from a magnetic resonance tomography product line.

The derivation process in Figure 1 has three stages, product specification, development, and on-site configuration stage. In each of them, different stakeholders adapt configuration artifacts such as spreadsheets, domain-specific models, or XML

files. Build tasks take them as input and produce product artifacts, ranging from textual documents to software.

Basic automated support for staged derivation should at least describe *which* of the configuration options needs to be set at *which* stage. However, there is further automation potential that has been vastly overlooked up to now:

a) Stakeholder Guidance. Each stage involves different stakeholders, such as salesmen, product experts, or administrators. Each of them needs individual process guidance. The salesman, for example, has less experience than the product expert; he or she requires more configuration hints and guidance to assess the side effects of configuration decisions. Some crucial decision might even require an expert's knowledge—and a salesman should be actively hindered from taking them.

b) Build Task Integration. In each stage, build tasks need to be invoked to generate product parts, for example, document or code generators, preprocessors, or installation scripts. Stakeholders would greatly benefit from their explicit integration into the derivation process. They need to know which build tasks are mandatory or optional at which stage. They need to manage the execution dependencies among build tasks, which possibly are based on different build tools (e.g., make, ant, or binary executables). Finally, each build task requires distinct configuration options to be set—tailored support should inform a stakeholder as soon as the execution prerequisites for a particular build task are fulfilled.

Additionally, a staged derivation approach should not be limited to a certain configuration mechanism, such as feature modeling, and should be applicable in a light-weight manner.

To address these issues, this paper presents a light-weight approach for staged derivation tool support. Compared to previous approaches (cf. Section 8), it can provide guidance to individual derivation stakeholders and ensures correctly-configured execution of build tasks in each stage. Furthermore the approach is not tied to a certain configuration mechanism. After a motivation for more comprehensive tool support via three concrete challenges (Section 2), this paper contributes the following:

1. A light-weight approach for staged derivation tool support. It can be used with any product line configuration mechanisms that support accessing the configuration model from a separate language in which constraints (necessary conditions for a consistent product) can be defined. In the simplest case, this might be plain Java. (Section 3)

2. A concise staged derivation modeling language. The language comprises the expressiveness of most existing approaches while it facilitates the tool support. (Section 4)

3. Comprehensive tool support based on constraint checks. The developed tooling integrates with the development environment used for configuration, for example, Eclipse. Constraint checks attached to the modeled elements serve as input to the tooling to provide staged derivation support including stakeholder guidance and build task integration, as motivated above in a) and b). (Section 5)

In combination with previous work on extraction of configuration data from legacy product lines [10, 12], Section 6 can provide evidence that the approach is applicable to legacy product lines in a light-weight manner and that it technically scales to thousands of constraints checks. Finally, Section 7 discusses the approach, Section 8 addresses related work, and Section 9 concludes the paper.

2. MOTIVATION AND EXAMPLES

This section introduces the staged derivation process of the product line demonstrator SmartHome, on which more details can be disclosed. Based on this product line, three concrete staging challenges are illustrated, which will be addressed in the remainder of this paper.

2.1 SmartHome Staged Derivation Process

SmartHome [29] is a product line demonstrator which aims at construction experts such as architects and interior designers. It enables them to model a building and its electrical interior devices. The models serve as input to generate the software for controlling the building's devices.

In order to illustrate the challenges and solutions for staged derivation, a staged derivation process was devised for the SmartHome demonstrator. The first two stages resemble the industrial experience of the author [25]; the latter two derivation stages follow the characteristics of the house automation software and its derivation mechanisms (cf. Figure 2).

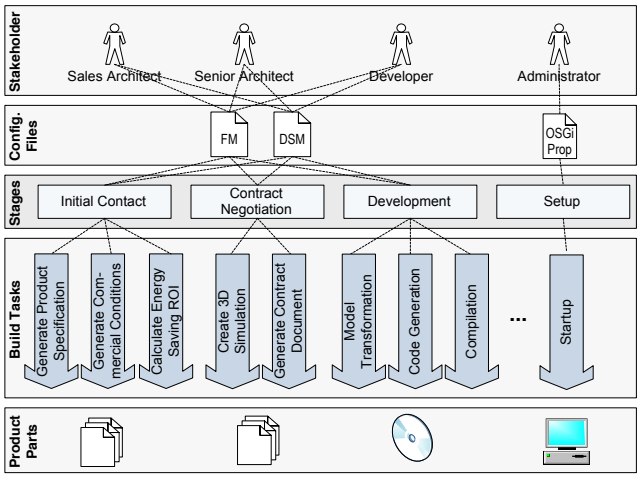

Figure 2: SmartHome's staged derivation process.

The derivation process comprises four configuration stages: initial contact, contract negotiation, development, and setup. In each stage, one or several STAKEHOLDERS, adapt *configuration files* in order to execute build tasks that produce product parts. The four stages have the following characteristics:

1. Initial Contact. During initial contact, the SALES ARCHITECT, or, in more complex cases, the SENIOR ARCHITECT, consults the customer and adapts the *feature model configuration* according to his or her wishes. Three build tasks—generate product specification, generate commercial conditions, and calculate energy saving return-on-invest (ROI)—take the feature model configuration as input to create product-related documents. Each build task requires a distinct set of features to be set for proper execution.

2. Contract Negotiation. During contract negotiation, the SALES or the SENIOR ARCHITECT refine the *feature model configuration* and create the *domain-specific house model* according to customer data. Exemplary build tasks are create 3D simulation and generate contract document.

3. Development. During development, the DEVELOPER refines the *feature model configuration* and the *domain-specific house model*. Various build tasks related to development

(model transformations, code generation, compilation, testing) need to be executed.

4. Setup. In the final setup stage, an ADMINISTRATOR installs the devices at customer site, sets up the *OSGi property file*, and triggers the startup of the SmartHome software.

Based on this staged derivation process, three challenges for comprehensive tool support will be illustrated: basic support, build task integration, and stakeholder guidance.

2.2 Challenge 1: Basic Support

Basic staged derivation support must check, during each stage, whether all data has been gathered and is consistent to proceed to the next stage. As an example, setting the customer name attribute (*CustomerName*, constraint 1 in Figure 3) is required for completing the initial contact stage.

2.3 Challenge 2: Build Task Integration

Additionally, build tasks should be integrated into the staged derivation process. The detailed characteristics of SmartHome's *InitialContact* stage are depicted in Figure 3. The configuration stage is considered completed when the two mandatory build tasks *generateProductSpecification* and *generateCommercialConditions* have been executed. In contrast, the build task *calculateEnergySavingROI* is optional; it does not need to be invoked for completing the stage.

The mentioned build tasks take the SmartHome feature model configuration as input. However, in order to execute properly, each build task has different expectations for the features and attributes to be set.

For example, let us assume the customer wants a calculation of the ROI of the energy saving feature. For this build task, the number of inhabitants feature attribute (*NoInhabitants*, constraint 2 in Figure 3) must be set. If, in turn, a customer does not want this calculation, no error message should be shown if the attribute is not set. So, the configuration is not incomplete in an absolute sense—the consistency of the configuration depends on a *build-task–specific configuration constraint*.

A further example for build-task–specific constraints can be found during development. If the developer only wants to generate source code for analysis (e.g., worst-case execution time), the compilation or packaging build tasks do not need to be properly configured yet. As well, build tasks such as component, sandbox, or system tests may not necessarily need to be correctly configured to proceed to the next stage (although their execution may be advisable).

Finally, in Figure 3, build tasks are based on various generation mechanisms, such as ant, make, and binary executables, whereas build tasks may have dependencies among one another. For providing *product generation support*, means are necessary to execute such build tasks and manage their dependencies, even if they are not based on the same generation mechanism. Note that the implementation of such generators is out of scope of this paper: The individual build tasks are considered as black boxes, which however can have dependencies among another and which have certain requirements on the configuration in order to execute successfully.

2.4 Challenge 3: Stakeholder Guidance

The involved stakeholders should receive detailed information, warnings, and should even be actively hindered from performing certain actions, depending on their stakeholder role in each stage. Commonly, the individual stakeholders have differing permissions and restrictions in each stage and they need a different extent and quality of process guidance.

Let us consider the three stakeholders roles *SalesArchitect*, *SeniorArchitect*, and *Developer*. Whereas the sales architect markets SmartHome products based on a limited amount of training, the senior architect also has an understanding of product internals. Therefore, the former needs more process guidance and more restrictions than the latter. The Developer, in turn, is only involved at later stages, during creation of the actual product software.

Figure 4 shows an excerpt of the permissions, restrictions, and the required process guidance of the three roles in the first three stages of SmartHome's derivation process. In the initial contact stage, both the sales and the senior architect should focus on editing the feature model. Editing the domain-specific house model in this stage is discouraged. The sales architect should receive derivation process guidance and should be notified to execute the ROI calculation build task if he or she selects the energy saving feature. The senior architect, in contrast, does not need such detailed derivation hints and would conceive them as disrupting. The developer, finally, is not involved in the initial contact stage.

In the contract negotiation stage, sales and senior architect may edit both the feature model configuration and the domain-specific model. However, the sales architect may not edit any features or model elements regarding burglar alarm and alarm devices—due to their relevance to security. Again, the developer is not supposed to edit any configuration file.

Finally, after entering the development stage, the handover of configuration files to the developer is completed, who is now exclusively in charge of adapting them.

3. APPROACH OVERVIEW

The previous section has motivated three major challenges of staged derivation tool support: basic support, build task integration, and stakeholder guidance. This section outlines an approach that can provide such tool support in a lightweight manner. The approach is not limited to a certain configuration mechanism (e.g., feature modeling)—its only requirement is that an external language has access to the configuration data in order to define constraint checks on it.

In the simplest case, the constraint language can be plain Java and access can be realized via the Java API of the configuration mechanism used. In previous work [12], we have developed an approach that uses automatic converters to map a variety of widespread configuration mechanisms[1] used by product lines to metamodels and the corresponding configuration files of the concrete products to respective models. Therefore, in the following, without loss of genericity, each product line will provide a set of *configuration metamodels*. On product level, in turn, configuration takes place via *configuration models*, which are instances of these metamodels.

Figure 5 gives an overview of the approach, which involves three types of roles: *product line engineer*, *product engineer*, and *derivation stakeholders*.

In the beginning, the *product line engineer* creates a *StageModel* for the product line (Figure 5, step 1). The StageModel

[1]In particular, pure::variants feature models, domain-specific models based on Ecore, domain-specific textual languages based on Xtext, XMLSchema XML, Java property files, Kconfig files, and C header files with #defines (cf. [12]).

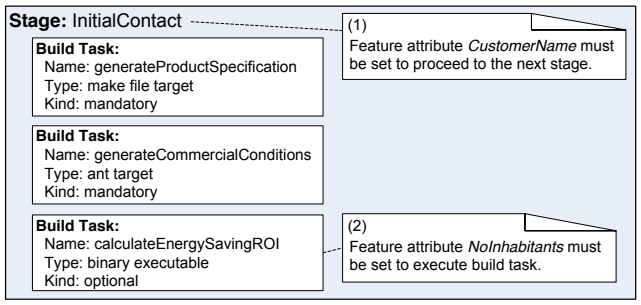

Figure 3: Build tasks during stage *InitialContact*.

defines the available configuration stages and the involved stakeholder roles of the product line. Furthermore, for each stage, the model describes its main build tasks implemented in certain build files (e.g., generate offer document, make test, or execute installer).

Constraints checks, which are implemented in separate artifacts, can be assigned to those stages, build tasks, or stakeholder roles in the StageModel for which they are valid. Hereby, the implementation of each constraint will usually refer to the configuration metamodel, as this *defines* the configuration variables (e.g., features, decisions, or attributes) which can be set to certain values in the configuration model.

On product level, the *product engineer* creates a StageInstance model that basically only assigns the individual derivation stakeholders to the stakeholder roles defined in the StageModel (step 2). Based on this input, the *derivation stakeholders* start configuring the product according to the staged configuration process defined in the StageModel (step 3). The *staged derivation tool* checks the defined constraints according to the current stage and the prospected build task and, if the constraint language also supports fix actions (e.g., [10]), may also allow for automated configuration fixing (step 4). In particular, configuration constraints that are only relevant for the current stage do not cause distraction in other stages, and constraints that are only valid for certain build task are only evaluated if the build tasks shall be executed. Furthermore, fine-grained management of stakeholder-specific permissions and restrictions becomes possible (step 5): A configuration constraint attached to a stakeholder can warn or even prevent particular configuration edits (e.g., forbid the junior engineer to set feature attribute *numberOfConnections* to more than 10.000).

Finally, when all constraints attached to a build task—and to all its dependent build tasks—are fulfilled, a stakeholder may advise the *build interpreter* to execute it (step 6). The interpreter forwards the actual execution of the build task and each of its dependents to dedicated plug-ins. This way, build management across build tools becomes possible.

4. STAGED DERIVATION MODELING

For applying the approach, the staged derivation process needs to be modeled using StageModel and StageInstance models. Although concise, the modeling language comprises all information elements required to enable tool support for such scenarios as described in Section 2. The three essential model elements are *ConfigurationStages*, *BuildTasks*, and *StakeholderRoles* (cf. Figure 6).

Each *StageModel* has a number of *ConfigurationStages* assigned. The stages are ordered via *followsOn* associations. Each of the stages may have an arbitrary number of *BuildTask* elements. Such an element represents an actual build task in the product line asset base by describing its *file* (e.g., src/Makefile), its type (e.g., GNU make), and its target name (e.g., all). Each configuration stage has a *defaultBuildTask* that denotes the final build task that needs to be successfully executed in order to proceed to the next configuration stage.

Build task order dependencies can be modeled between build tasks of different type (e.g., make, ant) with *dependsOn* associations. In some cases, a modeled build task dependency has an actual correspondence in the asset base managed by an external tool. One particular example is GNU make, which allows for defining prerequisite targets for a make target. This fact can be modeled with *managedDependsOn* associations. This way, the developed tooling knows about the build task dependency, while it can leverage the optimization of the external tool during product generation.

Each individual build task can have *Validator* elements assigned. A validator denotes a set of build-task–specific constraints that need to hold before the build task can be executed. A validator element identifies a set of constraint *files* to be checked by a constraint check plug-in.

In the simplest case, the plain Java plug-in can be used for implementing constraints. Then, a constraint is implemented as a simple Java method with boolean return value, and its implementer is in charge of how the constraint accesses the configuration data (e.g., via configuration-mechanism–specific Java APIs, such as the pure::variants API [4]). A further option is to use the approach in combination with pre-

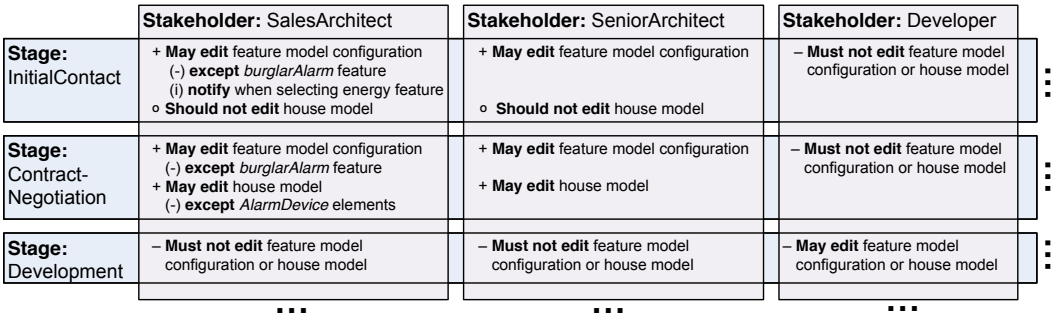

Figure 4: The permissions of *SalesArchitect*, *SeniorArchitect*, and *Developer*.

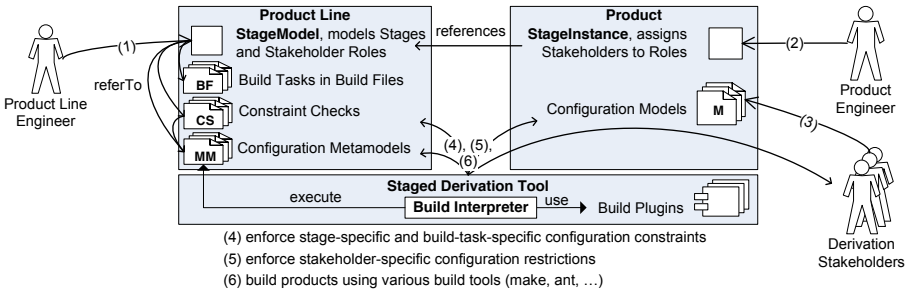

Figure 5: Application outline of the staged derivation automation approach.

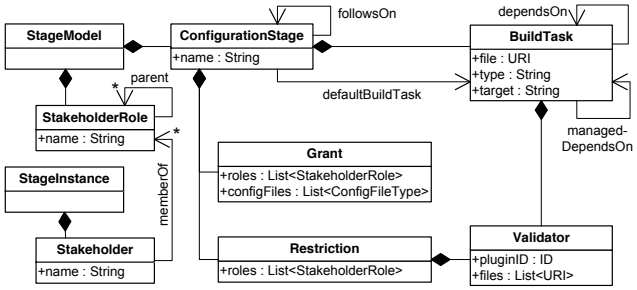

Figure 6: The stage derivation model specifies the configuration stages, build tasks, stakeholder roles, and the constraints attached to them.

vious work from the author, the PLiC framework [12]. The PLiC framework automatically extracts EMF [13] metamodels and models from the product lines and product involved. This facilitates the use of the constraint check plug-ins for the model-based languages Xpand Check [22] and OCL, as presented in [12]. In the meantime, additional plug-ins have been developed to access the generated EMF models from constraints implemented in Xtend 2 [28] and in Java. Therefore, the PLiC framework uses EMF functionality to dynamically generate Java APIs from the previously generated metamodels. This way, comfortable access to the EMF models generated from configuration files becomes possible, for all the configuration file types mentioned in footnote 1.

Next to these build-tasks–specific constraints, it is also possible to model stakeholder roles and their configuration permissions and restrictions. Therefore, each *StageModel* defines a set of *StakeholderRoles*. These roles inherit permissions and restrictions from other roles via *parent* associations. *Grant* and *Restriction* model elements, which are defined for each configuration stage, reference the defined stakeholder roles. A *Grant* element defines which *roles* are generally allowed to access a certain *configFile* in the configuration stage. All configuration files a product may provide are modeled on product line level as so called *ConfigFileType* model elements (not shown in Figure 6). The general allowance to edit individual configuration files can be limited by *Restriction* elements. Each restriction assigns a set of *roles* to a set of *Validator* elements. The constraints referenced by those elements denote stakeholder-specific restrictions. These can be used, for example, to warn or prevent inexperienced stakeholders from editing configuration options with unexpected side effects. Finally, a mechanism is provided to model the single stakeholders (e.g., the sales architect Bob) in a product derivation project. Therefore, each *StageInstance* model can define *Stakeholders* that can be members of multiple *StakeholderRoles* from which they inherit grants and restrictions.

Creating the stage model and mining the constraints (e.g., build-task- or stakeholder-specific constraints) requires expert domain knowledge, which can be gathered in workshops with domain experts. While we have not collected comprehensive data on the necessary information mining effort, our experience from previous work [12] is that it can be a rather straight forward process for embedded software experts to learn a constraint language and to define configuration constraints themselves, in case the language came with comprehensive editor support, as it is also the case for the tooling in this paper.

5. TOOL SUPPORT VIA CONSTRAINTS

After the product line engineer has created the stage model and has implemented the desired constraint checks, the developed tooling takes this information as input to provide fine-grained and comprehensive staged derivation support. For providing tool support, one further assumption is made: The developed tooling is assumed to extend an IDE, such as Eclipse, which is used for configuration.

Subsequently, this section formally models the examples from Section 2 as stage models with constraints, and illustrates the tool support that can be provided, for *basic support and build task integration* (Section 5.1) and *stakeholder guidance* (Section 5.2).

5.1 Basic Support and Build Task Integration

Figure 7 depicts the model for the example presented in Figure 3. It shows the *InitialContact* stage modeled in the StageModel during domain engineering of SmartHome.

Figure 7 describes the available build tasks: *generateProductSpecification*, *generateCommercialConditions*, and *calculateEnergySavingROI*. The model unambiguously defines these build tasks via denoting their file, their type, and, if required, the name of their build target. To provide the configuration stage with a single standard build task, an additional, "virtual" build task model element has been added (*initDefaultTask*). It does not represent an actual build task in the product line asset base. If it is executed via the build interpreter (cf. Figure 5), however, the two actual build tasks it *dependsOn* are invoked: *generateProductSpecification* and *generateCommercialConditions*. Finally, some of the build

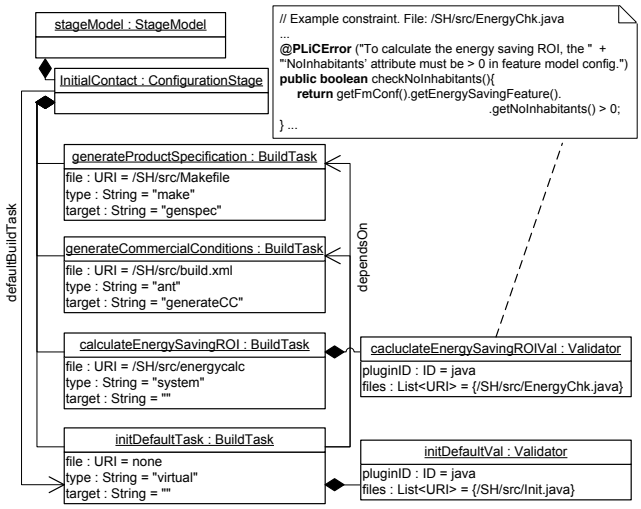

Figure 7: Build tasks of stage *InitialContact*.

tasks have *Validator* elements attached, which link to implemented constraint files. They are implemented in Java and use PLiC framework functionality for conveniently accessing the configuration data via EMF models.[2]

The constraint in Figure 7 defines that the *NoInhabitants* attribute in the feature model configuration file must be set to a value higher than zero in order to execute the *calculateEnergySavingROI* build task. Finally, constraints that need to hold in order to complete the stage ("basic support" according to Section 2) can simply be modeled by attaching constraints to the default build task of the stage (*initDefaultVal* Validator in Figure 7).

The model and the constraints serve as input for the developed tooling. It provides for build-task–specific consistency checking and for execution of build tasks of arbitrary type, as described subsequently.

5.1.1 Build-Task–Specific Consistency Checking

For product derivation, the configuring stakeholder first selects the current stage and the prospected build task from a GUI (see Figure 8, upper part). The prospected build tasks are those that the stakeholder wishes to execute subsequently. When selected, a background builder traverses the StageModel and only evaluates those constraint that are valid for the current stage and the prospected build task, together with all constraints of dependent build tasks (modeled via *dependsOn* and *managedDependsOn* associations). The inconsistencies are reported with the textual messages attached to constraints in the Eclipse problems view (cf. Figure 8, lower part).

The constraints are reevaluated in the background each time a configuration artifact changes, or when the current stage or the prospected build task are changed in the GUI. By making the validity of the configuration dependent on the current stage and the prospected build task, the constraint

[2]In Figure 7, for example, the Java class of the shown constraint inherits from a class generated by the PLiC framework. It provides the method *getFMConf()*, which facilitates convenient access to the EMF model of the feature model configuration of SmartHome.

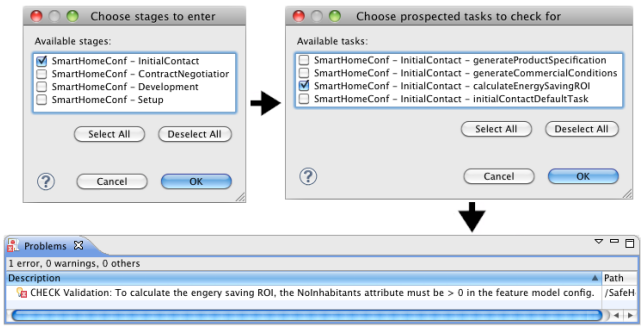

Figure 8: GUIs facilitate selecting the current configuration stage and the prospected build task, or display the currently relevant set of inconsistencies.

violations that the tool reports are minimized to the subset that is *actually* necessary to accomplish the prospected build tasks. This allows configuration stakeholders to strictly focus on the actually relevant configuration constraints.

This is particularly useful for long-running stages in which, most of the time, only intermediate build tasks are executed. For example, if the stage *Development* of SmartHome requires the implementation of additional modules, the developer will most of the time only execute the build tasks for compiling and testing the new module. Inconsistency messages regarding later build tasks in the same development stage, such as system tests, packaging, and installer configuration, only would distract the developer from focusing on the actual implementation task. Using the staged derivation tooling, the developer simply can select the "module test" build task as the prospected build task, and inconsistency messages regarding later build tasks are masked.

5.1.2 Build Task Execution

By declaring the *file*, the *type*, and the *target* of a build task in *BuildTask* model elements, the automated build task execution across build tools becomes easily possible. During product derivation, the stakeholder selects the build task to execute from a further GUI. After a final check for build task consistency, as described in the previous Section 5.1.1, a topological order is calculated based on the *dependsOn* links of build tasks and they are executed.[3] Further builder plug-ins can be added with little effort. Currently, builder plug-ins for ant, make, the modeling workflow engine (MWE) [20], and plain executables have been developed. See [11] for more details on their implementation.

Executing build tasks of various build tools via a single integrated GUI eases building product assets, while build task dependencies across build tools become explicit in models and are not hidden in any other kind of build script. Note that this mechanism should not be a replacement of existing build tools, such as ant or make. Rather, the build tasks modeled can be seen as the "public interface" of those tools. They ease product building for less experienced stakeholders by making the most crucial build tasks explicit.

[3]Note, that build tasks that only depend on others via *managedDependsOn* links can be ignored for actual build task execution. The corresponding build tools (e.g., make) decide whether to execute them.

5.2 Stakeholder Guidance

Figure 9 shows the StageModel of the motivating example in Figure 4. It defines three stakeholder roles: *SalesArchitect*, *SeniorArchitect*, and *Developer*.

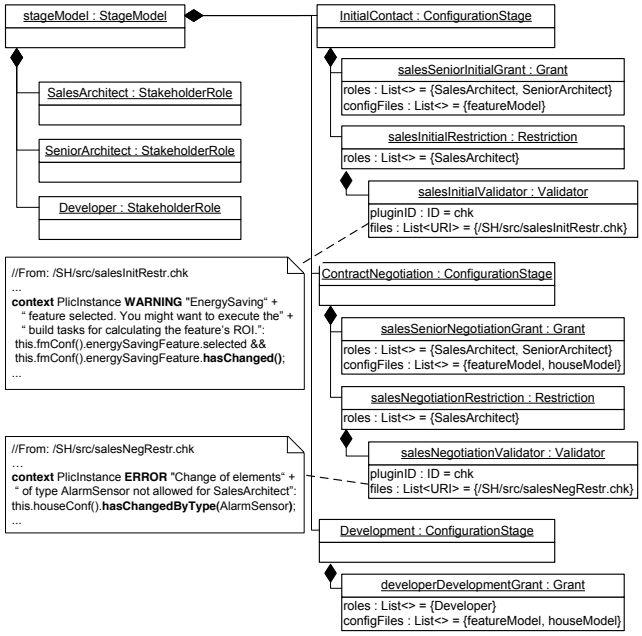

Figure 9: The permissions of the stakeholders of SmartHome modeled as Grants and Restrictions.

Within the *InitialContact* configuration stage, both the sales and the senior architect are granted access to the feature model only (*salesSeniorInitialGrant*). For the sales architect role, a further restriction (*salesInitialRestriction*) is defined. It contains one validator element (*salesInitialValidator*) pointing to a constraint file implemented in Xpand Check. It will remind the sales architect to execute the build task for calculating the ROI of the energy feature as soon as it is set from false to true.

Constraints that query whether a configuration option has just been *changed* need to be implemented in a different way than constraints for conventional consistency checks, as they need to access the configuration data both *before* and *after* the change. For this purpose, several convenience functions have been implemented that provide access to the new state, the old state, and the differences between the configurations. In Figure 9, for example, the implemented constraint makes use of the function *hasChanged()*, which evaluates whether the selection state of the energy saving feature has been altered during configuration.

In the second configuration stage *ContractNegotiation*, both sales and senior architect are granted change permissions to both the feature model and the domain-specific house model. Again, the possible actions of the sales architect are restricted. A further convenience function, *hasChangedByType()*, enables to query whether any model element of a certain type in the whole subtree of a model has changed. Note that the constraint uses the *ERROR* keyword of Xpand Check (not the *WARNING* keyword as the previous one). This indicates that the constraint strictly needs to hold, and the sales architect should not have any measures to circumvent this. Finally, during *Development*, only the *Developer* is allowed to change the configuration models (*developerDevelopmentGrant*). The following section explains the tool support provided to stakeholders on the basis of this model.

5.2.1 Tool Support for Stakeholder Guidance

During product derivation, the actual configuring *Stakeholders* (e.g., sales architect Bob), and their relation to *StakeholderRoles* (via the *memberOf* association), are defined in the StageInstance model (cf. Figure 6). Currently implemented via a login GUI, a user identifies as a certain stakeholder. From this time on, the files the stakeholder opens are monitored via the Eclipse infrastructure. When opening one of the product's configuration files, a lookup is started whether the stakeholder is granted change rights during the currently running configuration stage (*Grant* model element). If this is not the case, the Eclipse infrastructure is instructed to disable saving.[4]

In case the tool grants access, the stakeholder may freely edit the configuration file until saving it. As the Eclipse IDE enables hooking into the save functionality, it becomes possible to execute a constraint on the new state of the configuration file. If the API supports it (as the *hasChanged()* function implemented for Xpand Check), constraint may also access the differences to the old state of the configuration model. In case all stakeholder-specific constraints hold, the save action will be completed. Otherwise, the message attached to the constraints is displayed, which will inform the stakeholder which changes resulted in warnings or errors.

Warnings can be used to explain unexpected side effects or give hints how to proceed in the derivation process (e.g., execution of the ROI calculation build task). Errors result in strictly prohibiting the saving of a configuration file. While possible, strict prohibition may not always be the best option. Sole warnings, together with useful warning messages, may be as useful, but less incommoding to stakeholders.

6. APPLICATION EFFORT AND CHECKING PERFORMANCE

The goal of this section is to provide evidence for two central aspects concerning the practicability of the approach: light-weight applicability and technical scalability. In order to do so, two questions are crucial: Which effort is necessary to apply the approach to a legacy product line, and what is the performance of the tool regarding constraint checks?

6.1 Light-Weight Application

Enabling a product line for the approach requires identifying the crucial stages, stakeholders, and build tasks and modeling them. As the staged derivation process of SmartHome has been devised, no effort (e.g., in person hours) could be collected. The time effort for required expert interviews, however, should remain in reasonable boundaries.

The declaration effort for the stage model of SmartHome is shown in Table 1. On average, three to four model elements (classes, attributes, or references) are required to model one entity (88 model elements to 26 entities in total). This ratio

[4]Disabling saving works for any configuration file type, as long as files are opened with an editor that plugs into Eclipse. For example, all seven configuration file types supported by the PLiC framework [12] (cf. footnote 1) have such editors.

Table 1: Number of declared stage model elements.

Conf. Stages	Build Tasks	Stakeholder Roles	Further Entities	Total	Decl. Model Elements*
4	10	4	8	26	88

Conf.: Configuration | Decl.: Declared
* Including model classes, attributes, and references

is due to the fact that several entities require multiple data elements for characterization. For example, for a single build task, five model attributes need to be declared: an identifying name, the build file location, the build target name, the build tool (ant, make, etc.), and dependencies to other build tasks. Still, the effort remains in reasonable boundaries so that the required application effort can be considered light-weight.

The last step before applying the approach requires mining and defining the derivation constraints. Note that mining constraints is also required in other tool-supported configuration approaches that do not provide staged derivation support. In practice, the mining effort for staged derivation constraints can be measured in person hours spent in workshops with product line experts, which was not possible for the SmartHome demonstrator example. As constraints can be added incrementally, the effort spent in defining them can be flexibly adapted to the resources available.

Finally, the difficulty of constraint definition depends on the language and API used and cannot be generally assessed. In this paper, the PLiC framework [12] is used to access the configuration data. It facilitates constraints via several languages, such as Xpand Check, Xtend 2, and Java with EMF model access. All these languages can provide intelligent editors, which know about the type and name of available configuration variables and can provide functionalities such as tab completion and syntax checks, which should ease constraint definition considerably.

As also argued in [12], the PLiC framework as well only requires little application effort: For the seven configuration mechanisms mentioned in footnote 1, it suffices to create one model describing the configuration mechanisms used by the product line and one that defines the locations of the configuration files of the concrete product. For the SmartHome case, 26 model elements are needed in total. This information is sufficient for the PLiC framework to automatically create corresponding configuration models in EMF format of all configuration variables of the product line, on which then constraints can be defined.

6.2 Scalability of Constraint Checking

The constraints effectively implement the automation of product derivation. They provide for the tailored configuration checks and implement fine-grained configuration permission rules. As constraints need to be reevaluated each time a configuration artifact changes, a reasonable throughput of constraint checks remains to be shown. Again, the achievable performance depends on the used access mechanism to the configuration data. Table 2 shows the performance of three different constraint checking mechanisms, when choosing the PLiC framework for configuration data access: Xpand Check, Xtend 2, and Java with EMF model access.

All benchmarks were executed on a laptop with Core 2 Duo 2.4 GHz a hundred times in a newly started Eclipse instance. The initial measurement shown in a separate column reveals the additional execution time for just-in-time compilation.

Table 2: Constraint execution performance.

Performance	Init. (ms)*	For 100 Executions (ms) Min	Max	Average	Constr./ second
Xpand Check, Eclipse 3.5					
30 Constr.**	713	425	853	**513**	**58**
Xtend 2, Eclipse 3.7					
30 Constr.	45	1	40	**4**	**7915**
1.000 Constr.***	564	39	846	**65**	**15274**
10.000 Constr.	5012	483	1489	**634**	**15765**
Java, Eclipse 3.7					
30 Constr.	81	1	66	**4**	**7425**
1.000 Constr.	580	29	298	**41**	**24307**
10.000 Constr.	2123	358	1033	**428**	**23370**

Init.: Initial execution | Constr.: Constraints
* Xtend 2 and Java require just-in-time compilation for initial execution. Therefore, the first execution of constraints has been measured separately.
** The 30 constraints are actual product line constraints mined in expert workshops. Each of these constraints accesses 5 model elements on average.
*** The 1000 and the 10000 constraints are generated synthetically. Each constraint accesses 15 arbitrary configuration models elements, to also incorporate cases where product line constraints become more complex.

The initially developed Xpand Check validation plug-in (Eclipse 3.5) shows the general feasibility of the approach. The 30 product line constraints mined in an workshop with domain experts [12] were executed in about half a second on average (513 ms). Due to the fact that the language is interpreted textually, it was foreseeable that the language would not scale well for a larger number of constraints.

In order to evaluate scalability, the PLiC framework was ported to Eclipse 3.7 and a validation plug-in for the Xtend 2 language and for Java was developed. Xtend 2 constitutes the successor of the Xpand Check language family and is compiled to the Java Virtual Machine.

As can be seen, the performance of both Xtend 2 and Java is around two orders of magnitude higher than for Xpand Check. The 30 constraints mined in a workshop were executed with a rate of 7.915 (Xtend 2) and 7.425 (Java) constraints per second. The initialization effort for the startup of the respective validation plug-ins yet has a strong influence on the results. Therefore, the constraint execution rate improves when executing more constraints. For executing the 1.000 or the 10.000 randomly created synthetic constraints, the impact of plug-in initialization becomes irrelevant. Then, the throughput of the plug-ins is approximately 15.000 constraints per second for Xtend 2 and 24.000 constraints per second for Java.

An average waiting time of less than a second (634 ms and 428 ms) for 10.000 constraint checks is still acceptable, and incorporates many practical cases, such as, for example, the complete number of cross-tree constraints in the Linux Kernel [17]. Finally, although there is a certain variance in the performance figures (indicated as minimal and maximal values of execution times, presumably caused by the Java garbage collector and other interfering IDE and operating system tasks), it is within reasonable boundaries. In total, this shows that the constraint checking performance scales reasonably well to thousands of constraint checks per second.

7. DISCUSSION

This section discusses the appropriateness of stage model of the SmartHome product line, the expressiveness of the stage modeling language, and reliable figures on product derivation improvement.

The SmartHome stage model was devised by the author of this paper. Three measures were taken to foster the description of a realistic scenario. Early stages, such as the initial contact stage, are based on industrial experience

of the author, as published in [25]. Later stages, such as development and installation, are closely related to the actual code base of SmartHome, which has not been developed by the author. Finally, the fact that the stages are derived from industrial experience is a common practice in staged configuration research (e.g., [8, 18, 7, 14, 5, 3], cf. Section 8).

As there is little publicly available data on industrial-scale staged derivation, a stage modeling language (cf. Section 4) cannot be shown to be "sufficient" for real-world scenarios, yet. Still, as all involved participants (stages, build tasks, stakeholders), are first class modeling elements, and arbitrary complex constraints can be implemented in languages such as Java, the current language is in this regard more expressive than all previous approaches (cf. also Section 8). Finally, if there is the need for further modeling constructs (e.g., stage models with branch conditions, synchronization barriers, or loops), the stage modeling language may be adapted—there is currently no restriction except that the resulting language can be formulated as a metamodel.

With basically all other product line approaches that support automated product derivation [4, 16, 23, 27, 2], the PLiC approach shares the assumption that ensuring configuration validity not only improves on product quality due to avoidance of derivation errors, but also has a positive impact on derivation time. Reliable quantitative results on derivation time improvement, however, have not been produced up to now. In this regard, the paper argues with two particular features of the approach: It is applicable in a light-weight manner and it can scale to thousands of constraint checks. This creates an environment where introduction risk is small and potential benefits due to time and quality improvements are high, so that the approach gains high attractiveness for its application in practice, where reliable quantitative data can be gathered.

8. RELATED WORK

The presented approach is unique in explicitly integrating stakeholder roles and build tasks into the staged derivation process and in providing corresponding tool support. Furthermore, it is not tied to a certain configuration mechanism as previous work, which often also has only limited means for modeling the actual stages.

Previous automated concepts for basic staged derivation support, as defined in Section 2.2, are mostly limited to feature models as configuration mechanism. In [8], Czarnecki et al. propose two solutions for arranging configuration choices: specialization and multi-level feature modeling. Specialization denotes the step-wise selection of features one after another. It does not provide a notion similar to a "stage", which groups multiple related configuration choices.

In multi-level feature modeling [8], each stage or level must be modeled in a dedicated, separate feature model. As soon as one stage is completely configured, the model at the lower level is preconfigured according to the previously made configuration choices. A dedicated model of stages is missing, so that the configuration mechanism and the implicit stage model are highly tangled.

Classen et al. have developed a formalization semantics for multi-level feature modeling [7] in order to point out ambiguities and suggest improvements for the original concept. Mendonca at al. [18] extend basic feature modeling with decision sets, which partition the features of a feature model. According to priorities attached to each decision set, and to the configuration constraints defined in the feature model, a partial order is calculated for the decision sets. Based on this input, the sequence of stages is automatically calculated, instead of explicitly defined, as in other approaches.

Rabiser at al. [26] present the only instance of a staged configuration approach that is based on decision modeling. They use distinct views to assign decisions to three different stages (called levels in [26]). As they allow decisions to appear in more than one view, they are more flexible than the previously mentioned approaches, for which the assignment of features to stages is fixed. Although the authors only present their approach in context of a concrete product line project, there seems to be potential for generalization to various other kinds of staged configuration processes.

Hubaux et al. [1, 14], a feature modeling extension for explicit modeling of configuration stages with a so called "configuration workflow". Each stage in the workflow is assigned a set of configuration options (a "view" on the feature model) and a condition that needs to be fulfilled in order to proceed to the next stage. This enables more flexible definition of configuration stages as in previous feature-model–based approaches. In particular, it is not necessary that each configuration stage is modeled as a separate feature model.

Currently, Hubaux et al. provide the most advanced approach up to date by decoupling configuration stages and the configuration options. Still, the approach only concerns feature modeling and does not integrate build tasks or can provide fine-grained stakeholder derivation process guidance. In providing such capabilities, and in being independent of the used configuration mechanism, the approach presented in this paper adds to previous work both from a functional and from a generalization perspective.

9. CONCLUSION

This paper has presented a light-weight approach for staged derivation tool support. For application, it requires only little derivation process modeling and the implementation of the constraint checks. Its basic application prerequisites are a constraint language with access to the configuration data and an extensible configuration IDE to add the tool support.

The approach comprises a concise stage modeling language, which explicitly distinguishes stages, stakeholders, and build tasks to which constraints can be attached. This facilitates more comprehensive tool support than previous approaches: Next to basic staged derivation support, it provides stakeholders with guidance via showing individual configuration hints, or by even hindering some stakeholders to do specific actions in certain stages. Additionally, the tooling integrates build-task–specific consistency checking and actual product building into the staged derivation process.

The effort for initially applying the approach to the SmartHome product line has been little—it basically only required creation of the stage model comprising 88 model classes, attributes, and references. Finally, also the technical scalability of the approach to ten thousands of constraint checks per second could be proven.

What remains to be shown is the practical scalability: Will the number of defined constraints still be manageable for industrial scale scenarios? Or is it necessary to sacrifice on the expressiveness of the stage model or the constraint language in order to facilitate formal analysis? As the tooling can be applied in a light-weight manner and as it reveals scalability regarding the number of constraint checks, the

necessary prerequisites to gather these insights via practical application of the approach are there.

10. ACKNOWLEDGEMENTS

I would like to thank Paul Grünbacher for his valuable comments on an earlier version of this paper.

11. REFERENCES

[1] E. K. Abbasi, A. Hubaux, and P. Heymans. A toolset for feature-based configuration workflows. In *15th Software Product Line Conf. (SPLC '11)*, Washington, DC, USA, Aug. 2011. IEEE Computer Society.

[2] T. Asikainen, T. Männistö, and T. Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Advanced Engineering Informatics*, 21:23–40, January 2007.

[3] E. Bagheri, T. D. Noia, D. Gasevic, and A. Ragone. Formalizing interactive staged feature model configuration. *Journal of Software Maintenance and Evolution: Research and Practice*, 23, 2011.

[4] D. Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2006. `http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf`, visited 2011-11-12.

[5] M. Boskovic, E. Bagheri, D. Gasevic, B. Mohabbati, N. Kaviani, and M. Hatala. Automated staged configuration with semantic web technologies. *International Journal of Software Engineering and Knowledge Engineering*, 20:459–484, 2010.

[6] G. Chastek, P. Donohoe, and J. McGregor. A study of product production in software product lines. Technical Report CMU/SEI-2004-TN-012, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, USA, Mar. 2004.

[7] A. Classen, A. Hubaux, and P. Heymans. A formal semantics for multi-level staged configuration. In *3th Int. W'shop on Variability Modelling of Software-intensive Systems (VAMOS '09)*, pages 51–60, 2009.

[8] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[9] S. Deelstra, M. Sinnema, and J. Bosch. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):173–194, Jan. 2005.

[10] C. Elsner, D. Lohmann, and W. Schröder-Preikschat. Fixing configuration inconsistencies across file type boundaries. In *37th EUROMICRO Conf. on Software Engineering and Advanced Applications (SEAA '11)*, pages 116–123. IEEE, Aug. 2011.

[11] C. Elsner, D. Lohmann, and W. Schröder-Preikschat. An infrastructure for composing build systems of software product lines. In *15th Software Product Line Conf. (SPLC '11), Volume 2*, pages 18:1–18:8, New York, NY, USA, Aug. 2011. ACM. (MAPLE/SCALE '11 Proceedings).

[12] C. Elsner, P. Ulbrich, D. Lohmann, and W. Schröder-Preikschat. Consistent product line configuration across file type and product line boundaries. In Kang [15], pages 181–195.

[13] Eclipse Modeling Framework Homepage. `http://www.eclipse.org/emf/`, visited 2011-11-12.

[14] A. Hubaux, A. Classen, and P. Heymans. Formal modelling of feature configuration workflows. In Muthig and McGregor [19], pages 221–230.

[15] K. Kang, editor. *14th Software Product Line Conf. (SPLC '10)*, volume 6287 of *LNCS*, Heidelberg, Germany, Sept. 2010. Springer.

[16] C. W. Krueger. BigLever software Gears and the 3-tiered SPL methodology. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications (OOPSLA '07)*, pages 844–845, New York, NY, USA, 2007. ACM.

[17] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski. Evolution of the Linux kernel variability model. In Kang [15], pages 136–150.

[18] M. Mendonca, D. Cowan, and T. Oliveira. A process-centric approach for coordinating product configuration decisions. In *40th Hawaii International Conference on System Sciences (HICSS '07)*. IEEE, 2007.

[19] D. Muthig and J. D. McGregor, editors. *13th Software Product Line Conf. (SPLC '09)*, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[20] Eclipse Modeling Framework Technology (EMFT) - Modeling Workflow Engine (MWE). `http://www.eclipse.org/modeling/emft/?project=mwe`, visited 2011-11-12.

[21] P. O'Leary, R. Rabiser, I. Richardson, and S. Thiel. Important issues and key activities in product derivation: Experiences from two independent research projects. In Muthig and McGregor [19], pages 121–130.

[22] Eclipse Xpand Homepage. `http://www.eclipse.org/modeling/m2t/?project=xpand`, visited 2011-11-12.

[23] R. Rabiser, P. Grünbacher, and D. Dhungana. Supporting product derivation by adapting and augmenting variability models. In *11th Software Product Line Conf. (SPLC '07)*, pages 141–150, Washington, DC, USA, 2007. IEEE Computer Society.

[24] R. Rabiser, P. Grünbacher, and D. Dhungana. Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Information and Software Technology*, 52:324–346, March 2010.

[25] R. Rabiser, W. Heider, C. Elsner, P. Grünbacher, and C. Schwanninger. A flexible approach for generating product-specific documents in product lines. In Kang [15], pages 47–61.

[26] R. Rabiser, R. Wolfinger, and P. Grünbacher. Three-level customization of software products using a product line approach. In *42nd Hawaii Int. Conf. on System Sciences (HICSS '09)*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[27] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: A framework for modeling variability in software product families. In *11th Software Product Line Conf. (SPLC '07)*, Heidelberg, Germany, 2007. Springer.

[28] Eclipse Xtext Homepage. `http://www.eclipse.org/Xtext/`, visited 2011-11-12.

[29] M. Völter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. *11th Software Product Line Conf. (SPLC '07)*, pages 233–242, 2007.