

# Consistent Product Line Configuration Across File Type and Product Line Boundaries<sup>\*</sup>

Christoph Elsner<sup>1</sup>, Peter Ulbrich<sup>2</sup>, Daniel Lohmann<sup>2</sup>,  
Wolfgang Schröder-Preikschat<sup>2</sup>

<sup>1</sup>Siemens Corporate Research & Technologies, Erlangen, Germany

<sup>2</sup>Friedrich-Alexander University Erlangen-Nuremberg, Germany

`christoph.elsner.ext@siemens.com`

`{ulbrich,lohmann,wosch}@cs.fau.de`

**Abstract.** Creating a valid software configuration of a product line can require laborious customizations involving multiple configuration file types, such as feature models, domain-specific languages, or preprocessor defines in C header files. Using configurable off-the-shelf components causes additional complexity. Without checking of constraints across file types boundaries already at configuration time, intricate inconsistencies are likely to be introduced—resulting in product defects, which are costly to discover and resolve later on.

Up to now, at best ad-hoc solutions have been applied. To tackle this problem in a general way, we have developed an approach and a corresponding plug-in infrastructure. It allows for convenient definition and checking of constraints across configuration file types and product line boundaries. Internally, all configuration files are converted to models, facilitating the use of model-based constraint languages (e.g., OCL). Converter plug-ins for arbitrary configuration file types may be integrated and hide a large amount of complexity usually associated with modeling. We have validated our approach using a quadrotor helicopter product line comprising three sub-product-lines and four different configuration file formats. The results give evidence that our approach is practically applicable, reduces time and effort for product derivation (by avoiding repeated compiling, testing, and reconfiguration cycles), and prevents faulty software deployment.

## 1 Introduction and Motivation

Creating consistent configurations for larger-scale product lines often is a laborious task affecting various types of configuration files with subtle dependencies. Whereas customer-visible variability might be bound via selecting options in a feature model, the deployment of software to physical nodes may reside in domain-specific models or text files [10], while fine-tuning is done via preprocessor variables in C header files. Choosing certain features in the feature model may

---

<sup>\*</sup> This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/4 and SCHR 603/7-1.

impede certain choices in the domain-specific model, while setting a preprocessor variable in turn may presuppose a feature to be set. Configuration complexity increases even further when employing configurable off-the-shelf components (e.g., Apache, Oracle) or when building combined products including other product lines, which in turn expose their variability via certain configuration file formats.

One solution often recommended and applied to ease product configuration is to explicitly limit the scope of the product portfolio by offering only a small subset of possible configurations to the customer. When strictly applied, it is possible to use a single top-level configuration file (e.g., a feature model configuration) and automatically generate all the remaining configuration files no matter which format [4]. However, a very common scenario in industry [6] is that a customer actively negotiates the characteristics of the product. This makes application engineering a laborious task requiring implementation of additional modules, but also fine-grained configuration and customization at various locations with subtle, implicit dependencies. In such a case, a predefined subset of products, configurable via a single entry point, is not realistic.

Consistency constraints that span different configuration file types need to be enforced at configuration time to prevent inconsistencies, in particular if the misconfiguration is known to manifest only sporadically during runtime (e.g., due to race conditions). One promising approach for consistency checking is to transform all configuration files into a common representation and use constraint-checking languages. Model-based development offers powerful constraint languages, such as OCL [13] or XPand Check [21], and asserts that virtually every artifact may be transformed into a model [3]. In fact, for various file types, there do exist converters or converter frameworks into the modeling world, for example, for textual grammars (XText [22]) or XML files (XMLSchema [9]). However, combining all these single frameworks in an ad-hoc manner for a specific product line in order to check constraints is a considerable amount of work. A product line engineer familiar with a particular domain (e.g., embedded systems) might not be a modeling expert as well, who is able to set up, combine, and tame all the modeling technologies and tools. Moreover, such an infrastructure should be developed in a generic way in order to apply it to various product lines. Up to now, at best ad-hoc solutions for doing constraint checking are applied in industry. As we have argued in a previous paper [8], a general approach for constraint checking across arbitrary configuration file types is missing.

To tackle this problem, we have developed an approach and corresponding tooling for an extensible constraint-checking infrastructure. It supports scenarios where multiple configuration file types are involved, which may even be spread over several sub-product-lines and off-the-shelf components. Our infrastructure provides for comfortable definition and checking of constraints independently of the configuration file type used and achieves this by transforming all configuration files to models in a transparent way. We have already developed plug-ins for several file types (such as feature models, XMLSchema, Ecore [9], KConfig, C preprocessor defines) and support different constraint definition languages (OCL, XPand Check). Arbitrary file formats may be supported—by writing con-

verter plug-ins for transforming configuration formats into the internally used modeling format (Eclipse Ecore). We have applied our approach to a quadrotor helicopter product line (I4Copter) comprising three sub-product-lines and four different configuration file formats. By discovering and resolving inconsistencies at configuration time, we could considerably reduce the time and effort for product derivation. Detecting them at later stages of product derivation—at compile time, testing time, or after deployment—, as it was necessary beforehand, was far more costly.

After introducing the I4Copter product line and its configurability as a concrete scenario guiding through the paper in Section 2, we contribute a practice-oriented approach for mapping configuration files of different types to models (Section 3). Then, we account for our generic, extensible constraint-checking framework, which provides comfortable means for constraint definition while minimizing the necessary knowledge about model-based development for the product line developer and the configuration engineer (Section 4). Sections 5 and 6 report on its application to the I4Copter and the achieved results. Finally, we discuss our approach and related work (Sections 7 and 8).

## 2 Scenario: Quadrotor Helicopter Product Line

The evaluation platform for our product line research is the *I4Copter* [19] quadrotor helicopter, which has been designed and developed to resemble embedded real-time systems arising in real-world product line scenarios. The I4Copter software product line comprises three sub-product-lines: one product line for application logic, which also models the interface to the hardware (*CopterSwHw*), the commercial operating system product line *PXROS*<sup>1</sup>, and, as an alternative operating system product line, the department-internal *CiAO* [11].

The *CopterSwHw* product line is implemented in C++; various software features are optional and alternative (approximately 50 features in total). There are several hardware variants of the I4Copter, comprising different frames, sensors, and actuators, so that the hardware as well forms a product line. The software depends on information about the available hardware components (approximately 60 features). Both the software part and the available hardware components of the *CopterSwHw* product line are configured via C preprocessor directives.

The application logic may run either on the operating system product line *PXROS* or on *CiAO*. For configuring *PXROS* (tasks and other parameters), a simple textual domain-specific language (DSL) is used, out of which a generator creates the corresponding C start-up code. While *PXROS* is more reliable and mature, the department-internal operating system product line *CiAO* is much more versatile. It uses pure::variants [2] feature models for configuring various functional and architectural properties and comprises an own XML dialect, defined in XMLSchema, for configuration of operating system tasks.

Domain-specific configuration constraints of the I4Copter span several files and are therefore hard to enforce (see also Figure 1). When equipping the hard-

---

<sup>1</sup> HighTec EDV Systeme GmbH - <http://www.hightec-rt.com/>

ware with an acceleration sensor using the Serial Peripheral Interface (SPI) bus, for example, this has to be described in the corresponding hardware header file (`AC_PRESENT`). The application software product line usually would (although not necessarily!) be configured with the corresponding sensor device driver in the software header file (`AC_DRIVER`). Using any SPI device requires that an SPI bus controller software module is present, which is implemented as an operating system task. Thus, a corresponding operating system task (`SPITask`) needs to be initialized appropriately either by using the DSL of PXROS or the XML language of CiAO. When choosing CiAO as the operating system, further configuration details need to be enforced, as CiAO is highly configurable. When including the `SPITask`, it must be ensured that it sends its messages to the bus with the same priority as the task that triggered the message. This requires selecting an appropriate priority mechanism in the feature model configuration, for example the priority inheritance mechanism (`kernel_pip`).

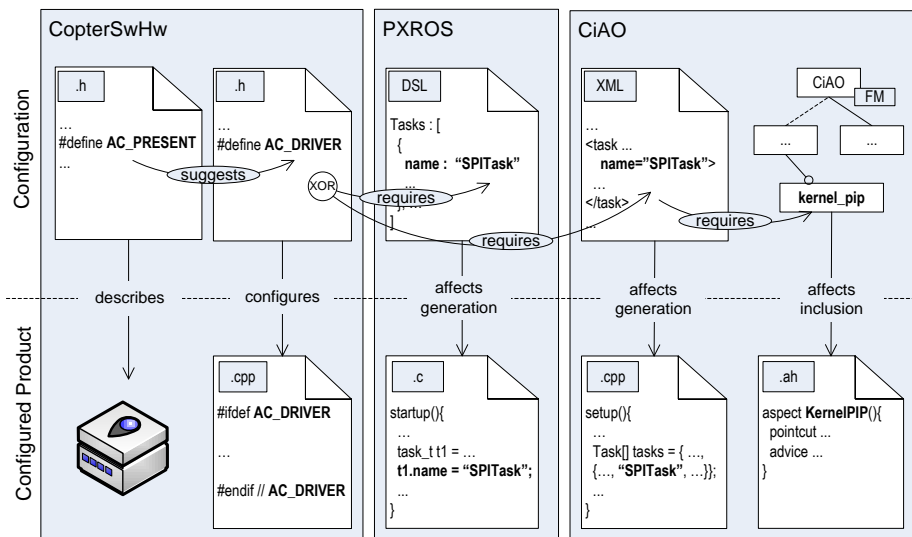


Fig. 1. Exemplary domain constraints within the I4Copter product line.

There exist various of such domain-specific constraints in the I4Copter product line. Some are rather recommendations or warnings, others are mandatory. The constraints span several different configuration file types and product lines. This makes the configuration of the I4Copter product line sufficiently complex to resemble the challenges faced in industrial-scale embedded systems development. We will return to the *I4Copter* as an illustrative example when we explain the general approach, its architecture, and validation.

### 3 Product Line Configuration and Modeling

Our approach is based on the assumption that every artifact involved in product line configuration has a corresponding representation in the modeling world [3]. Basically, the configuration of a concrete product is described by its *set of configuration files*. The product line itself defines, either implicitly or explicitly, the *set of configuration file types* it can deal with. As mapping of product lines and configurations into the modeling world is not without pitfalls, we will now first introduce general modeling terminology and then explain details how we perform the mapping of configuration files and configuration file types.

#### 3.1 Modeling Terminology

A *model* is a formal abstraction of a concept (e.g., a physical system or software) describing its concrete entities and relationships [16]. A model is abstract in the way that it is not tied to a certain textual or graphical representation. The formal rules, which specify the entity and relationship *types* allowed in a certain model, are provided by its *metamodel*. As an example, a simple metamodel for modeling operating system tasks will comprise the root entity type `TaskList`, which contains an arbitrary number of `Task` elements, which in turn have a `name` element of type string and a `priority` of type integer. A model conforming to this metamodel, for instance, defines a concrete `TaskList` comprising exactly one `Task` with `name = "SPITask"` and `priority = 1`. For specifying metamodels, a *metamodeling technology* is used (e.g., our implementation uses Eclipse Ecore).

Within one metamodeling technology, it is possible to define constraints, such as the SPI bus constraints described in Section 2, in a formal, machine-interpretable way. Common examples for such languages are OCL and XPand Check. Constraints on models usually are specified using the elements defined in the metamodel. For example, a constraint on a task model can leverage the fact that each `TaskList` has a number of `Tasks`, which in turn have a name and a priority. Querying whether there is any `Task` called “SPITask” for the whole system can therefore be formulated very concisely, for example in XPand Check, which implicitly iterates over sets: `myTaskList.tasks.name.contains("SPITask")`. Subsequently, we will describe how we map product lines and their configurations into the modeling world, that is, to metamodels and models.

#### 3.2 Mapping Product Line Configurations to Models

A configuration file is an artifact that specifies the characteristics of a product (e.g., a web-server configuration file, C header files, but also domain-specific model and text files). A *configuration file* therefore corresponds to a *model*. The elements and the relations allowed in the configuration file (i.e., the abstract syntax part of the *configuration file type* exempt from its concrete syntax) correspond to what is the *metamodel* of this model. Accordingly, a *configuration* of a product line can be seen as a *set of models*, the *product line* itself as a *set of*

*metamodels*. Although the mapping is straight-forward, it is still open how to actually *derive* metamodels from a product line.

We distinguish two classes of configuration file types, which differ in the way to derive their metamodel. Firstly, there are those for which the product line already provides an explicit specification file that can be converted to a *product-line-specific metamodel*. Secondly, there are those where this is not the case, and only a less expressive, *generic metamodel* can be used.

**Product-Line-Specific Metamodels via Specification File.** Some product lines comprise an explicit specification of what is a valid configuration file for them. For example, the CiAO product line defines the format for a valid task configuration XML file in XMLSchema. For PXROS, we developed a simple textual grammar specifying which constructs are valid in its domain-specific configuration file. Feature models can be interpreted as metamodels as well [17]. Finally, model-driven product lines specify their metamodels explicitly (e.g., [20]). Tools that map specification files to *product-line-specific metamodels* are already available in many cases, for instance, for XMLSchema files and for XText grammars there exist converters that derive the corresponding Ecore metamodels [9, 22].

**Generic Metamodels Without Specification File.** There are, however, also configuration files that lack any expressive and formal specification of what the valid constructs are in the context of one particular product line. This is, for example, the case for Java property files and for C header files containing preprocessor defines. In principle, arbitrary identifiers may be set to arbitrary values in a preprocessor define statement, such as `#define AC_PRESENT 1`. Which defines are necessarily required, optional, or unused, or what the permissible value ranges are for a particular product line, is not specified explicitly and can only be discovered by reading source code or documentation. Although a product line engineer could use this information to reconstruct a product-line-specific specification file (e.g., an XText grammar), this often will not be the case.

We therefore see the need to map certain file types to less expressive *generic metamodels*. For a C header file with preprocessor defines, for example, such a metamodel will only specify that a `DefineList` contains an arbitrary number of `DefineStatements` having an `identifier` of type string and a `value` of type string. This fact makes defining explicit constraints more chatty and error-prone. For example, having a specific metamodel, a constraint on the debug-level define may be formulated very concisely: `copterSwhw.debugLevel == 1`. Having only a generic metamodel, one needs to query the define value in a reflective way: `copterSwhw.getPropByName("debugLevel").toInteger() == 1`. However, the simplicity of such a generic metamodel also has one benefit: it can be reused across product line boundaries very easily.

**Wrap-Up.** To sum up, it is possible to map all configuration file types to metamodels and the corresponding configuration files to models. Tooling for conversion either exists or may be developed. However, combining all these single technologies and tools in an ad-hoc manner for a specific product line in order to check constraints is not appropriate. We will therefore present a corresponding generic tool framework in the next section.

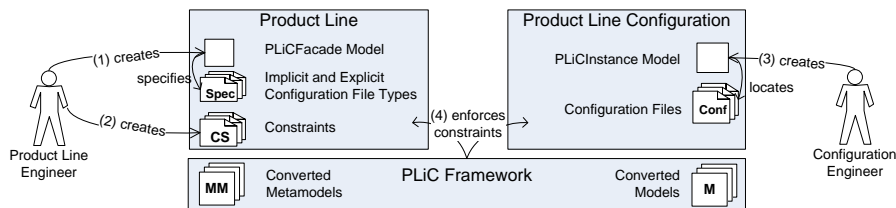
## 4 Product Line Constraint-Checking Framework

In this section, we present the *PLiC* (*Product Line Configuration*) framework, which allows for constraint-checking across product line boundaries and configuration file types, while minimizing the required modeling knowledge of product line and configuration engineers. The PLiC framework is implemented as an Eclipse extension and enriches the integrated development environment with a **builder** component, which performs model conversion and validation in the background when a configuration artifact changes within the workspace. Both model **converters** (for specific configuration file types) and model **validators** (for evaluating constraints of different checking languages) have been implemented using the Eclipse extension point mechanism. Thus, additional configuration file types and constraint-checking languages can be implemented and integrated easily.

We will describe the builder, the converters and validators that are already available, and the extensibility of the PLiC framework later in this section. First, we address the concepts that end users have to deal with.

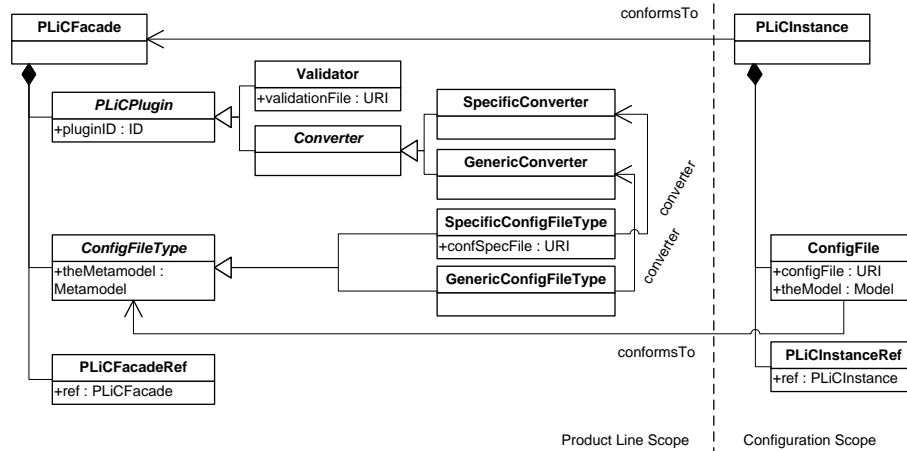
### 4.1 End User View on the PLiC Framework

There are two roles of end users: *product line engineer* and *configuration engineer* (cf. Figure 2). The former declares the set of possible configuration file types in a so called *PLiCFacade* model (step 1) and implements the domain constraints (step 2). The configuration engineer, in turn, creates the *PLiCInstance* model (step 3), which specifies the locations of the configuration files. The PLiC framework then permanently enforces the constraints on the configuration (step 4).



**Fig. 2.** Product line engineer and configuration engineer using the PLiC framework.

**The PLiCFacade model.** Within the *PLiCFacade* model (cf. Figure 3), the product line engineer declares three distinct sets of elements: the plug-ins required (*PLiCPlugins*), the configuration file types the product line can deal with (*ConfigFileTypes*), and references to other sub-product-lines (*PLiCFacadeRefs*). A *PLiCPlugin* has a unique ID to identify the plug-ins installed in the workbench. It may either be a *Validator* or a *Converter*. A *Validator* is parameterized with the path to a file (or directory) containing the set of constraints to



**Fig. 3.** First, the product line engineer defines a *PLiCFacade* model for each product line. It declares the plug-ins used, the configuration file types available, and references to other *PLiCFacades*. The configuration engineer, in turn, defines the corresponding *PLiCInstance* model. It specifies the locations of the actual configuration files and references to other *PLiCInstances*.

check. *Converters*, finally, exist in two flavors. *SpecificConverters* (in analogy to Section 3.2) convert a specification file to a product-line-specific metamodel. A *GenericConverter* simply provides a single, nonspecific metamodel (e.g., a C preprocessor define metamodel), that can be used in various product lines.

Accordingly, *ConfigFileType*s are either specific or generic. In case of a *SpecificConfigFileType*, the engineer needs to provide the URI to a specification file (XMLSchema, grammar, etc.) that may be transformed via the referenced *SpecificConverter*. For *GenericConfigFileType*s, this is not necessary as the metamodel of the referenced *GenericConverter* is generic and fixed.

**The *PLiCInstance* model.** In the *PLiCInstance* model, the configuration engineer first references the corresponding *PLiCFacade* model it conforms to. Then he declares all configuration files and their URIs within the file system (*ConfigFiles*). Each *ConfigFile* needs to reference the *ConfigFileType* it corresponds to. Finally, the configuration engineer draws the references to other used *PLiCInstances* (sub-product-line configurations).

## 4.2 Builder

The **builder** plug-in works in the background and is invoked on each change in the workspace. It keeps the created metamodels and models up to date, invokes the constraint checks, and displays the check results. It builds all projects in a linear order according to their project dependencies.

The builder achieves the actual conversion to metamodels and models by invoking the appropriate **converter** plug-ins. It stores the metamodels and models



in the file system and sets references to them using the *theMetamodel* and *theModel* member variables of all *ConfigFileTypes* and *ConfigFiles* (cf. Figure 3). Doing so, the PLiCFacade and PLiCInstance models provide single entry points for performing constraint checks. Eventually, the constraints checks referenced in the PLiCFacade model (via *Validator* elements) are evaluated on the PLiCInstance model calling the appropriate `validator` plug-ins. Subsequently, we will describe the development of converter and validator plug-ins.

### 4.3 Converters

`Converter` plug-ins perform the actual conversion to metamodels and models and need to implement certain interfaces. A generic converter needs to provide one method for querying its fixed metamodel and one for converting a configuration file to a model conforming to the metamodel. A specific converter works similar, however has, as an additional parameter, in each of its methods the specification file, which defines the product-line-specific metamodel.

Up to now, we have developed two generic converters and five specific converters. The generic converters comprise a converter for files containing statements of the form `#define <ID> <Value>` as output by the GNU compiler `gcc` when invoked over the source code with certain parameters. Furthermore, we developed a converter for property-value files, as common for Java.

The specific converters create metamodels for specification files in Ecore, XMLSchema, and XText textual grammars. EMF already provides Java APIs for this purpose, whose complexity is hidden behind the lean converter interfaces. Furthermore, we developed a specific converter for `pure::variants` feature models. We intentionally keep the metamodel generated from a feature model simple. In particular, we ignore any hierarchical or dependency information of features and only create one metamodel element for each feature, having the same attributes as the feature. Actually, this is the only information necessary for constraint checking, as the configuration editor of `pure::variants` will ensure that constraints defined *within* the feature model itself are adhered to. Finally, we developed a specific converter that generates a metamodel from KConfig files, which are used, for example, to specify the configuration options of the Linux kernel.

### 4.4 Validators

A `validator` plug-in evaluates the constraints for a certain constraint definition language. The corresponding interface comprises only one method, which receives the PLiCInstance object to check and the file containing the constraint rules and returns detailed information on warnings and errors that occurred during validation of the models. Note that the builder (cf. Section 4.2) has enriched the PLiCInstance object by setting the *theModel* property of each *ConfFile* element to the built models. Doing so, the PLiCInstance provides a single entry point to all generated models possibly subject to checking.

Currently, we provide two validator plug-ins, one for defining constraints in the OMG Object Constraint Language (OCL) [13] and one for the XPand Check

language [21]. Examples for constraints will follow in the next section, where we will present the application of the PLiC framework to the I4Copter.

## 5 Application Scenario: I4Copter

We have evaluated the approach and the PLiC framework with the I4Copter product line described in Section 2. Six steps need to be performed in general for any product line: (1) identify configuration file types, (2) select or develop converters, (3) create PLiCFacade projects and models, (4) define initial constraints, (5) configure products using PLiCInstance projects and models, and (6) constantly maintain and improve constraints during product line evolution.

**1. Identify configuration file types.** The first step is to identify the configuration file types of each involved product line and configurable component. Section 2 already comprises this task for the I4Copter. It comprises a hardware product line tangled with an application logic product line, both configured via header files. The operating system PXROS is configured via a configuration file DSL, while the operating system product line CiAO uses an XML dialect defined in XMLSchema and a pure::variants feature model for configuration.

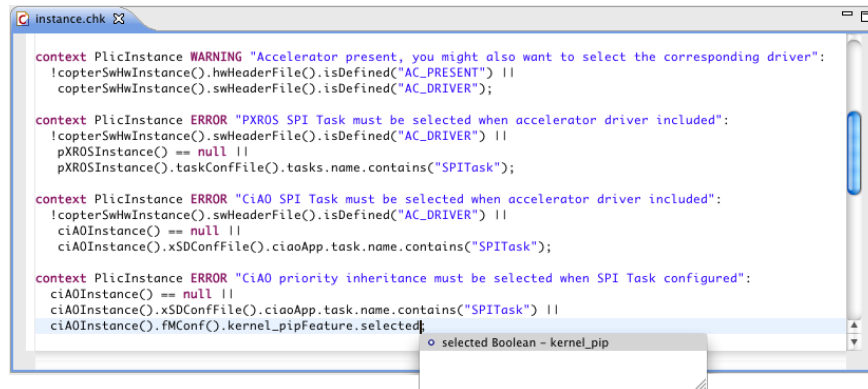
**2. Select or develop converters.** The initially developed converters (cf. Section 4.3) resemble the needs of the I4Copter product line. We use the preprocessor-based generic converter for extracting models from C header files. For converting PXROS' DSL configuration files, we have developed a simple grammar (less than 20 rules) for XText, which we use to derive a corresponding metamodel and a parser for converting a configuration file into a model. For the task configuration file specified in XMLSchema and the pure::variants feature model, we use the corresponding specific converters as well.

**3. Create *PLiCFacade* projects and models.** At this point, the product line engineer creates a new Eclipse PLiCFacade project (or converts the existing project) for each product line. Each project needs to provide exactly one PLiCFacade model file describing the required converter and validator plug-ins, the configuration file types, and possibly the location of corresponding specification files, as well as references to sub-facades.

**4. Define initial constraints.** For constraint definition, the product line engineer chooses (or possibly develops) validator plug-ins for the checking languages to use. For I4Copter, we only make use of the XPand Check language, as the tooling support in Eclipse is far better than for OCL. In particular, the XPand editor has an excellent code completion facility, which can be configured to load all generated metamodels. This eases the definition of constraints considerably. Furthermore, we generate additional helper functions for easy navigation through PLiCInstances and generated models. Thus, we can query, for example, the maximum priority of tasks defined in the CiAO feature model from the top level I4Copter product line configuration using the following string leveraging code completion support: `ciAOInstance().fmConf().kernel.maxTaskPriority`.

Figure 4 shows the XPand Check code necessary to encode the constraints regarding the selection of a sensor using the SPI bus as motivated textually in

Section 2. The example constrains span define files, XML and DSL files, as well as feature model configurations. Note that, for a consistent configuration, each constraint needs to evaluate to **true**. By initially interviewing the I4Copter experts, we were able to find various other obligatory and recommending constraints.



```
context PliCInstance WARNING "Accelerator present, you might also want to select the corresponding driver":
!copterSwhInstance().hwHeaderFile().isDefined("AC_PRESENT") ||
copterSwhInstance().swHeaderFile().isDefined("AC_DRIVER");

context PliCInstance ERROR "PXROS SPI Task must be selected when accelerator driver included":
!copterSwhInstance().swHeaderFile().isDefined("AC_DRIVER") ||
pxROSInstance() == null ||
pxROSInstance().taskConfFile().tasks.name.contains("SPITask");

context PliCInstance ERROR "CIAO SPI Task must be selected when accelerator driver included":
!copterSwhInstance().swHeaderFile().isDefined("AC_DRIVER") ||
ciaoInstance() == null ||
ciaoInstance().xSDConfFile().ciaoApp.task.name.contains("SPITask");

context PliCInstance ERROR "CIAO priority inheritance must be selected when SPI Task configured":
ciaoInstance() == null ||
ciaoInstance().xSDConfFile().ciaoApp.task.name.contains("SPITask") ||
ciaoInstance().fmConf().kernel_pipFeature.selected
```

selected Boolean - kernel\_pip

**Fig. 4.** Using the PLiC framework we specified the configuration dependencies of an acceleration sensor in the XPand Check constraint language.

**5. Configure products using *PLiCInstance* projects and models.** Having defined the PLiCFacade and the initial constraints, the product line configuration engineer can start configuring a product. This works via creating a new PLiCInstance project in Eclipse and filling its PLiCInstance model, which basically contains the locations of configuration files in the file system. The builder component now constantly observes the configuration files, converts them to models on each change and checks the constraints defined in the previous phase in a background process. If constraints evaluate to false, the textual messages associated with them (the error and warning strings in Figure 4) are displayed in the Eclipse problems view, and the configuration engineer gets immediate feedback and advice.

**6. Constantly maintain and improve constraints during evolution.** The more domain knowledge is encoded in formal constraints, the more powerful our approach is. When the product line engineer constantly maintains and improves the set of constraints, and the configuration engineer contributes as well with constraints gathered during configuration creation, our framework can give helpful guidance and considerably shortens time and cost of product configuration.

## 6 Results

Introducing our approach to the I4Copter was little effort. Setting up the PLiC framework and performing an initial workshop for constraint mining and for-

mulation took less than a day. While, during the workshop itself, only few constraints were actually defined, the I4Copter engineers got a feeling for the potential of the approach and delivered several dozens of constraints in pseudo code within the following days. We translated the pseudo code into XPand Check, and—via learning by example—the I4Copter experts rapidly grasped the relevant concepts and formulated the constraints in XPand Check themselves.

The recent introduction, however, impedes giving exact numbers on the achieved improvements. Furthermore, the success of our approach relies on various interdependent factors, such as the complexity of the product line, the quality and number of constraints, and the duration and rate of product derivations, so individual results are hardly transferable. We can, however, give anecdotal evidence that led configuration engineers to the estimation that derivation time (start of configuration to successful deployment) could be reduced by half with the currently defined constraints. Each avoided compiler run due to a triggered constraint saves up to three minutes, software unit testing ten minutes, avoiding software and hardware testing in the testbed saves a test engineer several hours.

Locating the actual source of a configuration error can be even more time-consuming. For example, choosing the SPITask but not the priority inheritance mechanism in CiAO resulted in intermittent errors such as sporadically missing of deadlines and even fatal deadlocks for the quadrotor system. Detecting and tracing back this behavior was a thankless, time-consuming task. Although our constraint infrastructure could not prevent this misconfiguration when it happened for the first time, the hereupon encoded constraint now directly triggers at configuration time and gives helpful advice for correction, so that this configuration error can no longer happen.

## 7 Discussion

In the following, we address several threats to the general applicability of our approach. Possible issues are the problem of semantic loss when converting among metamodeling technologies, the modeling expertise required for our approach, constraint evolution, and the relationship to generative product line approaches.

**Semantic Loss.** In principle, any formal, parsable file can be translated into a model and its formal specification into a metamodel. However, even if a converter is written very carefully, there will usually be semantic loss. UML’s MOF, for example, is far more expressive than Ecore, which we use. But, for the purpose of constraint checking, not all this semantic information is actually needed. One can even intentionally keep a metamodel simple to simplify checking. As mentioned, our metamodel converter for feature models does neither preserve the hierarchy of features nor their dependencies. As the corresponding feature model configuration tool already enforces these dependencies, converting and checking them a second time would be unnecessary complicating and redundant.

**Required Modeling Expertise.** The configuration via PLiCFacade and PLiCInstance models appears straight-forward to us, and, if desired, the same information could also be entered via plain text files or a graphical user inter-

face. So, the only point where modeling expertise is actually needed to a certain extent is for defining constraints. A product line engineer (but not the configuration engineer) has to learn the corresponding constraint language. However, as argued in Section 5, languages such as XPand Check come with mature, content-assisting editors minimizing the necessary learning efforts.

**Constraint Evolution.** If not maintained, the formal constraints defined with our infrastructure will go out of sync with the actual dependencies within the software over time. This is, however, a general problem when defining dependencies, and needs to be ensured for dependencies within features in feature models, as well as for dependencies between `#defines` (such as, `#ifdef DEBUG #define USE_SIMULATED_SENSORS ... #endif`). Thorough development processes including, for example, code reviews and configuration testing, may assist in keeping the set of constraints consistent.

**Generative vs. Constraining Approach.** Using constraints for creating valid configurations does not oppose to using generative technologies. Generating source code or some of the configuration files that the software requires is often essential for efficient product derivation. We see our approach as a complement to the generative strategy in three cases: First, when fine-tuning at various location becomes necessary that cannot be anticipated beforehand, second, when multiple, orthogonal configuration files are needed that cannot be created out of one another, and, third, when introducing a single top-level configuration file that basically mirrors all configuration options on the lower levels does not scale.

## 8 Related Work

Related work can be found both in the field of constraint checking and in large-scale product line configuration.

Considerable research has been conducted with respect to constraint enforcement involving different types of product line models. In [5], OCL constraints ensure that UML models enriched with feature templates result in well-formed models when parameterized with valid feature model configurations. The authors of [12] relate feature models and orthogonal variability models to each other to facilitate automated reasoning on variability. Commonly, existing approaches restrain their focus on few dedicated model types and do not focus on building a general infrastructure. The FAMA tool suite [1] is a notable exception. It also provides an infrastructure for importing models and performing analyses on them. In contrast to our approach, the internally used modeling format is not based on generic metamodeling but on feature modeling. Whereas this considerably limits the types of (configuration) files and models that can be imported, the framework provides means for performing more stringent analyses, for example, for satisfiability via SAT solving. It would be very interesting to integrate validators of this kind also into our infrastructure for checking properties on the corresponding subset of model types.

Currently, we evaluate all constraints each time a file is changed and its model is rebuilt. There is research on efficient algorithms regarding incremental

consistency checking, where only those constraints are reevaluated that actually may be affected by a change [7]. Up to now, we have not run into any performance problems, the builder background process finishes in less than a second for building all metamodels, models, and evaluating the constraints of the I4Copter.

There are several approaches that deal with multiple product lines and their configuration. The Koala approach, which is applied in industry, provides for configuration of large-scale product lines via an architectural description language [14]. Approaches stemming from research, for example, use a combination of class and feature modeling [15] or service-oriented abstraction [18] to configure product lines. One of the main features of our approach is that it is agnostic to the configuration file type used and can check constraints among arbitrary configuration files as long as there exists a converter for this.

## 9 Conclusion and Outlook

With this paper, we have presented an approach and an infrastructure for constraint checking across configuration file types and product line boundaries. We have evaluated our approach using the *I4Copter* product line, which yielded promising results for its applicability in similar complex contexts.

As future work, we consider blurring the boundary between constraint-based and generative strategies by not only checking constraints, but also actually changing configuration values according to values in other configuration files. This, however, requires further analyses and tooling. On the one side, it is necessary to identify the cases where automatic changing of configuration values is reasonable and comprehensible for the configuring engineer, and in which cases it would rather lead to unforeseen side effects. Furthermore, this approach requires converting the changes on model level back to the source configuration files. This back-transformation is far more intricate and we need to analyze the circumstances under which this can successfully be done.

**Acknowledgments.** We thank Martin Hoffmann for providing insights and defining various domain constraints within the I4Copter product line and Wanja Hofer and Christa Schwanninger for their valuable feedback on this paper.

## References

1. Benavides, D., Segura, S., Trinidad, P., Ruiz-Corts, A.: FAMA: Tooling a framework for the automated analysis of feature models. In: 1st Int. W'shop on Variability Modelling of Software-intensive Systems (VAMOS) (2007)
2. Beuche, D.: Variant management with pure::variants. Tech. rep., pure-systems GmbH (2006), <http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf>, visited 2009-03-26
3. Bezin, J.: On the unification power of models. *Software and Systems Modeling* 4(2), 171–188 (May 2005)

4. Czarnecki, K., Eisenecker, U.W.: Generative Programming. Methods, Tools and Applications. AW (May 2000)
5. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness OCL constraints. In: 6th Int. Conf. on Generative Programming and Component Engineering (GPCE '06). pp. 211–220. ACM, New York, NY, USA (2006)
6. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: a case study. *Journal of Systems and Software* 74(2), 173–194 (Jan 2005)
7. Egyed, A.: Scalable consistency checking between diagrams – the ViewIntegra approach. In: 16th IEEE Int. Conf. on Automated Software Engineering (ASE '03). IEEE Control Systems Magazine, Washington, DC, USA (2001)
8. Elsner, C., Lohmann, D., Schröder-Preikschat, W.: Product derivation for solution-driven product line engineering. In: 1st W'shop on Feature-Oriented Software Development (FOSD '09). ACM, New York, NY, USA (2009)
9. Eclipse modeling framework homepage, <http://www.eclipse.org/emf/>, visited 2010-02-22
10. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling: Enabling Full Code Generation. John Wiley & Sons, New Jersey, USA (2008)
11. Lohmann, D., Hofer, W., Schröder-Preikschat, W., Streicher, J., Spinczyk, O.: CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In: 2009 USENIX TC. pp. 215–228. USENIX, Berkeley, CA, USA (Jun 2009)
12. Metzger, A., Heymans, P., Pohl, K., Schobbens, P.Y., Saval, G.: Disambiguating the documentation of variability in software product lines. In: 15th IEEE Int. Conf. on Requirements Engineering (RE'07). pp. 243–253. IEEE Computer Society, Washington, DC, USA (2007)
13. Object Management Group (OMG): Object constraint language, version 2.2. formal/2010-02-01 (February 2010)
14. van Ommering, R.: Building product populations with software components. In: 24th Int. Conf. on Software Engineering (ICSE '02). pp. 255–265. ACM, New York, NY, USA (2002)
15. Rosenmüller, M., Siegmund, N.: Automating the configuration of multi software product lines. In: 4th Int. W'shop on Variability Modelling of Software-intensive Systems (VAMOS) (January 2010)
16. Stahl, T., Völter, M.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons (2006)
17. Stephan, M., Antkiewicz, M.: *Ecore.fmp*: A tool for editing and instantiating class models as feature models. Tech. rep., University of Waterloo, 200 University Avenue West Waterloo, Ontario, Canada (Aug 2008)
18. Trujillo, S., Kästner, C., Apel, S.: Product lines that supply other product lines: A service-oriented approach. In: First Workshop on Service-Oriented Architectures and Product Lines. Special Report. CMU/SEI-2008-SR-006. (September 2007)
19. Ulbrich, P.: The I4Copter project — Research platform for embedded and safety-critical system software, <http://www4.informatik.uni-erlangen.de/Research/I4Copter/>, visited 2010-02-22
20. Völter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. 11th Software Product Line Conf. (SPLC '07) pp. 233–242 (2007)
21. Eclipse XPand homepage, <http://www.eclipse.org/modeling/m2t/?project=xpand>, visited 2010-02-22
22. Eclipse XText homepage, <http://www.eclipse.org/Xtext/>, visited 2010-02-22