An Infrastructure for Composing Build Systems of Software Product Lines

Christoph Elsner
Siemens Corporate Technology, Germany
christoph.elsner.ext@siemens.com

Daniel Lohmann
Friedrich-Alexander University Erlangen-Nuremberg, Germany
lohmann@cs.fau.de

Wolfgang Schröder-Preikschat
Friedrich-Alexander University Erlangen-Nuremberg, Germany
wosch@cs.fau.de

ABSTRACT

Deriving a product from a software product line may require various build tasks, such as model transformations, source code generation, preprocessing, compiling, as well as linking and packaging the compiled sources. Usually implemented using simple scripting languages, such as Apache ant or GNU make, build systems tend to become monolithic entities, which are intricate to adapt and maintain.

This makes developing the build system for a multi–product-line, which is composed of several sub–product-lines and maybe other configurable components, particularly challenging. Several, previously independent build systems—possibly implemented using different build tools (ant, make, etc.)—need to be integrated. In this paper, we approach this by using models to describe the involved build tasks (including their input and output parameters) as well as their composition. An interpreter evaluates the models and executes the tasks in the composed order with the configured parameters to produce the final product.

Our approach enables the interaction of build systems implemented with different tools with only little development effort, whereas the build order and parameter flow is made explicit in the models. We have started to apply our tooling to model the build system of two multi–product-lines, where it reveals sufficient expressiveness and clarifies the build system interaction.

Categories and Subject Descriptors

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

General Terms

Design, Languages

Keywords

Software Product Line, Build System Integration

SPLC '11, August 21-26, 2011, Munich, Germany

1. INTRODUCTION

Building a product in software product line engineering consists of mapping a product configuration to software and possibly further product-related artifacts. The build system of a product line, which implements this mapping, needs to perform various distinct tasks. In case the configuration is based on models, multiple model transformations and code generation from the models might be executed first. Then, aspect weavers or preprocessors may be applied to the sources. Afterwards, the source code needs to be compiled, and be linked or packaged, before it can be deployed on the destination system. Usually, build systems are also used to generate various other artifacts, such as API documentation, product specifications, or the user manual.

Build systems are commonly implemented using build scripting languages and corresponding tools, where Apache ant[1] and GNU make[2] are the most popular representatives. These scripting languages are very powerful: they already provide convenience functions for the most common tasks (e.g., compiling, file copying) and allow integration of arbitrary other tools via extension mechanisms. However, their flexibility comes at a price—in many cases, the build systems designed with such languages are monolithic entities that are very difficult to reuse and compose. Their build tasks often have intricate dependencies among each other, and the available build variability is often hidden in some undocumented variables. This makes it easy to write a piece of code where a single build task cannot be understood unless the build system as a whole is considered.

Composition of build systems is needed when a multi–product-line needs to be assembled. We consider a product line to be a multi–product-line when it is built up out of several other sub–product-lines or further configurable commercial or open-source components. Each of these constituent elements, which we call *p*roduct *li*ne *c*omponents (PLiCs), brings its own means of configuration and its dedicated build system to create its part of the overall product based on configuration input. In previous work [4], we have developed an approach to check the consistency of the *configuration* of a multi–product-line, even if it is spread over various configuration files of heterogeneous types (e.g., feature models, header files, domain-specific configuration languages). In this paper, we present an approach for creating an integrated *build system* out of the build systems of the single PLiCs.

[1]Apache Ant: `http://ant.apache.org/`.
[2]Gnu make: `http://www.gnu.org/software/make`.

To avoid the effort of implementing a new, compound build system completely from scratch, we intend to reuse the already available build systems of the PLiCs as far as possible, even in case they employ different build languages and tools (e.g., make, ant). We intend to keep refactoring minimal, so that the build system of a single PLiC still works stand-alone. Second, based on the experience with our case studies, we assume that the individual build systems need means for interacting with each other. Output from the build task of one PLiC will serve as input for the other, and vice versa. This means that we need a way to specify the build order as well as the parameter flow in a way abstracting from the concrete build tools used.

To parameterize and compose build systems independent of the applied build scripting tool, we propose to represent their coarse-grained structure using models. This way, the single build tasks, their input and output parameters, and the build tasks composition, gain an explicit representation.

We developed an extensible tool infrastructure to model, compose, and execute build tasks for currently three different build script languages (Apache ant, GNU make, MWE workflow language[3].) We have started to apply it to model the compound build systems of two larger-scale product lines (I4Copter, SafeHome). Up to now, our modeling approach exhibits sufficient expressiveness and helps us to understand and visualize the compound build systems by making the previously implicit sequence of build tasks and their dependencies explicit. The required development effort in order to achieve the composition so far has been little. It consists in modeling and possibly small build system refactorings, which still enable each single build system to work without our tooling applied. Once the build system of each single PLiC has a model representation, compound build systems can be created with only little modeling effort.

In the following, we will first argue that parameterizing and composing build systems is crucial when a multi–product-line needs to be assembled, and that composing them using the common build script languages is a tedious and often intricate task (Section 2). Then, we present our approach for product line build system modeling (Section 3). We report on the ongoing appliance of the developed tooling to the SafeHome and I4Copter product lines in Section 4 and discuss the required effort and anticipated benefit of our approach in Section 5. Finally, we address related work and conclude the paper (Sections 6 and 7).

2. INTEGRATION OF BUILD SYSTEMS

In this section, we analyze the need of build system integration and report on the state of the art and its defects.

2.1 The Need of Build System Integration

We have studied the build systems of two multi–product-lines: I4Copter [4] and SafeHome [3]. It turned out that they basically use two variability mechanisms: (build-task-) internal variability, via *parameterization* of the input variables of a build task, and external variability, via *composing* (i.e., "wiring up") basic build tasks to a more complex one. In the following, we will introduce our two case study product lines and provide examples where parameterization and composition are used for build system integration.

[3]Eclipse EMF Modeling Workflow Engine: `http://www.eclipse.org/modeling/emft/?project=mwe`.

2.1.1 I4Copter: Parameterization Example

The *I4Copter* [8] quadrotor helicopter has been designed to resemble embedded real-time systems arising in real-world product line scenarios. Its software is implemented as a product line comprising three sub–product-lines: one for application logic, which also models the interface to the hardware (*CopterSwHw*), the department-internal, aspect-oriented operating system product line *CiAO* [6], and, alternatively, the commercial operating system product line *PXROS*.

To build an I4Copter product, GNU make needs to be called recursively both for CiAO and the CopterSwHw (Figure 1). However, their parameter settings depend on each other. For example, CiAO's parameter `EXTRA_AH_DIRS` is used by the CopterSwHw to feed in extra aspect files (implemented in AspectC++[4]) to be woven into the CiAO operating system. The CopterSwHw, in turn, requires the `OS_LIB` parameter in order to link to it after compilation.

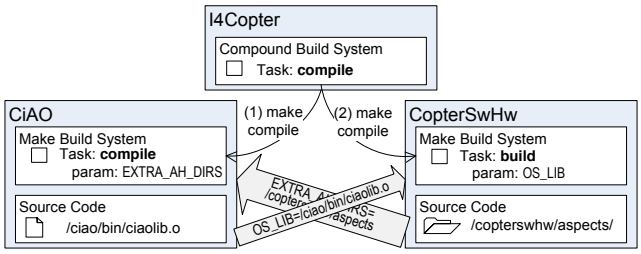

Figure 1: The parameters of CiAO and CopterSwHw depend on each other when compiling an I4Copter product.

2.1.2 SafeHome: Composition Example

Our second example is based on the demonstrator product line SmartHome [9], which has been developed for construction experts such as architects. It has been implemented using the Eclipse-based model-driven framework openArchitectureWare. SmartHome facilitates modeling of buildings and their electrical interior devices and generates the software from the model for automatically controlling these devices.

Originally developed for standard homes, we extended the product line to serve the market for commercial buildings as well. The business model is to offer a standard version of the product line as well as an extended version, which includes advanced safety features, so-called *SafeHome*. We implemented this extension as a separate sub–product-line, called SafetyPLiC, which can be included into the build process if the customer is willing to pay for it.[5]

Figure 2 shows the interaction of build tasks necessary to create a SafeHome product. The SafetyPLiC requires adding further build tasks (model transformations and code generators) at different stages of SmartHome's model-driven product generation, which is based on multiple transformations (*house2comp*, *comp2osgi*, *osgi2code*). As SmartHome and the SafetyPLiC stem from different contexts, they use different build tools (MWE vs. ant). Even if we ignore the parameter flow—which is also necessary in this example, but excluded for brevity—the mere composition order of build

[4]AspectC++ site: `http://www.aspectc.org/`.

[5]See [3] for details of the SafetyPLiC implementation.

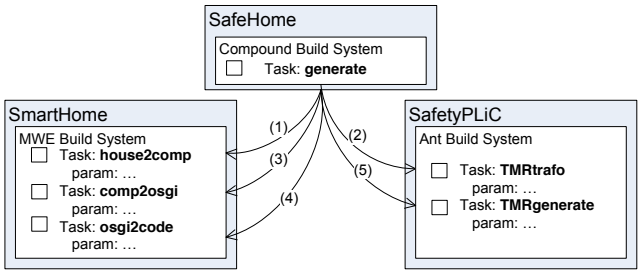

Figure 2: Generating the SafeHome software requires executing the build tasks of SmartHome and SafetyPLiC in a particular order.

tasks is already crucial. If executed in a different order, the build will fail or defective software will be generated.

2.2 Build System Integration State of the Art

In case the PLiCs (sub–product-lines or sub-components) of a multi–product-line use different build tools or follow different build patterns (e.g., recursive vs. non-recursive make [7]), the *full* integration of build systems can be quite an effort, as it often basically means implementing a new compound build system from scratch. Moreover, if the PLiCs are open-source components, or stem from external vendors, a redesign of their build systems would impede updating the component or be impermissible due to licensing reasons.

A common solution, which also goes in line with our industrial experience, is to integrate the build systems in a minimal invasive way. On the level of the multi–product-line, a high-level script (e.g., a simple Makefile, or a shell script) is developed. It invokes the build tasks of sub–product lines by calling the respective build tools (e.g., make, ant) in the correct order with correct parameters in order to build a compound product.

However, to be able to apply this solution, one must already have rightly understood how the subordinated build systems need to be handled. This is not at all a trivial task. As of our knowledge, there is no build language that allows declaring which parameters a build task requires or can optionally deal with. Commonly, all variables in build files (make, ant, oAW, but also CMake, automake, and Maven) have a global scope. Intricate dependencies among build tasks, in the case of Makefiles even with wildcards and implicit build tasks[6] make it hard to understand which build tasks are actually invoked and which variability parameters are available.

As an example for the trickiness of build system parameters, we present the implementation of CiAO's extension point for adding further aspect files into the build process (the parameter `EXTRA_AH_DIRS`, as introduced in Section 2.1). The CopterSwHw product line uses it to add aspect files for custom startup, shutdown, and error hooks.

Listing 1 illustrates the usage of the parameter `EXTRA_AH_DIRS` in the CiAO build system (3000 LOC, 18 files). It can be set from the command line, for example, when executing make, to hold a whitespace-separated list of directory paths. For each path, the variable `AGXXFLAGS`, which contains the aspect compiler flags, is appended with a further option to consider the directory. The flag will then be used each time a file is compiled via the `XX` compiler variable.

Such extension points are quite common in build systems to parameterize and extend them via setting some "magic" variables. However, as the parameters are commonly just as hidden in the code as in the above example, detailed knowledge on the build system is necessary to leverage them.

```
...
AGXXFLAGS += $(foreach p_dir,
                       $(EXTRA_AH_DIRS),
                       -p $(p_dir))
...
XX ?= $(AGXX_PATH)/ag++ $(AGXXFLAGS) ...
...
$(OUT_DIR)/%.o: $(SRC_DIR)/%.cpp ...
        $($XX)) -c $< -o $@-
...
```

Listing 1: The parameters for adapting the build process are well hidden in the implementation of CiAO's make files.

Textual documentation of build files can alleviate this problem. However, sole text cannot provide any further benefits to the involved engineers, for example, for checking (e.g., for parameter existence, or for mandatory parameter settings), visualization, or transparent bridging across build tools. Therefore, according to our experience, build systems tend to be poorly documented—if they are at all. As a result, each time a product line shall to be integrated into a multi–product-line, tedious analysis of its build system may become necessary to parameterize it appropriately.

Summing up, integrating build systems by writing high-level build scripts is a tedious undertaking. It requires analyzing the possibly intricate implementations of build systems implemented in different build tools to find out about their variability parameters. This analysis is required *each time* a certain PLiC needs to be integrated into a new multi–product-line. Moreover, the developed high-level script itself will usually become difficult to understand and inflexible to reuse in other contexts, as each adaptation of the script requires studying the original build system a further time.

3. MODELING BUILD SYSTEMS

In order to reduce the effort for compound build system development, we propose to describe the existing build systems (their crucial build tasks and parameters) using models. As an initial benefit, a model serves as concise and clear documentation of the build system. Much more, each build system model comprises sufficient information so that the creation of a compound build system becomes possible solely by creating a further model—the involved engineers do not need to dig into the build system source code a further time.

The modeled compositions with bound variables serve as input for an interpreter, which then executes the appropriately parameterized sequence of build tasks. Doing so, models become first-class artifacts that make the real structure and interaction of complex build systems explicit. In this section, we present our general approach and the build system modeling language.

[6]See the GNU make manual for a catalogue of implicit build rules: `http://www.gnu.org/software/make/manual/make.html#Catalogue-of-Rules`.

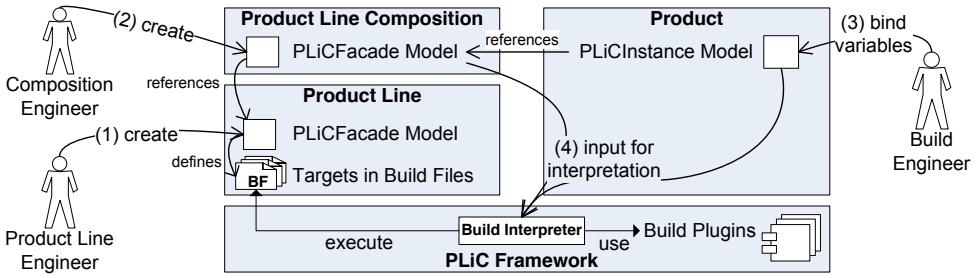

Figure 3: Product line engineer and build engineer using the PLiC framework for product building.

3.1 General Approach

As mentioned, we split up a multi–product-line into so-called product line components (PLiCs) [4], which may be sub–product-lines or other configurable components and which each has its own build system. The intended use of our framework for build system integration involves three roles: *product line engineer*, *composition engineer*, and *build engineer* (cf. also Figure 3).

The *product line engineer* is a specialist for a particular PLiC and specifies its build system in a so-called *PLiCFacade* model (Figure 3, step 1). It comprises all information required by others to build products. The model defines the externally available build tasks, their type (e.g., ant, make) and other attributes (discussed in detail in Section 3.2).

The *composition engineer* (step 2) then creates a further PLiCFacade model to build a composed build system. It is defined solely by wiring up build tasks of other PLiCFacade models. Due to concise and technology independent description on model level, composing becomes easier and more explicit than by composition via ad-hoc scripts.

On product level, the *build engineer* optionally may create a PLiCInstance model that references the respective PLiCFacade model and may bind those variables that have not yet been bound in domain engineering (step 3). For building, the build engineer requests the PLiC framework to execute one of the modeled build tasks (step 4). The interpreter forwards the actual execution to dedicated plug-ins, such that additional build languages can be integrated easily. Currently, we provide builder plug-ins for ant, make, and MWE.

3.2 The Build System Modeling Language

Concrete build systems and their compositions are described in PLiCFacade models, final parameter bindings and, optionally, additional product-specific build tasks in PLiCInstance models. Both models base on the same metamodel (cf. Figure 4). Although not strictly required, the following description of model elements is aligned to their most likely use in steps 1 to 4 in Figure 3.

Step 1: The product line engineer declares the set of externally available build tasks (*BuildTask*), within the *PLiCFacade* model of a product line. A concrete build task element (*ConcreteBT*) maps to an actual build task in a PLiC's build system. Essential characteristics are: the build task type (e.g., ant, make), the build file URI, its target name (e.g., preprocess, compile, ...), and the input and output parameters (*Variable*). As common for build tooling, all parameters are of type string. Parameters can be optional, multi-valued (e.g., using comma or whitespace as string separator) and may have further textual documentation attached. Only those build tasks and parameters that shall be externally available need to be modeled, what helps to keep the models concise.

Step 2: Composition engineers use sequence build tasks (*SequenceBT*) and reference build tasks to compose the build tasks of several other PLiCs (defined in other PLiCFacade models). A sequential build task wires up other build tasks, whereas a build task reference denotes an invocation of another, already defined build tasks. A simple reference (*SimpleReference*), invokes exactly one build task. For convenience, we also devised a more complex mechanism for invoking other build tasks. A *WeaveReference* element receives a list of build tasks names (*weaveOrder*) and a list of sequence build tasks (*tasksToWeave*) as input. When a WeaveReference is executed, the interpreter iterates over the build tasks names in the *weaveOrder* list. Each of the sequential build tasks in the (*tasksToWeave*) list is queried whether a child build tasks with a corresponding name exists. If this is the case, this child build task is executed. In the following Section 4, we will demonstrate the use of this mechanism via an example from our SafeHome case study.

The input parameters declared by concrete build tasks are bound using *VariableBinder* elements. They can be applied directly to a concrete build task, or to sequences and references that ultimately lead to one. This enables all involved stakeholders to bind variables: product line engineer, composition engineer, and build engineer.

Step 3: In the PLiCInstance model, the build engineer can bind those variables that have not yet been bound previously in PLiCFacade models. Even further build tasks may be added (e.g., for compiling additional, product-specific code).

Step 4: Finally, the build engineer can invoke the interpreter for a modeled build task. In case of a sequential build task (*SequenceBT*), its children are executed. For a *SimpleReference*, the referenced build task is executed. A *WeaveReference* mixes build task execution as described previously. For a *ConcreteBTs* model element, the model interpreter passes the build file and target to execute, together with the parameters, to the appropriate builder plug-in. The builder plug-in then executes the build task setting the parameter values to the specified values and returns the output parameters if available.[7]

[7]The implemented builder plug-ins for ant, make, and MWE all support output parameters. Therefore, we read out the values of those Makefile variables (or ant/MWE properties) that are defined in the model as output parameters, immediately after the build task has been executed.

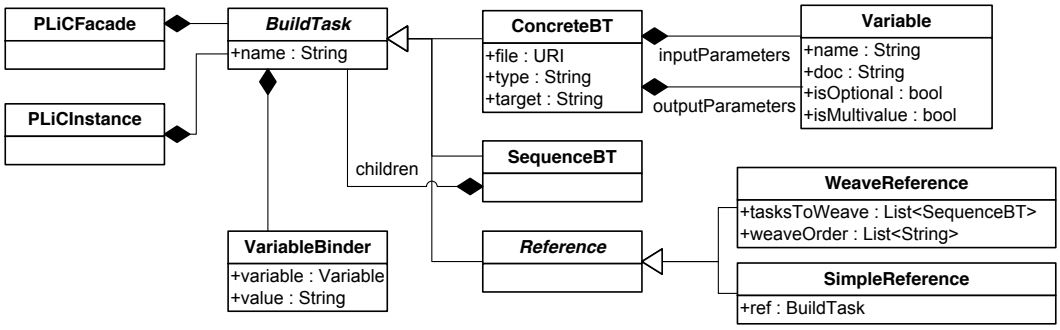

Figure 4: PLiCFacades model build systems in domain engineering, PLiCInstances in application engineering.

4. REVISITING THE EXAMPLES

In this section we revisit the two example product lines from Section 2.1. We first model the build systems of basic PLiCs in PLiCFacade models (step 1). These models serve as an executable description of all externally available build tasks and parameters. Doing so, the models comprise all information required to integrate several build systems, such that larger-scale build systems can be created solely by using modeling (step 2). Finally, PLiCInstance modeling and build system execution (steps 3 and 4) are addressed as well.

4.1 Parameterization: I4Copter

In Section 2 we introduced the extension point `EXTRA_AH_DIRS` of the CiAO sub–product-line, as well as its implementation. It allows for adding further aspect files into CiAO's build process. Whereas its extensibility was previously hidden in the Makefiles, we are now able to expose it via corresponding models.

Both CiAO and the CopterSwHw sub–product-line now provide a PLiCFacade model (cf. Figure 5). The models describe their concrete build tasks (*ciaocompile*, *copterswhwcompile*) with their respective type, build file, and target. Furthermore, they declare their respective input parameters *extraAspectDirs* and *osLib*. The I4Copter PLiCFacade, which is the model of the multi–product-line composing the other two product lines, uses simple build task references to model the composition. *VariableBinder* elements are used to bind the parameters to the concrete values ("/copterswhw/aspects/" and "/ciao/bin/ciaolib.o").

In application engineering, the build engineer creates the *i4product* PLiCInstance and uses variable binding to add a further product-specific aspect directory into the build process, as the *extraAspectDirs* variable is multi valued. Figure 6 shows the graphical user interface generated for the build task *i4coptercompile*, where the build engineer can again edit the parameters and execute the build.

4.2 Composition: SafeHome

As mentioned in Section 2.1, the model-driven product generation process of the SmartHome product line needs to be extended with safety-related model transformations and code generators (SafetyPLiC). We model SmartHome and the SafetyPLiC in dedicated PLiCFacade models (see Figure 7). The SafeHome PLiCFacade, finally, composes them.

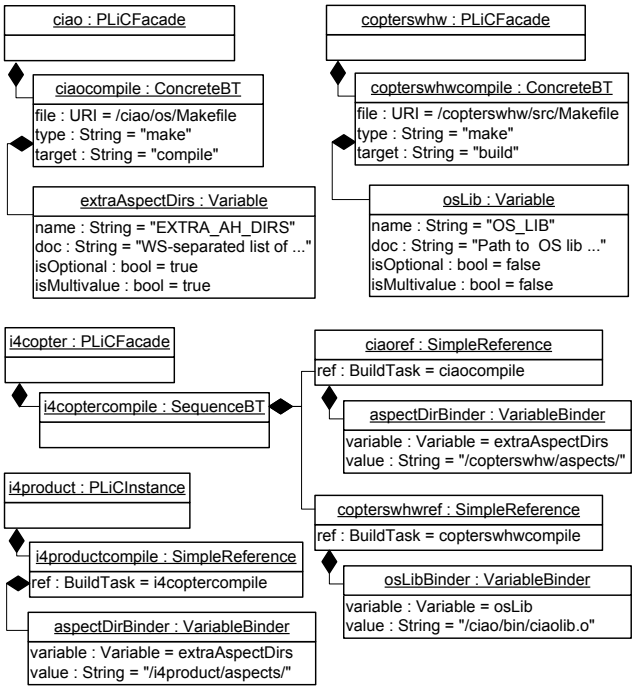

Figure 5: I4Copter uses SimpleReference and VariableBinder elements to compose the build tasks of the CiAO and the CopterSwHw PLiC.

The SmartHome PLiCFacade model in Figure 7 describes the basic workflow for model-driven generation, which is based on the MWE build system. The house model is first transformed to a generic component model (*house2comp*), then to an OSGi-specific component model (*comp2osgi*), from which, finally, code is generated (*osgi2code*). For brevity, the input and output variables of tasks are omitted in the figure. The SafetyPLiC PLiCFacade model provides an additional model transformation (*comp2comp*) and a further code generator (*osgi2code*), based on Apache ant. In the SafeHome PLiCFacade model, the two other models are composed. This is achieved via a WeaveReference element. It receives the two sequence build tasks of SmartHome and the SafetyPLiC as an input (*tasksToWeave*). Based on the order of task names given in the *weaveOrder* list, the two

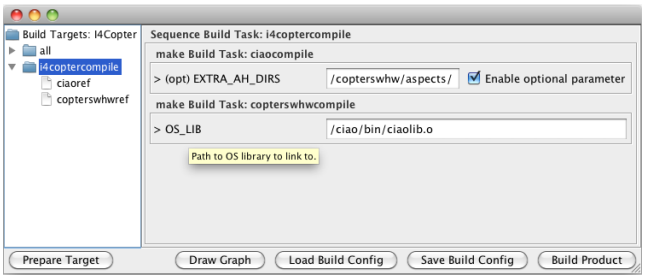

Figure 6: In the generated GUI the build engineer can review and alter the build parameters and can trigger product building.

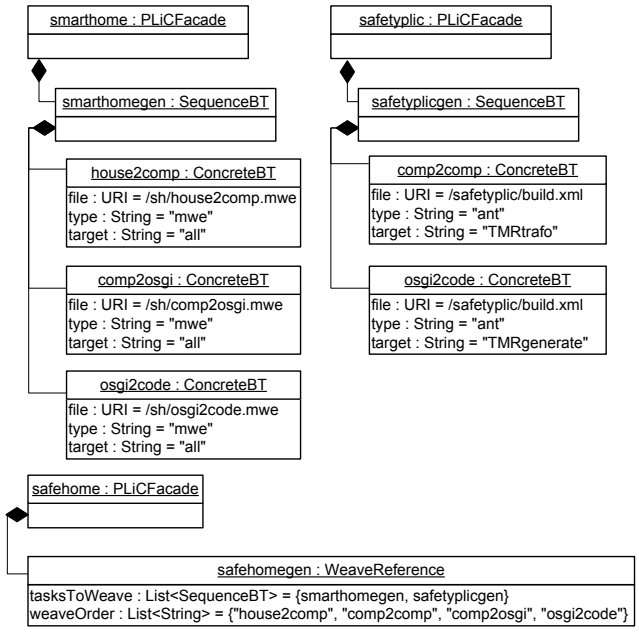

Figure 7: SafeHome uses a WeaveReference element to compose the build tasks of the SmartHome and the SafetyPLiC in the correct order.

sequences will be interwoven during interpretation.[8]

In application engineering, the SafeHome PLiCFacade model is interpreted. Figure 8 shows this result in an automatically generated diagram, which serves for better reasoning about the interaction of build tasks.[9] The diagram also shows the handling of output parameters. Their values can be accessed via `${<outputParameterName>}` in order to feed them as input parameters into subsequent build tasks. String operations, which can be implemented in plain Java (e.g., `String ${Util.rmSuffix(String)}`), are supported as well.

[8]In case sequences contain equally-named build tasks (e.g., *osgi2code* for SmartHome and the SafetyPLiC) the order of sequences in the *tasksToWeave* list decides which is executed first.

[9]For generating the diagrams, we use the software Graphviz: `http://www.graphviz.org`.

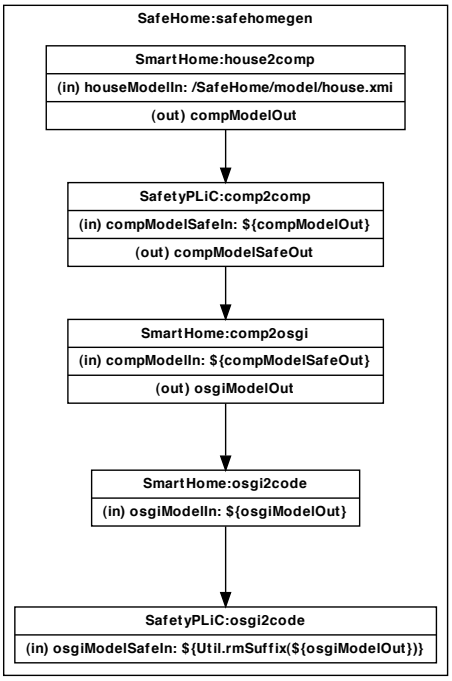

Figure 8: The automatically generated diagrams help to understand the interaction of build tasks and their parameter flow.

5. DISCUSSION

We have currently just started to apply our approach to the two product lines. After interviewing the I4Copter developers, we expect that around 20 build tasks will need to be modeled, ranging from static WCET code analysis over sandbox tests for the different components to flashing and testing on the physical device. In the SafeHome case study, we also plan to add very early build tasks of product derivation, such as generating offer documents and presenting a simulation of the house to the customer (approx. 25 build tasks in total). As the evaluation has not been completed yet, we will subsequently discuss the *anticipated benefits* and trade them off for the *efforts required* to introduce and maintain our approach. Furthermore, when we describe the required efforts, we will report on early experiences gathered and discuss the expressiveness of our modeling approach.

5.1 Anticipated Benefits

The first assumption we wish to prove is that our approach will *ease development of composed build systems*, in particular that build systems can be composed faster and with less expert knowledge about internal structures. This benefit in particular pays off when a PLiC is reused multiple times: Its PLiCFacade model must be created only once; then, composition engineers can integrate it into various compound product line build systems with less effort. Second, we hope to prove that our approach will *speed up product derivation* of multi–product-lines as well. Our composition models allow to bind and to adapt the parameters exposed on model level both in domain (PLiCFacade models for composition) and in application engineering (PLiCInstance models), for which we provide a unified interface for parameter editing

and execution of compound build tasks independent of the build tools involved. Further benefits that might be worth evaluating is whether the quality of build systems, build system compositions, and of derived products may be improved with using our approach for explicit build system modeling.

5.2 Required Effort and Discussion

The potential benefits of the approach need to be traded off against the effort required to set up and maintain it. Fully-fledged introduction of our approach requires (R1) analyzing the involved build systems and their variability, possibly, (R2) build system refactorings to expose single build tasks for external invocation, (R3) modeling the build systems via PLiCFacade models, and (R4) co-evolution of build system implementation and models.

Up to now, we could not identify any show stoppers and the required effort appears manageable. In the following, we will address the experiences we already have gathered regarding R1-R4 for our two case studies.

5.2.1 Analyzing Build Systems

Up to now, we can say that the effort for analyzing the build system (R1), which means mining the actually available variation points from its implementation, differs widely among the two studied multi–product-lines. For SmartHome, the effort was rather small. Its MWE-based code generator only comprised 700 lines of sequentially executed code. The CiAO operating system, however, together with the I4Copter, comprised around 3000 lines of hand-crafted build code spread over 18 Makefiles with subtle dependencies. It makes analyzing the actually available variation points an intricate, time-consuming task. This is, however, not a principle drawback of our approach. After having modeled the build tasks, users of the build system, and also composers of multi–product-lines, will be able to avoid digging into its source code for parameterizing a build. Therefore, the one time effort for the analysis is very likely to pay off over time.

5.2.2 Refactoring

The required effort for refactoring (R2) also differs. The CiAO and CopterSwHw product line, on the one hand, use Makefiles and are therefore target-based (generate, compile, test, clean, etc.). So, there already are separate build tasks that we can invoke externally and no refactoring is required. Note that, even if there are dependencies defined in a Makefile (e.g., "compile" prerequisites "generate"), the preconditioned target will usually not be executed a further time when it has been invoked manually before. The optimization efforts of make are therefore not hampered when a product line engineer decides to expose both the "generate" and the "compile" target in a PLiCFacade model.

SmartHome, on the other hand, uses the MWE workflow language, which does not know the concept of "targets". All build commands in a workflow file are simply executed sequentially. As the workflow of SmartHome was previously basically only within a single file, we had to refactor it. We factored out the different transformations into separate workflow files (house2comp.mwe, comp2osgi.mwe, osgi2.code.mwe, as shown in Figure 7). This way, we were able to weave in the safety transformations at the appropriate locations. The required refactoring effort was however manageable: it can be compared with factoring out code into dedicated methods in object-oriented programming.

5.2.3 Expressiveness of Our Modeling Approach

Up to now, our build modeling language, as shown in Figure 4, exhibited sufficient expressiveness (R3). As we only intend to model the coarse-grained build order and their parameterization, we have tried to keep the modeling constructs to a minimum. However, a remarking difference to common build tools is that our modeling approach only allows us to specify the build order as an explicit sequence. Make, ant, and other build tools, in contrast, allow declaring *build prerequisites* for each task, from which the build order is calculated implicitly. In our current case studies, we did not miss such a mechanism, in particular because it would scatter information about the build order over various model elements in possibly several PLiCFacade models and grasping the build order would become more challenging.

Furthermore, even if we added the capability of dependency modeling, we could not make use of its premier benefit: building up a dependency graph and exclude those tasks from executing that cannot produce a new result. This is due to the fact that dependencies on model level will not (!) be complete, as we do not intend to remodel *all* targets and *all* their dependencies (including, e.g., all single C files of the project). Therefore, we would nevertheless be forced to execute all involved build tasks.

Finally, we have experienced in both industrial and open-source build systems that the build prerequisite mechanism is not used comprehensively, which results in flawed builds. Over time, the build system prerequisites for some files get out of sync with the real dependencies in the code. At some rare occasions, the necessary recompilation then simply does not take place because it is "optimized away". As such defects are quite difficult to discover and debug, our experience is that build engineers often plainly delete all previously produced files (e.g., via executing "make clean") right before building—just to prevent the (incomplete) optimization from being performed. While this observation has motivated our current concept of explicit build sequences, we still plan to explore whether a more carefully designed build prerequisite approach can improve compound build system design—while keeping the associated complexity increase manageable.

5.2.4 Co-Evolution

When the build system evolves (R4), the corresponding PLiCFacade model needs to be adapted accordingly. As we only expect a subset of all targets and parameters to be modeled—those that shall be externally available—, we consider the effort for this to be manageable. At best, the developer of a PLiC's build system is also made responsible for maintaining the corresponding model.

Whereas our approach is on par with the state of the art (cf. Section 2.2) by providing a documentation of the build system, it also denotes an executable specification of its "public interface". Therefore, certain inconsistencies (e.g., deprecated parameters), become easier to detect. For example, renaming a build task or parameter will directly result in an error when trying to execute the PLiCFacade model the next time. Automated tools might as well be considered to help detecting a drift between models and build system implementation. As these tools would need to be implemented per build language—and would rather have limited capabilities, as they could not detect which tasks and parameters were *intentionally* left out in the PLiCFacade model—we do not provide support for this at the moment.

6. RELATED WORK

Related work can be found in the realm of build tools as well as within research on (product line) build system development, maintenance, and composition.

Next to make and ant, there are various other build tools on the market that promise to ease build system development. Prominent tools we like to consider exemplarily are CMake, automake, and Maven. Those tools all have in common that they have a strong declarative part, where it is possible to specify system prerequisites of the build system, the sets of files to compile, tests to be run, etc. Their common strength lies in the operating-system–independent building of products and in defining a standardized way to do common tasks such as compiling, linking, or packaging. But how do they deal with the input and output parameters of their build tasks, and how do they compose complex build tasks? Basically, automake and Maven fall back to the build languages we already have discussed: Custom tasks can be specified by defining make and ant targets respectively. CMake uses its own build language, which it can map to various other build languages (e.g., make, Visual Studio, Eclipse). However, it handles build task parameters and composition very similar to make and ant, via globally-scoped variables and via declaring dependencies to other build tasks. Summarizing, also those build tools would both support and profit from our build task modeling approach, what encourages us to develop our approach further.

Our approach achieves composition of build tasks by explicitly specifying the elements to compose (using Simple- and WeaveReferences, cf. Figure 4). In [1], Adams presents MAKAO, an aspect-oriented tool framework for make files. The author's primary intention for using aspect-orientation is to reason about and to refactor Makefiles. In our case, the most apparent use for aspect orientation would be for composing build tasks. However, aspects are commonly oblivious [5], what makes it rather difficult to maintain the overview which aspects affect which build tasks in which way. Therefore, as long as we identify no striking need, we well stick to our referencing mechanisms, in order to make the interaction and parameter flow of build tasks explicit.

In [2], de Jonge presents *build-level components* to systematically reuse code artifacts together with their build systems. In the notion of the author, a build-level component consist of one or more directories containing source files, together with the logic for building a product from the source files. This notion is quite similar to what we call product line components. The author, however, proposes to use one dedicated build tool (the "autotools" suite), which provides a standardized build interface (the targets: all, clean, install, check, ...) for deriving products. Doing so, the author does neither provide a concept how to integrate different kinds of build tools, nor does he address how to flexibly compose and parameterize elementary build tasks from different build-level components to more complex ones.

7. SUMMARY & OUTLOOK

In this paper, we have presented an approach and corresponding tooling for build system modeling and composition. It improves on the state of the art, which basically uses ad-hoc scripting and informal textual documentation, and which often may require understanding possibly complex build system code for build system composition and execution. We approach this problem by documenting the elementary build tasks, their parameters, and their compositions using a model-based approach. Hereby, the models are formal and complete enough to support the composition and execution of build tasks using solely models as well, without the need for digging into the source code a further time for this purpose.

We have implemented a framework that interprets the models and executes the composed build tasks, thereby abstracting from the fact that the build tasks are implemented with different build tools; currently, we support Apache ant, GNU make, and the MWE workflow engine. We have started to evaluate our tooling by applying it to two multi–product-lines. Early results indicate only moderate introduction effort and sufficient expressiveness of our modeling language, thereby making build task composition and parameter flow explicit, which was previously hidden in the implementations of respective build systems. The full application and evaluation of our approach on our two case study product lines will provide us with more insights about the conditions under which our approach may successfully be applied.

8. REFERENCES

[1] B. Adams, K. De Schutter, H. Tromp, and W. D. Meuter. Design recovery and maintenance of build systems. In *23st IEEE Int. Conf. on Software Maintainance (ICSM'07)*, pages 114–123, Washington, DC, USA, October 2007. IEEE.

[2] M. de Jonge. Build-level components. *IEEE TOSE*, 3(7):588–600, July 2005.

[3] C. Elsner, C. Schwanninger, W. Schröder-Preikschat, and D. Lohmann. Multi-level product line customization. In K. Sugawara, editor, *2010 Conf. on New Trends in Software Methodologies, Tools and Techniques (SoMeT '10)*, Frontiers in Artificial Intelligence and Applications. IOS Press, 2010. 37–58.

[4] C. Elsner, P. Ulbrich, D. Lohmann, and W. Schröder-Preikschat. Consistent product line configuration across file type and product line boundaries. In K. Kang, editor, *14th Software Product Line Conf. (SPLC '10)*, volume 6287 of *LNCS*, pages 181–195. Springer, Sept. 2010.

[5] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *W'shop on Advanced SoC (OOPSLA '00)*, Oct. 2000.

[6] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *2009 USENIX ATC*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX.

[7] P. Miller. Recursive make considered harmful. *AUUGN Journal of AUUG Inc.*, 19(1):14–25, 1998.

[8] P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat. I4Copter: An adaptable and modular quadrotor platform. In *Proceedings of the 26th ACM Symposium on Applied Computing (SAC '11)*, pages 380–396, New York, NY, USA, 2011. ACM.

[9] 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.