Verification of Variable Software: An Experience Report *

Richard Bubel, Crystal Din and Reiner Hähnle

Department of Computer Science and Engineering
Chalmers University of Technology
bubel@chalmers.se, crystal@student.chalmers.se, reiner@chalmers.se

Abstract. We report on our experiences with formal specification and verification of variable and customizable software realized in a software product family architecture using the Java Modeling Language (JML) and the KeY verification system. Software product families can be adapted to different deployment scenarios and provide instantiable feature sets as requested by the customer. Along a small case study we explore how to generate JML specifications for/from a given feature configuration and report on verification attempts of selected methods of the derived product. We identify challenges that need to be solved to allow scalable specification and verification of variable software.

1 Introduction

One of the biggest saving potentials for increasing the efficiency of software development lies in the reusability of software artefacts. In order to make software artefacts reusable, two essential qualities must be achieved: flexibility and abstraction. The first is needed, because reusable software is supposed to work in a variety of different contexts and requirements. The second is important to achieve a separation between the level of design and that of executable products. There is a large number of suggestions on how to achieve reusability. Among the most systematic approaches are model-driven engineering (MDE) and software product families (SWPF).[1] Of these, software product families are arguably the more successful method in practice and are very widely used in industry.[2]

The core idea of software product families is to split software development into two separate streams called *Family Engineering* and *Application Engineering*, see Fig 1. In the former, the commonalities of all anticipated products are specified in a structured manner centered around the notion of a *feature*. The resulting interfaces, libraries, and partial implementations are collected in an *Artifact Base*. Concrete products are obtained by feature selection and feature instantiation.

* This work has been supported by the EU project FP7-ICT-2007-3 HATS *Highly Adaptable and Trustworthy Software using Formal Methods*.

[1] Both terms "Software Product Families" and "Software Product Lines" are in use and can be considered to be equivalent within the scope of the present paper.

[2] See the Software Product Line Hall of Fame at http://www.splc.net/fame.html.

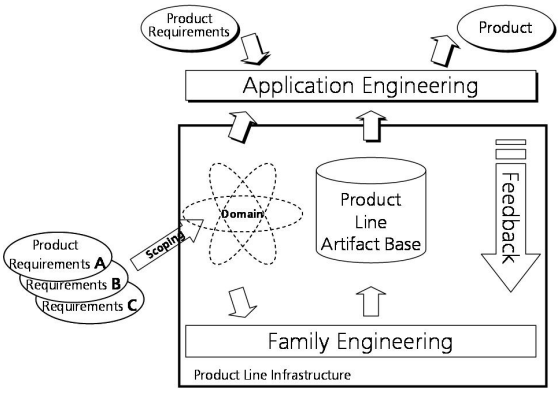

Fig. 1. Sketch of life cycle in Software Product Family development

Software product family-based development is increasingly used in safety-critical applications, for example, in health care products or in automotive software. In addition, the designs of product families reach complexity limitations, because different features may interact in unanticipated ways. It is fairly standard to automatically check the compatibility of features [5], but this is done with structural descriptions, not based on a precise behavioral model of feature functionality. For these reasons it is highly interesting to investigate formal verification of functional properties in software product families. To the best of our knowledge this has not been attempted before. The present paper is a first case study where we seek to verify certain functional properties of a small product family. We report on our experiences, discuss different design choices, and list a number of encountered problems. We also state a number of requirements for the design of verification methods and tools to scale up to industrial-size software with high variability.

It is clear start that—unless verification and specification is compositional and incremental—full functional verification at the family engineering level is doomed to fail, because of the targeted variability. Already small product families give rise to an infeasibly large number of products with different properties. Suitably compositional and incremental verification methods are the subject of future research, therefore, in our case study we aimed at verification at the level of a single derived product in the implementation language Java. At first sight this seems to be merely a standard verification problem. Depending on the implementation of feature selection, however, it becomes much harder: the reason for this is that we chose an implementation that resolves variability points only at run-time, not statically at compile-time. The reason for this choice is that it allows for a more flexible architecture and is, therefore, favored in practice.

The case study in our paper has been done with the verification system KeY [4]. The software product family was implemented in Java and we used the Java Modeling Language (JML) to specify properties.

The paper is organized as follows: in Sect. 2 we describe our case study and provide some background on feature modeling. In Sect. 3 we present the Java implementation of our case study. The formal specification of properties for our case study is explained in Sect. 4, specifically, how we translate FDL into JML. The results of the verification experiments are presented in Sect. 5. We discuss related and future work in Sect. 6.

2 Background

2.1 The Common Component Modeling Example

The Common Component Modeling Example (CoCoMe) [11] is an academic case study and has been widely used as a benchmark for the evaluation of modeling formalisms in the context of software product families (SWPF).

The CoCoMe scenario describes a trading system as it may be used in a supermarket. The basic components of the trading system are cash desks (see Fig. 2) and a store server. A cash desk is responsible to register the products a customer is going to buy. Each product is uniquely identified by a product identification number (productID). During product registration, the cash desk queries the store server for the product name and price tag associated to the entered productID. After all productIDs of the customer's purchase are registered, the customer is accounted for the purchase. Finally, if the payment transaction is successful the store server records the purchase and updates its inventory list accordingly.

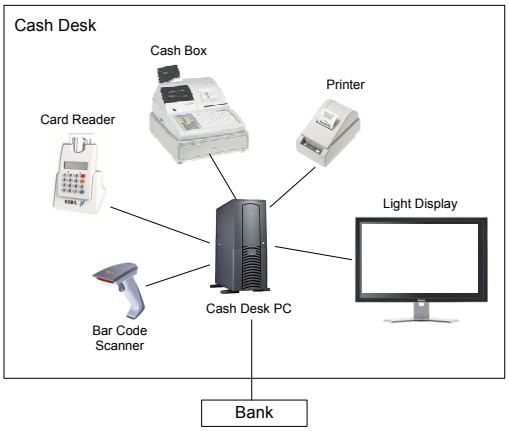

Fig. 2. The hardware components of a single cash desk. Image taken from [11].

The CoCoMe challenge comprises not only to model a cash desk system that is able to handle the above scenario, but the modeled trade system should also

be adaptable to different environments. For instance, shops may have barcodes encoding the product id and want to be able to read them automatically using a scanner rather than having to enter them manually. Businesses may accept only cash or card or both payment kinds. In case of card payments the supported type of cards (prepaid card or credit card) should also be customizable.

2.2 Feature Modeling

In this section we present the specific feature model used for the case study and explain the necessary elements of the feature modeling language. As a basis for our case study we used the feature model in [12] which we also took as a starting point for our implementation.

Different modeling formalisms have been developed to capture the requirements as sketched in Sect. 2.1 in a structured manner. Best known are perhaps decision diagrams [3] and feature-based models [8, 1]. For our case study we use the feature modeling approach first introduced in Feature Oriented Domain Analysis (FODA) [8] and extended in subsequent work.

A software family can be seen as a set of features, while a concrete software product is then derived by selecting a subset of the features; such a feature selection is called a *feature configuration*. Not each combination of features represents a valid feature configuration, as for example, certain features may require the presence or absence of others. Feature diagrams and a feature description language (FDL) provide structured means to describe valid feature configurations. We restrict ourselves to tree like structures for representing valid feature configurations. The feature diagram representing all valid configurations of the feature CashDesk is shown in Fig. 3.

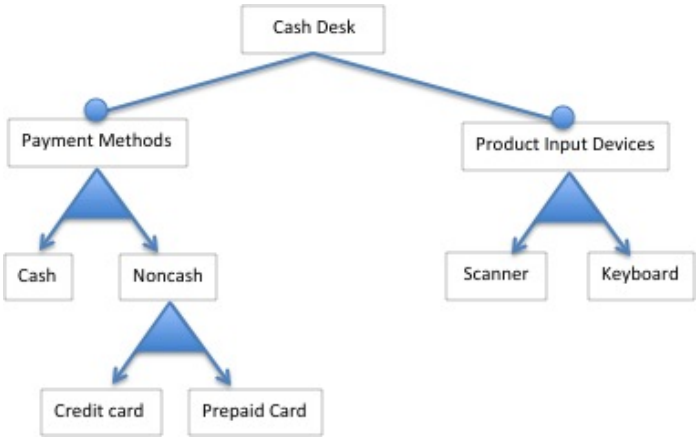

Fig. 3. Feature diagram of the CoCoME cash desk feature

The root of a feature diagram represents the top-level feature whose valid configurations are modeled (here, the cash desk component). A feature can then be composed of subfeatures represented as children of the root node (e.g., the Cash Desk has two subfeatures, namely Payment Method and Product Input Devices). There are different types of edges (see Fig. 4) that can be used to connect the children to its parent. Depending on the type of edge certain restrictions apply. An edge with a filled circle at the end represents a mandatory feature, i.e., the feature *must* be selected when its parent is selected, while an empty circle represents an optional feature. To express that at least one of a group of sibling features has to be selected, the edges to these siblings are connected by a filled triangle. An empty triangle means that *exactly* one of the grouped siblings has to be selected, but not more.

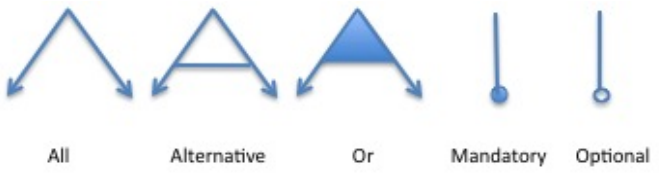

Fig. 4. Feature Diagram Notations: All: all subfeatures must be selected; Alternative: exactly one subfeature must be selected; Or: at least one subfeature must be selected; Mandatory: required feature; Optional: optional feature

In our case a valid configuration of a CashDesk must include the direct subfeatures Payment Methods and Product Input Devices. The feature Payment Methods requires at least one of the features Cash or Noncash to be present.

The most basic product that can be derived from a feature configuration that is valid under the model given in Fig. 3 is the one that allows only keyboards as product input devices and accepts only cash payment.

Alternative to the graphical notation, equivalent textual notations can be used to encode valid feature configurations. We presented only those notions required for the understanding of the paper: there exist several others that allow to express further dependencies and restrictions of features.

3 Implementation

In this section we describe the Java implementation of the cash desk component. We explain how the variability of the cash desk component is achieved so that for all feature configurations described in Fig. 3 a corresponding product can be derived.

The implementation follows closely the feature diagram shown in Fig. 3. For each node there is a similarly named interface or class that represents or implements the feature. The class `CashDesk` shown in Fig. 5 implements the

behavior common to all possible cash desk configurations. A cash desk can be equipped with an arbitrary number of input devices and payment processes. It provides, therefore, methods to add input devices `addInputDevice(IDevices)` and payment methods `addPaymentMethod(IPayments)`.

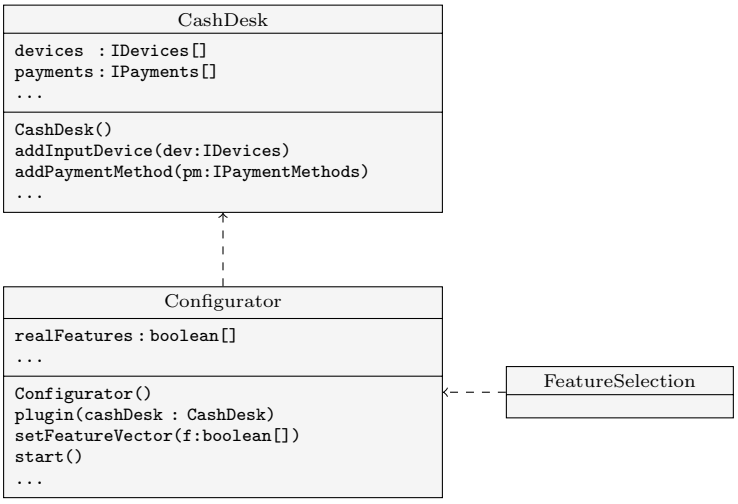

Fig. 5. Class diagram of `CashDesk` controller

Each input device has to implement the interface `IDevices`. The interface `IDevices` defines the protocol for entering product identification numbers by declaring a common set of methods initiating and finalising the product input process. In our scenario, the supported input devices are keyboards (class `KeyboardProductInput`) and barcode scanners (class `ScannerDevice`) as shown in Fig. 6. Supported payment methods need to implement the `IPayments` interface which defines the common protocol for financial transactions. It provides the `CashDesk` class to implement billing of the customer in a transparent way with respect to the underlying low-level payment protocol.

Our implementation of variability points is substantially different to the CoCoMe implementation in [12] and is an almost complete rewrite of it except for the graphical user interface. In principle, our implementation admits to change the feature configuration of an already deployed system at run-time. This means that resolution of variability points happens dynamically rather than statically. In our case study, however, dynamic variability point resolution is not exploited, but restricted to simulate static resolution. Thus, once the system has been setup, its configuration is considered to be fixed. The dynamic evolution of features after system initialisation are beyond the scope of this paper and subject of future work.

We explain now how feature selection and the initialisation of the cash desk system are implemented. At start of the configuration phase the user is asked to

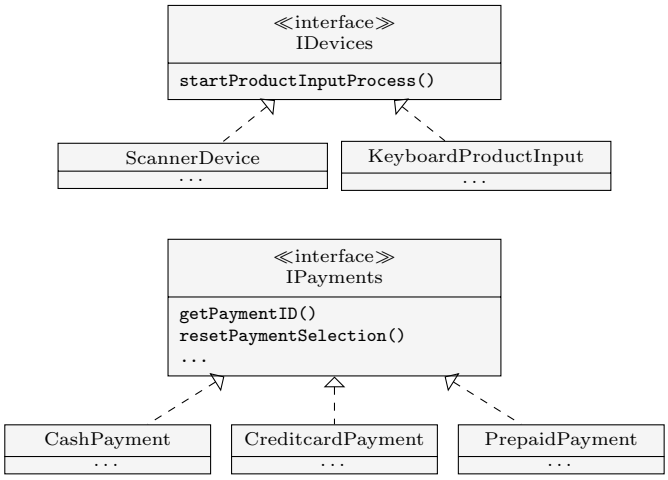

Fig. 6. The feature hierarchy as implemented in Java

customize the system by selecting a feature combination with help of a graphical user interface (see Fig. 7). When the user finished feature selection the chosen

Fig. 7. Feature Selection Interface

configuration is passed to an instance of the `Configurator` class which is responsible for the cash desk system deployment phase (see Fig. 5).

The feature configuration is passed as a bitvector (represented as boolean array) to the method `setFeatureVector(boolean[] f)`. The encoding of feature configurations as bitvectors is canonical: the length of the vector is the same as the number of available features and each bitvector element represents exactly one feature (feature f_i is selected iff `f[i]==true`). If the selected feature configuration is invalid, then the configuration phase is aborted and an exception of type `FeatureException` thrown. Otherwise the feature array is assigned to the field `realFeatures`. Subsequent invocation of the `start()` method triggers creation and initialisation of the cash desk system.

First an instance of the class `CashDesk` is created. Then the `plugIn()` method of the `Configurator` is called which equips the created `CashDesk` instance with

the chosen features and accessories like keyboards or scanners by creating the respective instances and registering them at the `CashDesk` instance. The presence of this plug-in mechanism makes dynamic feature selection principally possible.

4 Specification

Feature model diagrams provide a high-level, structural specification of valid feature configurations, but do not relate to a concrete implementation, that is, the actual behavior of a software product family. As we want to verify that the Java implementation described in Sect. 3 permits only valid configurations to be deployed, we need to connect the feature model specification and the actual Java implementation.

4.1 The Java Modeling Language

For Java programs the Java Modeling Language (JML) [9] is widely used as specification language. JML follows the design-by-contract paradigm and is supported by numerous tools like the Java verification system KeY [4] used here.

JML specifications are added as comments to Java source code. They start either with `//@ ....` or are enclosed in `/*@ ... @*/`. Among other things, JML allows to specify invariants

```
//@ public invariant bExp;
```

method contracts for the normal behavior case

```
/*@ public normal_behavior
  @ requires ⟨bExp_req⟩;
  @ ensures ⟨bExp_ens⟩;
  @ assignable ⟨store_ref_list⟩;
  @*/
```

and for the exceptional behavior case

```
/*@ public exceptional_behavior
  @ requires ⟨bExp_req⟩;
  @ signals (Exception e) ⟨bExp_ens(e)⟩;
  @ assignable ⟨store_ref_list⟩;
  @*/
```

where

- the **requires**/**ensures** keywords followed by a boolean JML expression $\langle bExp_{req|ens} \rangle$ represent the method's pre-/postconditions
- the **assignable** keyword followed by a list of store references (fields, array components) specifies the locations that might be at most changed by the method
- the **signals** keyword specifying the postcondition in case that an exception of the indicated type has been thrown.

JML expressions are a superset of Java expressions with a number of additional operators including

- the boolean operator `==>` denoting logical implication
- universal and existential quantifiers

  `(\forall` T i; $\langle bExp(i)_{guard}\rangle$; $\langle bExp(i)\rangle$`);`
  `(\exists` T i; $\langle bExp(i)_{guard}\rangle$; $\langle bExp(i)\rangle$`);`

  where the second semicolon means implication in the universal case and conjunction in the existential case.

Finally, we mention ghost and model fields, declared similar to standard Java fields. A ghost field declaration such as `//@ public ghost int i = 5;` declares an integer typed field named `i` and initialises it with the value `5`. Ghost fields have nearly the same meaning as standard fields and can be assigned values within method body statements using the JML **`set`**-primitive `//@ set i = 10;`.

Model fields can be referred to like standard fields in JML specifications, but it is not possible to assign them a value directly as is the case for ghost fields. Typically, they are used in interfaces where they are related to an (abstract) datatype and used to specify interface methods in terms of model fields and the operations its type provides. Implementing classes of the interface express then how their implementation relates to the model field by providing a **`represents`** clause mapping their internals to the model field. For more details see [9].

4.2 JML Representation of the Feature Model

We describe how a feature model is translated into an equivalent JML specification. The obtained JML specification will be self-contained and independent of a concrete implementation. Our translation of feature models into JML follows the approach presented in [7] for propositional logic.

Let FM denote the feature model to be translated and $\mathcal{F} = \{f_0, \ldots, f_n\}$ the set of all its features. The translation $tr(FM)$ of the feature model consists of:

- A model field declaration

  ```
  //@ model public nullable boolean[] feature;
  ```

  including an invariant stating that the length of the `feature` array is equal to the number of features declared in FM.
- A sequence of ghost field declarations

  ```
  //@ ghost public final static int f_0 = 0;
                              ⋮
  //@ ghost public final static int f_n = n;
  ```

  Each ghost field declaration `f_i` defines a compile-time constant associating the corresponding feature f_i uniquely with an array component of the previously declared model field `feature` such that feature f_i is selected iff `feature[f_i]==true`.

– A set of conjunctively connected boolean JML expressions $\mathsf{Inv} = \{e_0, \ldots, e_n\}$ such that each expression encodes the relationship between a feature and its immediate subfeatures.

The conjunction of the JML expressions in Inv encodes the *FM* diagram (recall that we only consider *FM* models being trees). Wlog. we describe now the construction of the JML expression $e_0 \in \mathsf{Inv}$ encoding the relationship between the parent feature f_0 and its children $f_1, \ldots, f_m$: e_0 is the conjunction of

1. `feature[f_i]==>feature[f_0]` (for all $0 \leq i \leq m$) encoding the ancestor link
2. `feature[f_0]==>feature[f_i]` for each mandatory feature f_i
3. ```
   feature[f_0] ==>
      (feature[f_1] &&!feature[f_2] && ... && !feature[f_k])
       || ... ||
      (!feature[f_1] && ... && !feature[f_(k-1)] && feature[f_k])
   ```
   for each alternative relationship between parent and a subgroup of its children $f_l, \ldots, f_k$ where $1 \leq l < k \leq n$)
4. Analogous expressions for the remaining parent-child relationships.

Based on this definition, we implemented an automatic translation from feature models to a JML specification fragment to be used as part of JML invariants and method specifications.

4.3 Connecting Specification and Implementation

While the specification generated in Sect. 4.2 describes all valid feature configurations, it is not yet connected to the actual implementation of the cash desk system. In this section we explain how to relate the feature vector used in the specification to the implementation. The generated feature specification is used to ensure that

1. the `Configurator` accepts only *valid* feature configurations;
2. the `CashDesk` system built by the `Configurator` has all components required by the selected feature configuration.

We start with item 1. In a first step the model and ghost field declarations from above are inserted into the `Configurator` class. In addition, we need to add the invariant Inv, however, since the invariant can only be expected to hold after the feature vector is determined and initialized we add a guard that ensures it:

`//@ public invariant !feature == null ==>` Inv

Next, the model field `feature` is related to the actual implementation by adding a JML `represents` clause defining how the model field can be mapped to concrete Java constructs. This mapping is trivial and simply states that `feature` is represented by the field `realFeatures` of class `Configurator`.
   ```

The JML semantics says that each non-helper method preserves all invariants, so our specification expresses already that `setFeatureVector(boolean[])` may only accept valid feature configurations. It is straightforward to construct a normal behavior method specification for `setFeatureVector(boolean[] f)`: simply rename `feature` in e with the method parameter `f` and use `feature == f` as postcondition, the only a valid configuration is actually accepted.

Moving to item 2. above, the `feature` array needs to be more closely related to the underlying Java implementation. This can be done in a systematic manner by annotating the Java feature model *FM* with a mapping that maps each feature f_i to a JML expression $\phi(f_i)$ to be used as an additional invariant to be established after deployment of the product and preserved thereafter. For example, an annotation ensuring that the created cash desk `cashDesk` is equipped with a properly registered keyboard is:

```
feature != null && feature[_keyboard] ==>
 (\exists KeyboardProductInput kpi;
    (\exists int i; 0<=i && i < cashDesk.stateChangeListenerSize;
             cashDesk.stateChangeListener[i]==kpi) &&
    (\exists int j; 0 <= j && j < cashDesk.devicesSize;
             cashDesk.devices[j] == kpi))
```

5 Verification

We used the KeY verification system [4] to prove that the feature configuration validity check and the cash desk system setup procedure are implemented faithfully with respect to the specification given in Sect. 4.

We were in particular interested how well a current state-of-the-art verification tool scales when verifying highly adaptable software as developed in the context of software product families.

```
public void plugIn(CashDesk cashDesk) {
   if (realFeatures[SCANNER]) {
     final ScannerDevice scanner = new ScannerDevice(cashDesk);
     cashDesk.addInputDevice(scanner);
     cashDesk.addStateChangeListener(scanner);
   }

   if (realFeatures[NONCASH] && realFeatures[CREDITCARDREADER]) {
               ...
   }
   ...
}
```

Fig. 8. If-cascade implementing the cash desk initialisation logic

Before we could start the verification of our CoCoMe subsystem, we had to adapt the derived JML specification slightly. The reason is that KeY's support for model fields is somewhat rudimentary. Thus we decided to replace the `feature` model field by a ghost field of the same name. As the semantics of model fields is much more complex than that of ghost fields, we had also to change and extend JML specifications referring to the model field to achieve an equivalent and correct specification. Such a replacement is not possible in general but worked here well in our context due to the simple `represents` clause and by assuming a closed system, i.e., that all classes implementing input devices and payment methods are known in advance.

We were able to verify the correctness of the validity check and most parts of the actual cash desk creation and initialisation. In its original version the latter had been a monolithic method (`plugIn()` of class `Configurator`) consisting of if-cascades as shown in Fig. 8. Verification of this method was infeasible as the proof size exploded. We modularised the monolithic method and separated each if-cascade representing the creation and registration of a device or payment method into different methods. The specification of one of these methods `checkScanner(CashDesk)` is given in Fig. 9.

```
/*@
  @ public normal_behavior
  @ requires feature!=null;
  @ requires feature[_scanner];
  @ ensures
  @  (\exists ScannerDevice sd; \fresh(sd);
  @    (\exists int i; 0<=i && i< cashDesk.stateChangeListenerSize;
  @       cashDesk.stateChangeListener[i]==sd) &&
  @        (\exists int j; 0<=j && j< cashDesk.devicesSize;
  @                                      cashDesk.devices[j]==sd));
  @ assignable
  @  \object_creation(ScannerDevice),\object_creation(Scanner),
  @  cashDesk.stateChangeListenerSize, cashDesk.stateChangeListener,
  @  cashDesk.stateChangeListener[cashDesk.stateChangeListenerSize],
  @  cashDesk.devicesSize, cashDesk.devices,
  @  cashDesk.devices[cashDesk.devicesSize];
  @*/
```

Fig. 9. Specification of the `checkScanner` method

Afterwards we were able to verify most of the individual methods in isolation. Fig. 10 shows statistics about the performed proofs (all fully automatic) and their size. For two methods, `checkCreditCard` and `checkPrepaidCard` we could not yet obtain proofs. We are currently analyzing the problems and we are confident that we can present proofs in the final version of this paper.

Method	Nodes	Branches
checkScanner	22032	107
checkCash	14150	77
checkKeyboard	15755	64

(a) Ensure Postcondition

Method	Nodes	Branches
checkScanners	40439	429
checkCash	20265	161
checkKeyboard	46392	664

(b) Correct Assignable Clause

Method	Nodes	Branches
checkScanners	89492	703
checkCash	49421	327
checkKeyboard	70962	485

(c) Preserve Invariant

Fig. 10. Proof Statistics

6 Related & Future Work

Related Work. In [6] the authors describe an approach to open system verification of software product lines by parametrised interfaces. The verification technology is (3-valued) model checking. Features and the core product are equipped with interfaces externalising certain states as input and output states. Features can be composed to complex features or to whole products by connecting to the core product using these interfaces.

The authors aim to allow compositional (contract-based) reasoning using model checking by computing subcontracts for the interfaces. When composing the features to a complete product only the subcontracts have to be discharged. Specifically, for a given global property of a product, constraints to be posed onto the exposed input and output states are computed independently for each feature. At composition time one has then only to ensure that these constraints are satisfied by the preceding/succeeding features. The constraints for the preceding features are propositional formulas restricting the values of the input values, while those of the succeeding features are temporal logic formulas ensuring that certain properties are adhered to in the future. The presented approach allows to compose products arbitrarily and eases verification by having only to discharge the computed constraints for the derived product, but is limited to incremental features.

The authors of [2] describe an approach to verification of a software product lines based on ASMs and the AHEAD methodology. Their case study is built upon the Jbook [14], where a complete virtual machine for Java 1.0 has been modelled including an interpreter and compiler. The compiler was proven correct wrt. the interpreter.

The authors restructured the Jbook case study to fit into the feature modelling approach as enforced by the *feature-oriented programming* (FOP) design

methodology which provides a technology for compositional program assembly. The so obtained structure has a base layer or core representing only a subset of Java expressions (imperative expressions). This core is stepwise extended by adding new features (layers) such as imperative statements, class and object features until complete coverage of the Java 1.0 language is reached.

In this framework a strong structural connection exists between model extension and the correctness proof of the compiler. This allows to alter the existing correctness proof by adding new independent cases (e.g., for new supported language constructs) or to refine existing cases by an additional invariant to be proven. Features having a non-compositional or destructive influence for existing cases occurred either rarely or not at all. Correctness has been proven (mostly) by hand without any automation or even machine-checked proof support.

Future Work. We differ substantially in our objectives and the underlying technology from the work discussed above: we aim at a highly automatised compositional design and verification system that is applicable to adaptive systems in general and specifically to software product line engineering. In basing our work on an expressive program logic and specification framework realized in a verification system with a high degree of automation we overcome some principal limitations, however, we are fully aware that there are considerable research challenges ahead:

- We believe that it is not sufficient to achieve compositionality by manually adding case distinctions or refining existing ones. The verification system and methodology must inherently construct proofs that are accessible to compositional reasoning and—where this is not possible—apply proof reuse techniques.
- Support for destructive features is essential: the restriction to mostly incremental features and consequently conservative extensions is not sufficient for our purposes. Adding new features may easily render existing proof cases invalid and require a completely new proof. Again, proof reuse is of essence.
- Independence of new features and existing proofs: even if a new feature has no influence on, say, a certain class invariant, this needs to be proven (or enforced) explicitly. In case of real-world languages like Java with aliasing this is still an area of active research [10, 13].

Our case study showed that formal specification and verification of software product families is, in principle, possible with current technology and can actually be achieved for small examples. Nevertheless, the results of our case study are not satisfactory from our point of view. Specific problems, such as missing support for model fields which are crucial for verification of open systems, are specific shortcomings of the used verification tool KeY and will be resolved in the near future. Others issues, however, such as proof-size explosion due to the resolution of variability points needs to be solved on a methodological level. Research regarding this issue is under way. It would also be interesting to explore

how separation-logic based approaches perform in the context of software families and if they can overcome some of the problems we faced because of framing issues.

References

1. D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines*, pages 7–20. Springer-Verlag, 2005.
2. D. S. Batory and E. Börger. Modularizing theorems for software product lines: The jbook case study. *J. UCS*, 14(12):2059–2082, 2008.
3. J. Bayer, C. Gacek, D. Muthig, and T. Widen. Pulse-i: Deriving instances from a product line infrastructure. *Engineering of Computer-Based Systems, IEEE International Conference on the*, 0:237, 2000.
4. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.
5. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 2010.
6. C. Blundell, K. Fisler, S. Krishnamurthi, and P. V. Hentenryck. Parameterized interfaces for open system verification of product lines. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 258–267. IEEE Computer Society, 2004.
7. K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 23–34, 2007.
8. K. C. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon University Software Engineering Institute, 1990.
9. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. *JML Reference Manual*, Sept. 2009. Draft revision 1.235.
10. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer, 2002.
11. A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors. *The Common Component Modeling Example: Comparing Software Component Models [result from the Dagstuhl research seminar for CoCoME, August 1-3, 2007]*, volume 5153 of *LNCS*. Springer, 2008. Preliminary version of the chapter describing the Trading System is available at: http://agrausch.informatik.uni-kl.de/CoCoME/downloads/documentation/cocome.pdf.
12. I. Schaefer, A. Worret, and A. Poetzsch-Heffter. A Model-Based Framework for Automated Product Derivation. In *Proc. of Workshop in Model-based Approaches for Product Line Engineering (MAPLE 2009)*, 2009.
13. J. Schäfer, M. Reitz, J.-M. Gaillourdet, and A. Poetzsch-Heffter. Linking programs to architectures: An object-oriented hierarchical software model based on boxes. In *CoCoME*, pages 238–266, 2007.
14. R. F. Stark, E. Börger, and J. Schmid. *Java and the Java Virtual Machine: Definition, Verification, Validation with Cdrom*. Springer, 2001.