Hephaestus
A Tool for Managing SPL Variabilities

Rodrigo Bonifácio and Leopoldo Teixeira and Paulo Borba

[1]Informatics Center – Federal University of Pernambuco (UFPE)
Recife – PE – Brazil

`{rba2,lmt,phmb}@cin.ufpe.br`

Abstract. *In this paper we present Hephaestus, a suite of tools that follows a crosscutting approach for the product engineering phase of software product line (SPL) development. Here we focus on some design decisions that led the development of Hephaestus, and also present how it could be used for generating product specific use cases and build files of a well known case study: the Mobile Media product line.*

1. Introduction

A software product line (SPL) is a set of related software products that are generated from reusable assets. Products are related in the sense that they share common functionality. Assets might correspond to different kinds of artifacts, such as requirements, design, source code, configuration files and tests. The reuse of assets targeted at a specific set of products can bring significant productivity and time to market improvements [Pohl et al. 2005]. Indirectly, quality can be improved too [Linden et al. 2007], since assets are typically more exposed and tested by being used in different products. Software mass customization is one of the trends of SPL engineering, which includes tool support for automatically composing and configuring assets in different ways [Krueger 2006]. In order to do that, tools usually have to consider different models:

- **Feature models (FM)** that describes the domain by representing the common and variable features of an SPL. It is understood that the relationships and constraints of a FM in fact represent a propositional formula whose instances correspond to the SPL members [Batory 2005].
- **Instance models (IM)** that represent SPL members. Therefore, an IM is usually represented as a selection of features that satisfies all FM constraints.
- **Product line assets (PLA)** that represent (configurable) artifacts of a SPL. It is important to remind that PLA might involve different kinds of artifacts.
- **Configuration knowledge (CK)** that is responsible for relating configurations of features to product line assets. Actually, there exists different CK representations, whereas the most simple basically maps one feature to one asset.

The main contribution of this paper is to present *Hephaestus*[1], a suite of libraries and tools that evaluates SPL models and generates artifacts for specific instances of a SPL. *Hephaestus* provides a simple graphical interface that basically allows product engineers to select the input models of the product derivation process (FM, IM, CK, and PLA). In the current version, two types of PLA are supported: use case scenarios, to generate product specific use case models, and mappings between names and source code files, to generate

[1]Hephaestus was the Greek god of technology, craftsmen, and artisans.

build files in a suitable format for compiling Java/AspectJ programs. The result of the building process is an abstract representation of a SPL member that could be exported to different formats. For instance, currently we export product specific use case models to both LaTeX and MS Word Documents.

Figure 1 shows the big picture of *Hephaestus*, highlighting the models used as input for the product derivation process, as well as the tool main window, presenting some of its features. For example, besides deriving product specific use case models and build files, *Hephaestus* also allows product engineers to check satisfiability and detect some mistakes that are likely to occur in feature models. These later operations are not covered in this paper.

It is important to notice that the input models are not specified using *Hephaestus*. Actually, different tools might be used for specifying them. For example, we make use of MS Word documents, following a particular template, to write use case scenarios. These documents are exported to XML and then parsed by a specific *Hephaestus* library. Similarly, feature models, as well as instance models, can be specified using available tools, such as *Feature Modeling Plugin* or *Feature IDE*.

As a final remark for this section, in this paper we draw special attention to the design of our configuration knowledge, which differs from existing tools [Beuche 2003, Cirilo et al. 2008] in the sense that it relates feature expressions to actions that evolve PLA to product specific artifacts. Thus, a given feature expression could be related to several actions. This design simplifies the configuration knowledge, since it is able to remove multiple copies of an expression. Such a duplication might arise when relating a single component to its required features.

2. Hephaestus

Hephaestus development was initially proposed to validate one approach for representing variability in requirements models [Bonifácio and Borba 2009]. Recently, we introduced new features, allowing users to generate product specific build files and check several properties of features models. We start this section presenting usage scenarios for managing variabilities in two different types of artifacts: requirements and build files. Then we present an overview of the main building blocks of Hephaestus. The usage scenarios are based on examples from the Mobile Media product line [Figueiredo et al. 2008], which contains applications that manipulate media such as photo, music, and video on mobile devices.

2.1. Managing requirement variabilities

Following the terminology proposed by [Bachmann and Bass 2001], *Hephaestus* supports three types of requirements variability:

- **Variability in function.** Occurs when a particular function (detailed as use case scenarios here) might exist in some products and not in others.
- **Variability in control flow.** Occurs when a pattern of *user-system* interaction within a scenario varies from one product to another.
- **Variability in data.** Corresponds to fine grained variations and occurs whenever two or more scenarios share the same behavior and differ in relation to the values of a same concept.

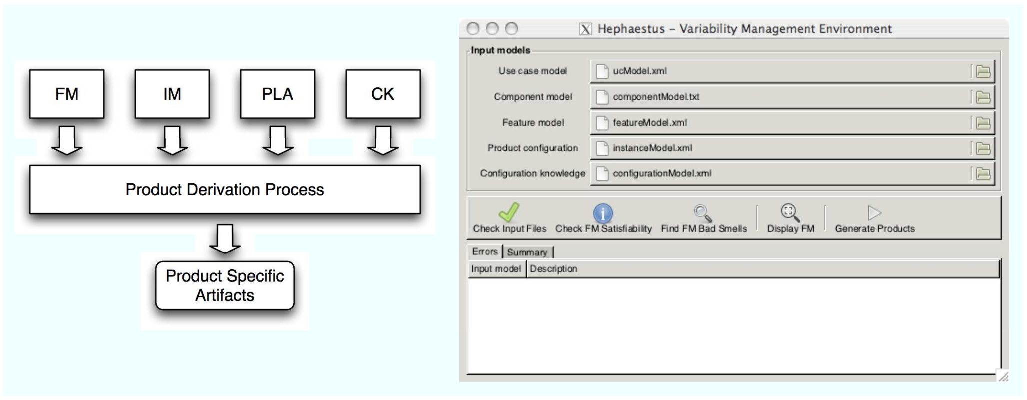


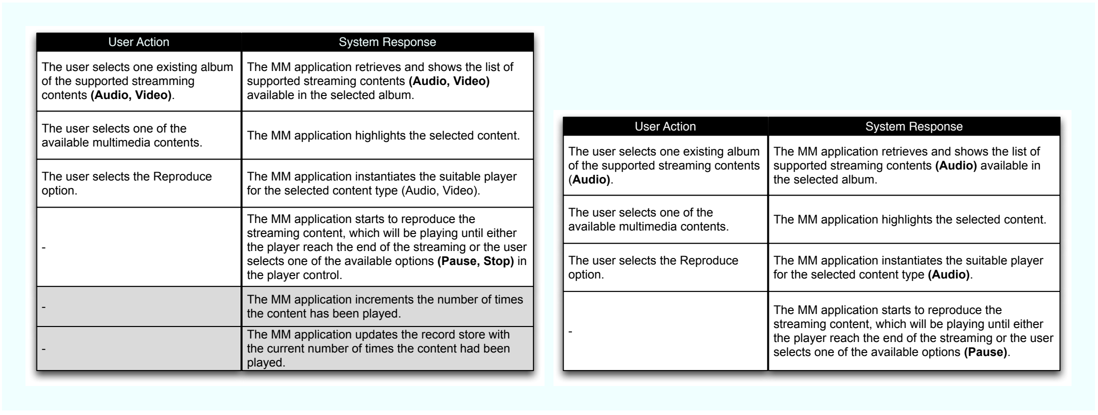
Figure 1. High level view and GUI of Hephaestus.

For instance, consider the reusable scenario *Reproduce Media*, which can be configured according to the supported types of streaming, the available control options, and the selection (or not) of the *Favorite* feature. Two instances of this scenario are shown on Figure 2. The first instance of *Reproduce Media* scenario (in the left side of the figure) will be found in products with the following configuration:

⋆ *Streaming Content (Audio, Video), Control Options (Pause, Stop), Favorite*

whereas the second one will be found in products configured with:

⋆ *Streaming Content (Audio), Control Options (Pause),* **Not** *(Favorite)*



User Action	System Response
The user selects one existing album of the supported streamming contents **(Audio, Video)**.	The MM application retrieves and shows the list of supported streaming contents **(Audio, Video)** available in the selected album.
The user selects one of the available multimedia contents.	The MM application highlights the selected content.
The user selects the Reproduce option.	The MM application instantiates the suitable player for the selected content type (Audio, Video).
-	The MM application starts to reproduce the streaming content, which will be playing until either the player reach the end of the streaming or the user selects one of the available options **(Pause, Stop)** in the player control.
-	The MM application increments the number of times the content has been played.
-	The MM application updates the record store with the current number of times the content had been played.

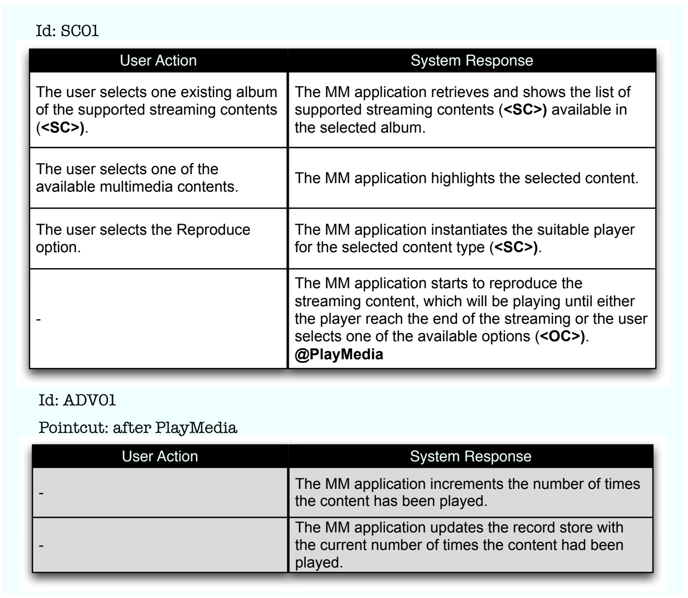
User Action	System Response
The user selects one existing album of the supported streaming contents (**Audio**).	The MM application retrieves and shows the list of supported streaming contents **(Audio)** available in the selected album.
The user selects one of the available multimedia contents.	The MM application highlights the selected content.
The user selects the Reproduce option.	The MM application instantiates the suitable player for the selected content type **(Audio)**.
-	The MM application starts to reproduce the streaming content, which will be playing until either the player reach the end of the streaming or the user selects one of the available options **(Pause)**.

Figure 2. Two configurations of *Reproduce Media*

In order to deal with variabilities between instances of the same scenario, we had to propose new constructs to describe use cases [Cockburn 2000]: aspectual use cases and parameters. Aspectual use cases deal with variability in control flow (as required by the *Favorite* feature in the current example, whose behavior is represented by the *gray* steps in the first configuration of Figure 2). Scenario parametrization, on the other hand, deal with variability in data (as required by the *Streaming Content* and *Control Options* features). Variability in function did not demand any new construct to use case modeling. Indeed, we represent which selection of features requires a scenario through the configuration knowledge (as we explain later).

Using these new constructs, we could represent both configurations of Figure 2 using: (a) a parameterized scenario (SC01), which defines the common behavior of *Reproduce Media* and (b) an aspectual use case (ADV01) that specifies the optional behavior required by the *Favorite* feature. We represent this design in Figure 3.

In our notation, parameters are represented as names enclosed by "$<$" and "$>$". There are four references to parameters in the scenario of Figure 3. In the first and third steps, references to the $< SC >$ parameter abstract the options of the *Streaming Content* feature. Similarly, the reference to the $< OC >$ parameter abstracts the *Optional Controls*. Further, aspectual use cases differ from scenarios because they have a pointcut clause (either After or Before). In this clause we refer to special annotations written in steps of scenarios (or even steps of advice). The evaluation of an advice combines its steps before (or after) any step that matches the pointcut clause. In the current example, the fourth step of the scenario SC01 (see Figure 3) has the `@PlayMedia` annotation. Consequently, evaluating the advice ADV01 results in a flow of events similar to the scenario described in the left side of Figure 2.



Id: SC01

User Action	System Response
The user selects one existing album of the supported streaming contents (**<SC>**).	The MM application retrieves and shows the list of supported streaming contents (**<SC>)** available in the selected album.
The user selects one of the available multimedia contents.	The MM application highlights the selected content.
The user selects the Reproduce option.	The MM application instantiates the suitable player for the selected content type (**<SC>**).
-	The MM application starts to reproduce the streaming content, which will be playing until either the player reach the end of the streaming or the user selects one of the available options (**<OC>**). **@PlayMedia**

Id: ADV01

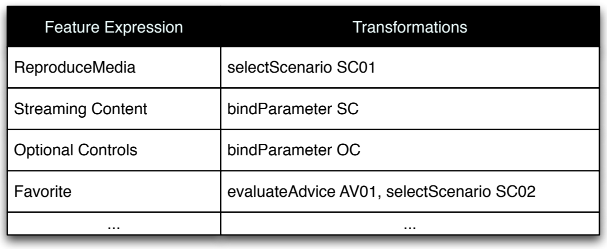
Pointcut: after PlayMedia

User Action	System Response
-	The MM application increments the number of times the content has been played.
-	The MM application updates the record store with the current number of times the content had been played.

Figure 3. Specification of the features Reproduce Media and Favorite.

Note that scenarios (or advices) do not make explicit references to features. Actually, the configuration knowledge is responsible for relating PLA (in this section, use case scenarios) to features. Indeed, the structure of our configuration knowledge relates feature expressions to transformations that translates PLA into product specific artifacts. If a feature expression is evaluated as *True* for a given instance model, the related transformations are applied.

Three distinct transformations deal with the types of variability discussed here: select scenario (deals with variability in function), evaluate advice (deals with variability in control flow), and bind parameter (deals with variability in data). Therefore, the configuration knowledge of Figure 4 covers the configurability of the *Reproduce Media* scenario.



Feature Expression	Transformations
ReproduceMedia	selectScenario SC01
Streaming Content	bindParameter SC
Optional Controls	bindParameter OC
Favorite	evaluateAdvice AV01, selectScenario SC02
...	...

Figure 4. Instance of the CK for managing requirements variability.

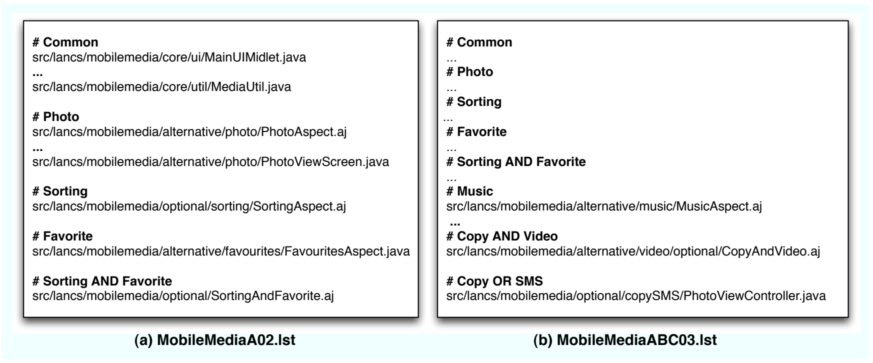
2.2. Managing build files variabilities

To create an instance of the Mobile Media product line [Figueiredo et al. 2008], we use a build file (*.lst*) to list all the source files (classes and aspects) that should be compiled by the build task. In the original way that the Mobile Media product line is set up, for each instance of the SPL we have to manually create a build file that describes it. As an example, consider the *MobileMediaA02.lst* file depicted on Figure 5 (a) and the *MobileMediaABC03.lst* file depicted on Figure 5 (b). The first file describes the classes and aspects that comprise the product that includes the following features:

⋆ *Photo, Sorting, Favorite*

whereas the second file describes products configured with:

⋆ *Photo, Music, Video, Sorting, Favorite, Copy, SMS, Capture*



```
# Common
src/lancs/mobilemedia/core/ui/MainUIMidlet.java
...
src/lancs/mobilemedia/core/util/MediaUtil.java

# Photo
src/lancs/mobilemedia/alternative/photo/PhotoAspect.aj
...
src/lancs/mobilemedia/alternative/photo/PhotoViewScreen.java

# Sorting
src/lancs/mobilemedia/optional/sorting/SortingAspect.aj

# Favorite
src/lancs/mobilemedia/alternative/favourites/FavouritesAspect.java

# Sorting AND Favorite
src/lancs/mobilemedia/optional/SortingAndFavorite.aj
```

(a) MobileMediaA02.lst

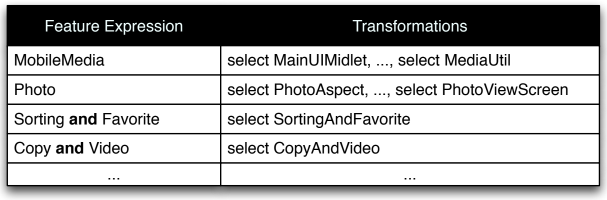
```
# Common
...
# Photo
...
# Sorting
...
# Favorite
...
# Sorting AND Favorite
...
# Music
src/lancs/mobilemedia/alternative/music/MusicAspect.aj
 ...
# Copy AND Video
src/lancs/mobilemedia/alternative/video/optional/CopyAndVideo.aj

# Copy OR SMS
src/lancs/mobilemedia/optional/copySMS/PhotoViewController.java
```

(b) MobileMediaABC03.lst

Figure 5. Simplified examples of build files for MobileMedia.

By analysis of these files, we noticed that some lines will be present in both products, while others are specific to a product. Actually, the *#Common* section will be the same for all products. In order to manage variability in these build files, we have to be able to relate feature expressions to classes and aspects. Therefore, we can generate build files for specific instances of the SPL, instead of having to manually create them. This facilitates the maintenance and evolution of build scripts. We can also clearly notice the need for a mapping of feature expressions to artifacts. For instance, this can be seen on Figure 5 (a): the line corresponding to the file *SortingAndFavorite.aj* will only be present when both features are selected. If we only mapped features to assets in a 1:1 relationship, creating another feature called *SortingAndFavorite* would be the only way we could achieve the same result. However, we would pollute the feature model with implementation decisions.

Again, the configuration knowledge will be responsible for relating features to PLA, which in this section, are classes and aspects to be compiled. In this particular case, the build file that is going to be generated only needs to include the path to the class or aspect to be compiled. Therefore, the structure of our configuration knowledge, in this case, will relate feature expressions to the transformation *select*, that maps a name to a source code file path. This mapping is important, for instance, to enable us to use classes and aspects with the same name, something that can be used on a SPL to represent variations. In a regular program, this would normally result in a compile error. The configuration knowledge of Figure 6 illustrates how we can manage variability on the build files, based on the examples from Figure 5.



Feature Expression	Transformations
MobileMedia	select MainUIMidlet, ..., select MediaUtil
Photo	select PhotoAspect, ..., select PhotoViewScreen
Sorting **and** Favorite	select SortingAndFavorite
Copy **and** Video	select CopyAndVideo
...	...

Figure 6. Simplified instance of CK for managing build files variability.

2.3. Hephaestus libraries

In fact, Hephaestus is a suite of Haskell libraries and tools for product line development. We justify our choice for Haskell and discuss about some design decisions regarding its implementation in Section 3. The following libraries have been developed:

- **Core:** Defines functions and data types that are available to all other *Hephaestus* libraries. For instance, it declares the *ParserResult* data type, which follows a monadic approach for dealing with parsers. All parsers defined in our tool suite depends on *ParserResult*.
- **Feature model:** Defines data types for feature modeling and several functions for reasoning about feature and instance models. For example, this library provides functions for:
  - translating feature models to propositional logic
  - checking feature model satisfiability
  - mining feature model bad smells

  Additionally, we have implemented a generic traversal function that can be used for developing new operations on FMs. Finally, there is a command line application within this library. As a consequence, users and developers interested only in FM reasoning and manipulation could use this library independently.
- **Use case model:** Defines an abstract representation of the *product line use case model*. This includes the definition of some data types, such as use cases, aspectual use cases, scenarios, and advice. Additionally, it implements all transformations that map a product line use case model into product specific use cases.
- **Component model:** Defines the mapping between names and source code files. Some items of the configuration knowledge refer to these names, in order to select the source files that should be included in a build file. Transformations related to the component model are defined in this library. Currently, only the *select* transformation, discussed before, is available.

- **Configuration knowledge:** Defines a novel representation of the configuration knowledge and implements a reusable engine of the product derivation process (see Figure 1). This engine is able to evaluate different kinds of transformations, so it does not have to change when new transformations are defined. Moreover, the output of the product derivation process is an abstract representation of a SPL member. Currently, this representation is composed by a product specific use case model and a list of references to source code files.

3. Hephaestus implementation

We have chosen Haskell as the programming language since the semantics of the *product derivation* process was formalized [Bonifácio and Borba 2009] using the crosscutting modeling framework [Masuhara and Kiczales 2003], whose original interpreters were implemented using Schema— another functional programming language. After formalizing that semantics, we realized that Haskell was suitable for developing *Hephaestus*, mainly because higher order functions and partial application of functions, two mechanisms that underlie the design of functional languages [Hughes 1989, Bird 1998], led to a simple and elegant implementation of both CK data type and product derivation process.

For instance, thanks to higher order functions, we could define the family of transformations as a type. Indeed, the *Transformation* type synonymous (see code below) corresponds to any function that receives as arguments the product line assets (PLA) and a (probably partial) version of the product assets. Then, this family of functions returns a refined version of the product assets. By following this design, the configuration knowledge was represented as a list of configuration items, which are pairs $(FeatureExpression, [Transformation])$. Therefore, the product derivation process filters the configuration items that declare valid feature expressions for an instance model and generates products by just applying all transformations (t:ts) declared in these configuration items.

```
type Transformation ::   PLA → Product → Product
type ConfigurationKnowledge = [ConfigurationItem]
data ConfigurationItem = ConfigurationItem {
  expression :: FeatureExpression,
  transformations :: [Transformation]
}
productDerivation fm im ck pla = refine tasks pla newProduct
  where
    tasks = concat [transformations c | c ← ck, eval im (expression c)]
    newProduct = ...
    refine [] pla product = product
    refine (t : ts) pla product = refine ts pla (t pla product)
```

The partial application of functions allowed us to instantiate configuration items with transformations that, in fact, could have different signatures. For instance, although the transformations required for managing the variabilities of Section 2.1 have the signatures:

```
selectScenario  :: Id → PLA → Product → Product
evaluateAdvice :: Id → PLA → Product → Product
bindParameter :: Id → Id → PLA → Product → Product
```

their partial application result in functions that matches the family of *transformations*. This occurs because the application `evaluateAdvice ''ADV01''` returns a function with type `PLA -> Product -> Product`. Therefore, we are able to define new transformations without changing the *product derivation* function (described above).

Although we do not want to claim here the reduction in size of the programs as a general and significant advantage of functional programming, generic traversals and combinators, supported by SYB, Parsec, and HXT libraries, simplify in a large amount the source code. For instance, we found that the Haskell implementation of an XML parser for use case documents is three times smaller than the corresponding one written in Java. Also regarding size, in this section we discussed about the concise implementation of the product derivation process, which required less than 15 lines of code. Of course, transformations are responsible for part of the *hard work*. Indeed, each transformation required a different effort to be developed. But again, thanks to the generic traversals offered by the Scrap Your Boilerplate library [Lämmel and Peyton Jones 2003], we could keep their implementation concise, varying from 8 to 30 lines of source code.

We started *Hephaestus* development with basic knowledge of the Haskell programming language. Therefore, we had to learn about some design principles and advanced mechanisms (such as combinators, monads, arrows) of functional programming throughout its development. However, most of the third-party libraries used in *Hephaestus* (such as Scrap Your Boilerplate, HXT, Parsec, and gtk2HS) are well documented and they have really simplified our learning process and reduced our development effort. Nevertheless, developers habituated to the imperative style of programming might consider the design of instances of these mechanisms a challenging task. Finally, we would like to point a question about portability. For those not familiar with compiling programs in Unix like systems, porting Haskell code is a bit more tricky than porting Java code. Even deploying applications using the CABAL package format [2], we have to follow a process of *configure-build-install* to port a Haskell application to different environments.

4. Summary

Variability management is considered a challenge for product line adoption, mainly because a feature usually requires variation points to be scattered through different models. Additionally, there are some types of interactions that occur when the presence (or absence) of a feature changes the behavior of the other ones. In order to partially solve these problems, tools for product line development decouple features and product line assets by means of configuration models [Beuche 2003, Cirilo et al. 2008]. This paper presents *Hephaestus*, a set of libraries and tools that supports SPL variability through a novel representation of the configuration model. Differently from existing tools, our CK relates features expressions to extensible transformations. In this paper we show two usage scenarios of *Hephaestus*, applying it to manage different types of variabilities of the Mobile Media product line. An official version of *Hephaestus* is available in the *Software Productivity Group* web site[3]. This version has been used for managing variabilities in different case studies, such as the Mobile Media and TaRGeT product lines. A comparison between *Hephaestus* with other tools is a matter of future work.

[2]http://www.haskell.org/cabal/
[3]http://www.cin.ufpe.br/spg

Acknowledgment

We gratefully acknowledge Prof. João Saraiva for guidance on functional programming and Haskell. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES[4]), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08. The first and second authors are respectively supported by the grants CNPq CT-INFO 17/2007 and MCT/CNPq/CT-INFO 07/2007.

References

Bachmann, F. and Bass, L. (2001). Managing variability in software architectures. *SIG-SOFT Softw. Eng. Notes*, 26(3):126–132.

Batory, D. (2005). Feature models, grammars, and propositional formulas. Technical Report CS-TR-05-14, University of Texas at Austin, Dept. Computer Science.

Beuche, D. (2003). Variant management with pure:: variants. Technical report, Pure-Systems GmbH. At http://www.pure-systems.com.

Bird, R. (1998). *Introduction to Functional Programming using Haskell*. Prentice Hall PTR, 2 edition.

Bonifácio, R. and Borba, P. (2009). Modeling scenario variability as crosscutting mechanisms. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 125–136, New York, NY, USA. ACM.

Cirilo, E., Kulesza, U., and Lucena, C. (2008). A product derivation tool based on model-driven techniques and annotations. *Journal of Universal Computer Science*, 14(8).

Cockburn, A. (2000). *Writing Effective Use Cases*. Addison-Wesley Professional.

Figueiredo, E. et al. (2008). Evolving software product lines with aspects: an empirical study on design stability. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 261–270, New York, NY, USA. ACM.

Hughes, J. (1989). Why functional programming matters. *Comput. J.*, 32(2):98–107.

Krueger, C. W. (2006). New methods in software product line practice. *Commun. ACM*, 49(12):37–40.

Lämmel, R. and Peyton Jones, S. (2003). Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

Linden, F. J. v. d., Schmid, K., and Rommes, E. (2007). *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*, chapter 8–10. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Masuhara, H. and Kiczales, G. (2003). Modeling crosscutting in aspect-oriented mechanisms. In *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pages 2–28. Springer.

Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[4]http:://www.ines.org.br