Techniques for Developing a Product Line of Product Line Tools: a Comparative Study

Lucinéia Turnes and Rodrigo Bonifácio and Vander Alves
Computer Science Department
University of Brasília
Brasília, Brazil
Email: lucineiaturnes@gmail.com, {rbonifacio,valves}@cic.unb.br

Ralf Lämmel
Software Languages Team
Universität Koblenz-Landau
Germany
Email:rlaemmel@gmail.com

***Abstract*—Tool support is essential for Application Engineering in Software Product Lines (SPL). Despite a myriad of existing tools, most lack adequate support for flexibility and adaptability, so that it is hard for them to be applied in different contexts, e.g., addressing variability in different artifacts. Addressing this issue requires exploring underlying commonality and adequately managing variability of such tools. In order to provide systematic guidance in this direction, we have conducted a comparative analysis of variability management techniques for SPL tool development in the context of the SPL Hephaestus tool. The analysis reveals that two techniques, one annotative and another transformational, are most suitable to variability management in Hephaestus, and that their combination is a feasible strategy to improve such management.**

I. Introduction

A Software Product Line (SPL) is a set of software intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [1]. Potential benefits include improved productivity with lower development costs and time-to-market and increased quality. To achieve these, tool support for its underlying activities is essential. In particular, this holds for Application Engineering, in which a product is defined by selecting a group of features and then a carefully, coordinated mixture of parts of different components are involved. Given the inherent complexity and the coordination required in the derivation process [2], this activity is slow and error-prone. As a result, the derivation of individual products from shared software assets is still a time-consuming and expensive activity in many organizations [3].

As reported by a contemporary Systematic Literature Review and expert survey [4], a key requirement of product derivation tools is flexibility and adaptability, which the same study identifies as a shortcoming of most existing tools. These must be adapted to different contexts, e.g., dealing with different artifacts. Additionally, the changing needs of users and the continuous evolution of the product line further motivate flexibility and adaptability of product derivation tools for addressing future needs. Therefore, to provide flexibility and adaptability, it becomes necessary to adequately manage the variability within these tools, addressing the tools themselves as SPLs, as also suggested by Grünbacher et al. [5].

Accordingly, this paper presents a comparative analysis of variability management mechanisms for SPL tool development, particularly suitable to use in functional languages. We explore these in the context of existing variants of Hephaestus [6], a SPL tool developed in Haskell and originally aimed at managing variability in requirements, but which has evolved to handle variability in different kinds of artifacts. A detailed description of Hephaestus, including usage scenarios, could be found elsewhere [6]. The analysis assesses such mechanisms and proposes a strategy more suitable for variability management within the development of different versions of that tool. Results reveal that two techniques (CIDE and Stratego/XT) are most suitable to variability management in Hephaestus (whose variability is described in the next section), and that their combination is a feasible strategy.

The remainder of this paper is organized as follows. Section II briefly describes Hephaestus and its evolution to address different artifacts. Section III presents the setting of the comparative analysis, which is then carried out in Section IV. Section V addresses related work and Section VI offers concluding remarks.

II. Hephaestus

Hephaestus [6] is a public[1] available product derivation tool [3], which receives contributions from different institutions (Federal University of Pernambuco, University of São Paulo, University of Brasília). Initially developed as a proof of concept tool for managing variabilities in use case scenarios, Hephaestus provides a declarative specification (Haskell code) of the composition style for solving SPL variability in MSVCM (Modeling Scenario Variability as Crosscutting Mechanisms) [7]. Currently, Hephaestus supports variability in different types of assets, ranging from business processes and Simulink models to source code; and it has been used as the derivation tool for TaRGeT product line [8].

For the initial purpose of the tool, we first implemented:

- specific data types representing use case models, feature models and configuration knowledge models [9], which relates feature expressions in propositional logic to transformations that deal with variability in use cases.

[1]http://bit.ly/iRMMZM

```
type Transformation = SPL → Product → Product
type ConfigurationKnowledge = [ConfigurationItem]
data ConfigurationItem = ConfigurationItem {
  expr = FeatureExpression,
  transformations = [Transformation]
}
build fm fc ck spl = derive ts spl emptyInstance
where
  ts = concat [transformations c | c ← ck, eval fc (exp c)]
  emptyUCM = ...
  emptyInstance = ...
derive [] spl product = product
derive (t : ts) spl product = derive ts spl (t spl product)
data SPLModel = SPLModel {
  splFeatureModel :: FeatureModel,
  splUseCaseModel :: UseCaseModel
}
data InstanceModel = InstanceModel {
  featureConfiguration :: FeatureConfiguration,
  useCaseModel :: UseCaseModel
}
exportProduct :: Path → InstanceModel → IO ()
exportProduct t product = do
  exportUcmToLatex (t ++ "/doc.tex") (ucm product)
exportUcmToLatex = ...
```

Fig. 1. Code snippet of the initial implementation of Hephaestus

- specific functions that solve SPL variabilities in use case scenarios, by selecting use cases or scenarios from an SPL model and binding parameters according to specific feature configurations. In addition, Hephaestus provides a *build function* that behaves like an interpreter for the configuration knowledge and is responsible for building a specific product given a selection of features (or feature configuration).

Figure 1 presents a piece of code of the initial implementation of Hephaestus, highlighting the configuration knowledge data type and corresponding interpreter (the build function) and the signature of the transformation functions. In addition, this code snippet also shows the initial representation of the *SPLModel* and *InstanceModel* data types, as well an *exportProduct* function that generates a LaTeX representation of a product specific use case model.

A. First Hephaestus evolution

Hephaestus was extended to use MSVCM in practice. Thus, starting from a prototype for experimenting with MSVCM, we evolved Hephaestus into a tool that could be used by students and practitioners. Additionally, within a short period of time, we had to extend Hephaestus in another direction, so that it could manage variability not only in use case scenarios, but also in higher level requirements and source code (by selecting specific files that should be compiled as well by starting a preprocessing engine to solve variability in source code). At that time, Hephaestus should be used as a replacement for a proprietary tool that was used to manage variabilities in the TaRGeT product line [8], and new product assets should be exported as a consequence of the build process. We show some of the features related to this version on the right-hand side of Figure 2, while on the left-hand side we show the feature model of the first implementation.

In order to achieve these goals, new data types and transformations were required, as well as part of the existing code had to change. Precisely, to introduce support for variabilities in high level requirements (or requirements for short) and source code, we had to:

(a) implement new data types for representing requirements and references to source code assets.
(b) implement new transformations for resolving variabilities in requirements and source code. Existing transformations vary according to their complexity, each one requiring 10 to 100 lines of Haskell code.
(c) evolve both *SPLModel* and *InstanceModel* data types, as well as we had to review both the export function and the configuration knowledge XML parser, so that it could recognize the concrete syntax of the new transformations (see Figure 3).

Based on the mentioned items, a first conclusion is that, using current Hephaestus' architecture, we are not able to introduce new data types (representing the abstract syntax of a new SPL asset) and transformations in a modular way. This occurs because, to introduce support for a new type of asset, we have to change both *SPLModel* and *InstanceModel* data types, the configuration knowledge XML parser, and the export function— even though the *ConfigurationKnowledge* data type and interpreter present some degree of stability (we do not have to change their implementation when we introduce variability support for new assets). Here we could say that the *SPLModel* and *InstanceModel* are not open, since to introduce new SPL assets we have to change the data type definitions.

Evolving Hephaestus to support source code variability presents an interesting issue (see Figure 3), since we had to introduce a new kind of asset into the *SPLModel* (*splComponents*). This asset is a list of pairs that relate a name to the relative path of a source code file. The same kind of asset was also introduced into the *InstanceModel*. Besides that, other two fields were required in the InstanceModel: (a) *buildEntries*, which declares pre-processing directives, and (b) *preProcessFiles*, which declares a list of files that should be pre-processed by a third part tool. These fields are instantiated when Hephaestus builds a product, considering the proper transformations of a product configuration.

The piece of code in Figure 4 shows the impact on the configuration knowledge XML parser. The first version of Hephaestus declares just the first four case statements on the *xml2Transformation* function (from *selectScenarios* to *bindParameter*). The *selectRequirement* transformation deals with variability in the requirements models, whereas the remaining transformations solve variability in source code. We could say that the *exportProduct* and *xml2Transformation* are not open,

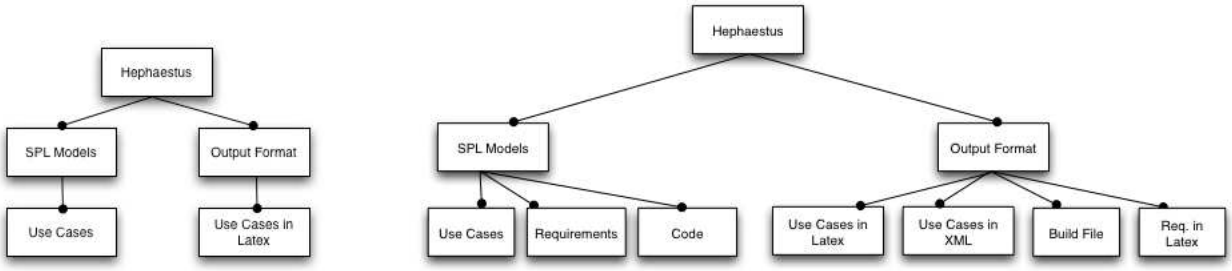

Fig. 2. Feature model of the first two releases.

```
data SPLModel = SPLModel {
  ...
  splReq :: RequirementModel,
  splComponents :: ComponentModel
}
data InstanceModel = InstanceModel {
  ...
  req :: RequirementModel,
  components :: ComponentModel,
  buildEntries :: [PreprocessingDirective],
  preProcessFiles :: [PreprocessingFiles]
}
exportProduct sourceDir targetDir product = do
  exportUcmToLatex ... (ucm product)
  exportUcmToXML ... (ucm product)
  exportRequirementsToLatex ... (req product)
  copySourceFiles ... (components product)
  exportBuildFile ... (preProcessDirectives product)
  preprocessFiles ... (preProcessFiles p)
```

Fig. 3. SPLModel and InstanceModel data types, after introducing support for managing variabilities in requirements and source code

```
xml2Transformation :: XmlTransformation
  → Parser Transformation
xml2Transformation transformation =
let
  args = ...
  tnsName = xmlTransformationName transformation
in
  case tnsName of
  "selectScenarios" → Success (selectScenarios args)
  "selectUseCases" → Success (selectUseCases args)
  "evaluateAspects" → Success (evaluateAspects args)
  "bindParameter" → ...
  "selectRequirements" → ...
  "selectComponents" → ...
  "selectAndMoveComponent" → ...
  "createBuildEntries" → ...
  "preprocessFiles" → ...
  otherwise → Fail "..."
```

Fig. 4. Piece of code used during the XML configuration knowledge parser

since new output formats and new types of transformations could not be introduced in a modular way.

B. Hephaestus Product Line

Although tailored to the TaRGeT product line needs, some users of this version of Hephaestus would appreciate more specific configurations of the tool. For instance, some users could be interested in managing variability only in requirements and use cases; others could be interested in managing variability only in source code; and TaRGeT engineers should be interested in managing variabilities in requirements, use cases, and source code.

Moreover, new extensions of Hephaestus were recently proposed. For instance, the current version of Hephaestus also supports variability in business processes models [10] and *Simulink* assets. Again, to introduce variability support for these assets, we have to implement new data types for representing the abstract syntax of these models, implement new transformations for solving variability, evolve the *SPLModel* and *InstanceModel* data types and review the configuration knowledge XML Parser. Otherwise, the *build* function, other supporting functions and data types are shared among all such extensions.

Therefore, there exists significant amount of commonality among these versions. Further, the variability has regular form (requiring both open data types and open functions), as we have explained previously. In order to explore the underlying commonality and to manage the variability systematically, it becomes essential to bootstrap these versions of Hephaestus into a SPL— hereafter referred to Hephaestus-PL–and then evolve it as a proper SPL. The intended feature model of Hephaestus-PL is represented in Figure 5. The following sections explore and compare different techniques for accomplishing this.

III. Study Settings

In order to refine the study, we apply the Goal Question Metric (GQM) method [11], so that it helps defining the context, the object of study, its properties, the goal, and how this latter can be operationalized and answered. In this section we first discuss about the goals, questions and metrics (GQM)

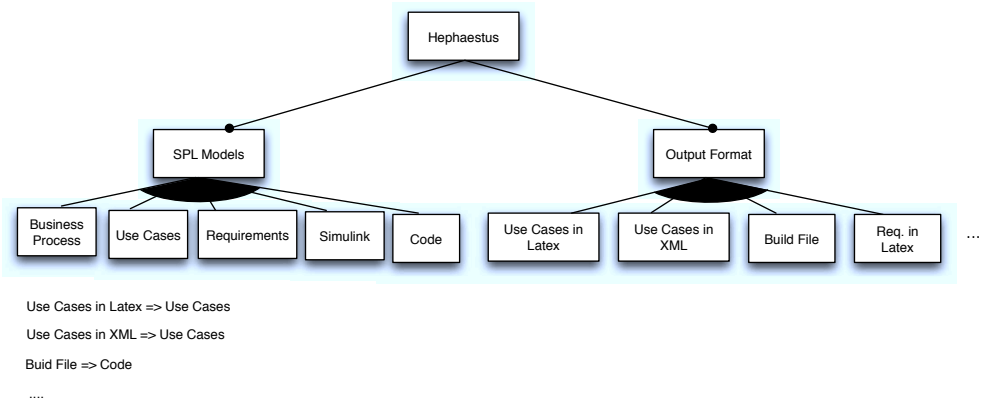

Fig. 5. Feature model of Hephaestus line. In this version, the features SPL Models and Output Format define an *or-exclusive* relationship with their children.

of our investigation and then present an overview of the evaluated techniques. Their detailed assessment is accomplished in Section IV.

A. Goals, Questions, and Metrics

This work aims to compare the applicability of different techniques for variability management, regarding the domain engineering phase for software product development and within the context of the different versions of Hephaestus discussed in the previous section. According to GQM, Table I summarizes the general goal of our work.

Purpose	compare
Issue	applicability
Object	different techniques to manage variability
Viewpoint	domain engineering perspective
Context	different versions of Hephaestus

TABLE I
GQM GOAL OF THIS RESEARCH.

In Section II, we characterized the *context* and the *viewpoint* of the initial Hephaestus design, which was targeted to manage variability in use case scenarios, and its evolution to support variability in other assets. In Section III-B, we shall briefly introduce the *object*, i.e., evaluated mechanisms and related languages and tools. After that, Section IV proceeds with the analysis of each technique, using a qualitative evaluation that answers our GQM metrics, thus meeting the *purpose* and *issue* components of the study's goal.

From the study's goal presented in Table I, we derive several questions that best characterize our study. Moreover, related to each question, we use one or more qualitative metrics to indicate the compliance level of the techniques in relation to the study goal. Qualitative assessment "metrics" are also conceived in the GQM method. This qualitative investigation is well suited because it allows us to evaluate high-level attributes that are relevant for the design of Hephaestus-PL before its actual implementation. Therefore, the use of quantitative metrics is outside the scope of this work and is scheduled within our plans for future work. In what follows, we present the questions (**Q**) and the metrics (**M**) of our GQM model.

Q1 *Does the technique have expressiveness to support Hephaestus variability?*
- Metric **M1.1** (Yes/No): The technique supports open data types.
- Metric **M1.2** (Yes/No): The technique supports open functions.
- Metric **M1.3** (Yes/No): The technique supports the instantiation of one SPL asset.
- Metric **M1.4** (Yes/No): The technique supports the composition of assets without having to instantiate all assets defined in Hephaestus-PL.

Q2 *Does the level of granularity supported by technique to handle variability meet Hephaestus-PL's needs?*
- Metric **M2**: Level of granularity of the technique for dealing with variability.

Q3 *Does the technique provide modular management of Hephaestus-PL's assets?*
- Metric **M3** (Yes/No): The technique provides support for feature modularization.

Q4 *What is the effort to use the technique in the current code of Hephaestus to manage the variability in Hephaestus-PL?*
- Metric **M4**: Impact level of using the technique in the current code of Hephaestus.

Q5 *What is the maturity level of the technique?*
- Metric **M5.1**: Related works, case studies, and applications using the technique.
- Metric **M5.2**: Domains of business where the technique has been applied.

Q6 *Is the technique applied in the Haskell functional language?*
- Metric **M6** (Yes/No): The technique could be easily applied to existing code written in Haskell .

Questions Q1, Q2, Q3, Q4, Q5 and Q6 represent relevant characteristics to the development of Hephaestus-PL, to

achieve the project requirements and minimize the impact on the current system. Q1 focuses on the most important attribute in our evaluation, which is the expressiveness to support the Hephaestus variability. Correspondingly, Metrics M1.1, M1.2, M1.3 and M1.4 investigate whether a technique supports the types of variability present in Hephaestus-PL, classified as open data types, open functions, single asset instantiation, and assets composition.

In particular, metric M1.4 focuses on a current problem of Hephaestus: the need to instantiate more than one SPL asset defined in *SPLModel* and *InstanceModel* algebraic types. The possible answers to these metrics are "yes" or "no". Q2 focuses on the level of granularity for implementing variabilities and its adherence to the current Hephaestus implementation. M2 may have possible answers as "coarse" or "fine-grained". Q3 assesses whether a technique supports modularity or not, an important property to the reactive approach for SPL development since it facilitates the introduction of new features to Hephaestus and minimizes prior knowledge of the details of system's modules and the location of features' implementation, which are often scattered. The possible responses to M3 are "yes" or "no". Q4 focuses on the effort required to apply a technique to the current code of Hephaestus. Therefore, M4 measures the impact level of the technique to the current code of Hephaestus. The possible answers to M4 are "high", "medium" or "low". Q5 assesses the maturity level of each technique with respect to its use in previous works, case studies, and business domains. Possible values of M5 are the names of works and business domains in which the technique has been applied. Finally, Q6 focuses on the adherence of the techniques to the current implementation of Hephaestus, more specifically, the use of the technique in Haskell. The possible responses to M6 are "yes" or "no".

B. Evaluated mechanisms

Kästner and others discuss about two common ways to implement an SPL: the compositional and annotative approach [12]. Regarding our analysis, we also consider complementary mechanisms based on program transformation and parametric polymorphism. These are essential to broaden the scope of design and implementation alternatives that are not restricted to given paradigms (e.g., OO and AOSD). Correspondingly, in this section we briefly introduce these approaches, postponing the details of their use to solve Hephaestus SPL variability to the next section. We evaluate the application of each mechanism by using (either at the implementation or at the design level) specific languages (Stratego and Aspect Caml), languages' constructs (Haskell type classes), or tools (CIDE); which have been chosen considering mainly our experience and their feasibility regarding our technical domain (extracting a SPL from an exising software developed in Haskell).

- **Annotative based:** Implementing SPL features using annotative approaches require some form of annotations in the artifacts into which features are projected. Annotations could vary from `#ifdef` and `#endif` statements in C or C++ to type-safer, language-independent mechanisms to implement variability as supported by CIDE [12], [13] or its semi-automated extension CIDE+ [14]. In this paper we evaluated the use of CIDE to implement Hephaestus variability.
- **Transformational based:** Solving SPL variability using this mechanism involves the development of domain specific languages and *source-to-source transformations*, which could be implemented by languages and tools like Stratego/XT[15]. In this paper we evaluate a domain specific language tailored to solve Hephaestus variabilities and implemented using Stratego/XT [16].
- **Compositional based:** In this approach, features are implemented as distinct modules, and a product line member is generated by composing a set of modules. A number of techniques fall into this approach such as component technologies, mixing layers, AHEAD, multi-dimensional separation of concerns, and Aspect-Oriented Programming (AOP). In this context, AOP could be used as a compositional approach for implementing SPL variability. AOP aims to better modularize crosscutting concerns in application development, and SPL features are usually crosscutting [17]. In spite of most implementations being target to extend object-oriented languages, there exist some AOP implementations for functional languages, such as Aspectual Caml [18] and AspectFun [19]. In our study, we evaluate Aspectual Caml as a mechanism for managing SPL variability using the compositional style, because it is already available in a functional language and it is more stable than AspectFun.
- **Parametric-polymorphism based:** Parametric polymorphism is one of the extension mechanisms that functional languages leverage. In Haskell, as well as in other functional languages, functions could be defined as polymorphic, with respect to their arguments and return types. This means that a polymorphic function could be applied to any type. However, often we have to express that a polymorphic function should be only applied to arguments of a type T given that T is an instance of a *type class*.

IV. Evaluated techniques

In this section, according to our GQM (Section III-A), we evaluate techniques within the mechanisms presented in Section III-B from both design and implementation perspectives.

A. Aspectual Caml

Aspectual Caml is an AOP language, derived from the strongly-typed functional language Objective Caml, which is a dialect of the ML functional language. Aspectual Caml extends the parser and type checker of the Objective Caml compiler, working as a translator, and provides support for static and dynamic crosscutting. In particular, static crosscutting allows extending data types by addition of new constructors into a data type and addition of new fields into the constructor of a data type. Dynamic crosscutting relies on pointcuts capturing

```
xml2Transformation transformation =
let
  args = ...
    tnsName = xmlTransformationName transformation
  in parse' tnsName args
parser' :: String → String → Parser Transformation
parser' "id" _ = Success Id
```

Fig. 6. Refactoring the configuration knowledge XML parser

events such as function calls and then applying a piece of advice with new behavior.

1) Refactoring Hephaestus into Hephaestus-PL using Aspectual Caml: In order to bootstrap Hephaestus-PL from the current design and implementation of Hephaestus, it is necessary to refactor the *SPLModel* and *InstanceModel* data types and the code of configuration knowledge XML parser shown in Figure 4. Accordingly, the refactored *SPLModel* and *InstanceModel* data types only have the *FeatureModel* field. Furthermore, the refactored *xml2Transformation* function does not implement the recognition of the concrete syntax for the transformation of features (see Figure 6). This is delegated to a new function *parser*, which has only basic behavior. The refactored code represents the commonality to be shared among different Hephaestus-PL instances. For each variant feature in Hephaestus-PL, an aspect is created as follows: 1) an advice will extend the *parser'* function to implement the recognition of the concrete syntax of the transformations of the feature; 2) static crosscutting will extend the *SPLModel* and *InstanceModel* data types to define additional field(s) corresponding to new asset(s) of such variant feature. The code in Figure 7 illustrates such an aspect in Aspectual Caml, *Aspect UCM*, that implements the variability associated to *UseCaseModel* feature.

```
aspect UCM {
type + SPLModel = SPLModel of ...* UseCaseModel
type + InstanceModel =
  InstanceModel of ...* UseCaseModel
pointcut pcUCMParser nameT argsT =
  call Parser' tnsName args
advice UCMParser =
  [around pcUCMParser tnsName args]
match tnsName with
  "selectScenarios" → Success (selectScenarios args)
  "selectUseCases" → Success (selectUseCases args)
  "evaluateAspects" → Success (evaluateAspects args)
  "bindParameter" → ...
}
```

Fig. 7. Modularizing features using Aspectual Caml

2) Aspectual Caml Evaluation: The AOP mechanisms in Aspectual Caml meet the main requirements for the development of Hephaestus-PL (Q1), i.e., Aspectual Caml addresses variability of data types (open data types) and functions (open functions). The composition of assets is implemented by the composition of aspects. The granularity supported by the technique is fine-grained (Q2), because it allows managing variability in the level of field in data types and that is adherent to the variability space in Hephaestus that is mostly fine-grained. Aspectual Caml allows features to be modularized (Q3) into aspects which provide good separation of cross-cutting concerns, since extensions related to a given variant feature are confined to a single module and such module only addresses extensions related to this feature. Regarding the maturity level (Q5), Aspectual Caml was already applied in the implementation of a prototype compiler of programming language. However, Aspectual Caml is not integrated into Haskell (Q6). It is then not possible to apply the technique to the current code of Hephaestus, which has been developed in Haskell. There are then two alternatives: to develop an extension of Haskell with the features of Aspectual Caml; to translate Hephaestus' code into Aspectual Caml. In either case, the impact on Hephaestus for the use of this technique is high (Q4).

B. CIDE

CIDE is an annotative approach to represent variability in different SPL artifacts. Using this environment we first create a feature model and relate each feature to a specific color. Then, we assign pieces of code to features, and the environment highlights that piece of code using the corresponding color [12]. After that, we could select a feature configuration and CIDE exports only the code that implements the mandatory and selected features.

CIDE implements variability in SPL without obfuscating code, as occurs when using preprocessor directives, the most common way of annotative approach. For this reason, some authors claim that CIDE employs a *virtual separation of concerns*, and simplified views of the code could be obtained.

1) Refactoring Hephaestus into Hephaestus-PL using CIDE: From the feature model of Figure 5, we started by coloring the pieces of code related to each option of the *SPL Models* and *Output Formats* features (see Figure 8). More specifically, we colored the optional lines related to the *SPLModel* and *InstanceModel* data types, optional lines of the configuration knowledge XML parser, and the optional lines of the export function.

2) CIDE Evaluation: Regarding Q1, CIDE has enough expressiveness to support all variability types in Hephaestus. CIDE supports variability in data types, functions and the composition of assets is guaranteed by the process of generating a variant that represents an instance of product which combines any assets of the SPL Feature Model. Given its annotative nature, CIDE supports both fine- and coarse-grained variability (Q2).

As explained earlier, CIDE provides weak separation of concerns. For this reason, we are not able to separate features' projections into distinct modules (Q3 is not supported by

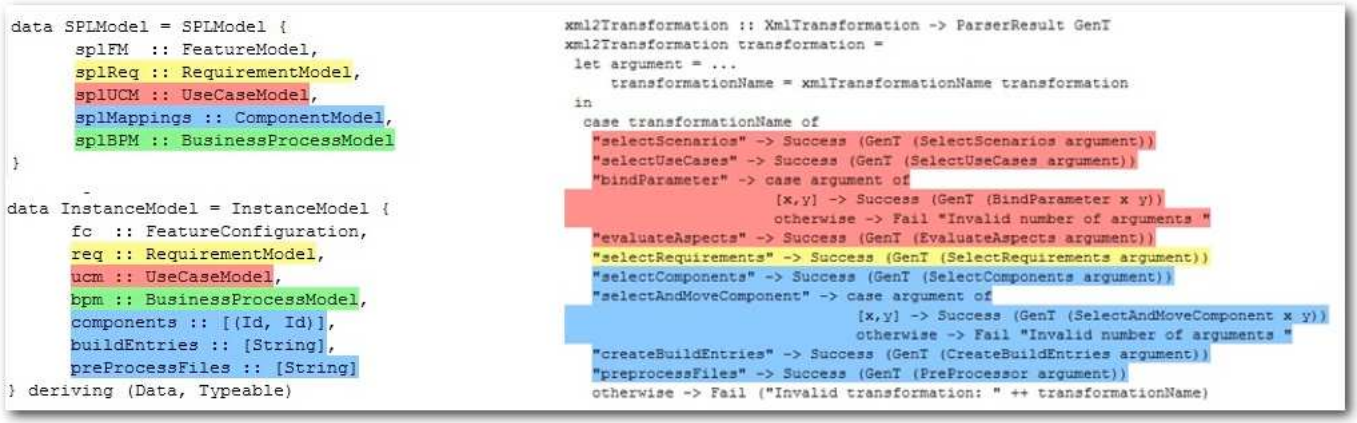

Fig. 8. Piece of code of Hephaestus annotated by CIDE.

CIDE). Although we could visualize source code corresponding to a specific feature selection, the feature implementation preserves some degree of scattering and tangling. We could figure this out by observing the different colors on specific data types and functions on Figure 8. For instance, using CIDE, fields related to different assets are still declared in the same *SPLModel* and *InstanceModel* data types. It could bring some modularity issues when evolving a product line, because in order to introduce a new SPL asset, we have to change existing modules in Hephaestus. CIDE then does not support the open-closed principle, that recommends software design that is open to extensions and closed to modifications. However, this shortcoming is not a critical issue in Hephaestus because the degree of scattering and tangling of features is considerably small and confined to a constant and small number of modules to be affected by the addition of new features.

We rank as average the effort to use the technique to the current code of Hephaestus (Q4). Considering the artifacts to be annotated, the activity performed coloring the pieces of code associated with features, is not too time demanding. Furthermore, the effort is minimized and it becomes low when the activity is supported by the use of CIDE+, an extension of CIDE that allows semi-automatic annotation (coloring) of pieces of code representing the features through the definition of the feature seed. Regarding the maturity level of CIDE (Q5), there is a myriad of publications on the use of CIDE, e.g., programming languages, database case studies, operating systems. Usage of CIDE+ has been reported in bootstrapping a SPL of CASE tools [20]. Because CIDE supports Haskell it was possible to refactor Hephaestus using CIDE, thereby being evaluated positively regarding Q6.

C. Transkell

Transkell is a *domain specific language* that aims to manage Hephaestus variability [16]. Actually, Transkell was the first attempt to evolve Hephaestus into a product line, even though its current implementation does not support all variability described in this paper and it was designed to simplify the process of introducing new types of transformations. Here we follow a different perspective, since the SPL assets are our main decomposition strategy.

1) Refactoring Hephaestus into Hephaestus-PL using Transkell: To create a specific version of Hephaestus using this approach, product engineers write a program in the Transkell language and then, applying source to source transformations implemented in Stratego/XT [15], translate this source Transkell program into Haskell code.

A Transkell program comprises a set of *transformations*, where each transformation specifies what it needs to run. For instance, a transformation for selecting source code assets must specify the expected fields of the SPL Model and Instance Model data types, the expected parsers, the expected steps of the export function, and so on. Figure 9 shows a concrete example of a transformation in Transkell.

```
Transformation SelectComponents {
  BasicType { ... }
  Input { ... }
  Output {
    Import(...)
    Code {
      exportUcmToLatex f ucm = ...
      exportUcmToXML f ucm = ...
    }
    ExportCode { ...
     exportUcmToLatex ... (ucm product)
     exportUcmToXML ... (ucm product)
    }
  }
  SPL {
   ...
   type {
     ucm = UseCaseModel
   }
  }
}
```

Fig. 9. Transkell code for implementing the *selectComponents* transformation.

2) Transkell evaluation: Based on the current implementation of Transkell, we realize that a transformational based,

domain specific language implemented on top of Stratego/XT is able to manage all Hephaestus variability (Q1), solving both coarse-grained as well fine-grained variability (Q2).

Moreover, the design decisions of Transkell present some degree of modularity (Q3), since developers could describe everything related to a transformation within a single construct of the Transkell language. However, we consider that implementing a domain specific language for Hephaestus using a SPL model (requirements, business process, ...) as the main decomposition concern, instead of transformations, could lead to a more modular design. The main reason is that, based on the current Transkell design decisions, whenever two transformations (T_1 and T_2) require similar contributions to the final product we have to either duplicate code or create dependencies between the transformations T_1 and T_2.

Regarding Q4, to bootstrap a product line from the existing Hephaestus code we have to rewrite all Haskell modules that present some variability using the Transkell language. Therefore, in order to refactor Hephaestus into a product line using a DSL, a substantial effort is necessary. Nevertheless, Marcos and Borba report on the maturity of Stratego/XT tools (including Spoofax) that make easier (Q5) the construction of the Transkell language [16] and the source to source transformations between Transkell and Haskell (Q6).

D. Type classes

Creating generic Hephaestus transformations, for instance a generic *select asset* transformation, is the main idea of using type classes to solve product line variability in Hephaestus. In its basic form of usage, a type class allows developers to define functions that are applied to some, but not all types.

Advanced usage of type classes [21] has been used to implement open functions and open data types in functional languages [22], two classes of variabilities found in Hephaestus (see Section II).

1) Refactoring Hephaestus into Hephaestus-PL using type classes: Generalizing Hephaestus transformations using type classes involves two steps. First, the definition of type classes to allow the implementation of generic transformations (such as *select asset*, *bind parameter*, and *evaluate advice*). Note that transformations for asset selection are available to all SPL models (requirements, use case scenarios, business processes models, components, and so on); whereas similar implementations for binding parameters and evaluating advices are only available to use case scenarios and business processes models. Second, we have to specify which type classes a given SPL model is an instance. For example, the use case model must be an instance of all type classes required by the generic transformations *select asset*, *bind parameter*, and *evaluate advice*; while the component model must be an instance of the *select asset* type class.

This solution uses advanced type mechanisms, as it is possible to realize observing the piece of code on Figure 10, which declares three type classes for generalizing the *select asset* transformation. First, it declares a type class Id for values that could be used to identify assets. This is required because

```
class Eq x ⇒ Id x
class Id i ⇒ Identifiable x i | x → i
where
  identify :: x → i
class (Eq y, Identifiable y i) ⇒ Composite x y i | x → y i
where
  toComponents :: x → [y]
  replaceComponents :: ([y] → [y]) → x → x
selectAssets ::
  (Composite (f a) c i,
    Composite (g a) c i,
    Composite a c i,
    Identifiable c i, Id i
  ) ⇒ [i] → f a → g a → g a
selectComponents ids spl product = ...
```

Fig. 10. Type class hierarchy for generalizing the *select asset* transformation.

the type of a value used to identify assets might vary according to a specific SPL model. An instance of the Id type class must also be an instance of the Eq type class (see the constraint $Eq \Rightarrow Id\ x$). Moreover, we declared an $Identifiable$ type class that declares a function to get the identity of an element (in our case, an SPL asset). This class is parameterized according to the type of an identifiable element x and the type of its identity i, where i must be an instance of the Id type class. With this we establish a type relation between identifiable elements and identities. Finally, a $Composite$ type class was declared to get the components of an asset (for instance, the requirements of a requirement model) and to replace somehow the components of an asset.

Based on these type classes, we are able to implement a generic $selectAssets$ functions, that expects three parameters:

- A list of identities $[i]$.
- An SPL asset $f\ a$, which could comprise a use case model, a requirement model, and so on.
- A product asset $g\ a$, which could comprise a use case model, a requirement model, and so on. But notice that, if the second argument comprise a use case model, the resulting product must also comprise a use case model (both $f\ a$ and $g\ a$ refers to the same type parameter a).

As explained, after declaring the required type classes and generic transformations, we have to define that specific assets must be instances of the $Composite$ type class, respecting all restrictions and providing implementations to both $toComponents$ and $replaceComponents$ functions of the $Composite$ type class.

2) Type classes Evaluation: After the initial effort to generalize some of Hephaestus transformations, we realized that using type classes alone we would not be able to derive some instances of Hephaestus-PL. For example, using the mechanism explored here, we are able to derive products that support variability in a single type of asset. This is not a huge limitation, since we believe that most of Hephaestus users are interesting in managing variabilities in specific models. Never-

theless, such a solution does not attend the need of the TaRGeT team that needs to manage variabilities in requirements, use cases, and components models. Therefore, with respect to the expressiveness of the technique (Q1), our initial design using type classes does not fulfill the Hephaestus-PL requirements.

With respect to the granularity level (Q2), the use of type classes as described here supports both coarse and fine grained variabilities in a modular way (Q3), since we are able to modify field types of the *SPLModel* and *IntanceModel* data types by declaring new instances of the existing type classes. Nevertheless, to introduce variability support into the *export* and *xml2transformation*, probably we have to either define additional type classes to this specific purpose or combine type classes with monads, something that requires a deeper investigation and is a matter of future work.

The effort (Q4) to refactory Hephaestus into Hephaestus-PL using type classes is high, since most of the existing code described here have to be rewritten. In addition, the use of this technique requires a deeper knowledge on type classes and type systems, not being easily acessible to some developers that contribute to Hephaestus. Nevertheless, type classes have been deeply discussed (Q5) in the functional language comunity with many applications described— mostly related to the open function and open data type dilemma. The technique is fully implemented in the Glasgow Haskell Compiler using some type extensions of Haskell.

E. Synthesis Techniques

In this section, we present an overview of the techniques with respect to the GQM model proposed and discuss the most appropriate techniques for the design and implementation of Hephaestus-PL considering the existing Hephaestus code. Table II represents the overview of the assessment of each technique by each metric of the GQM model.

Regarding expressiveness of the techniques to handle the variability types in Hephaestus, we observe that all techniques meet this criterion, except for Type Parametrization technique that does not support open functions and composition of assets. In Hephaestus, the variability space is mostly fine-grained, and to this criterion, all the techniques could be applied to the development of Hephaestus-PL. However, the requirement to support modularization when handling addition of new assets and their variability is not fulfilled only by CIDE, which is an annotative approach. In general, the level of impact on existing Hephaestus code to the application of the technique is considered low when the technique is supported by Haskell, as with CIDE and Type Parameterization techniques. For the Transkell technique, the impact is high because we have to rewrite all Haskell modules that present some variability using the Transkell language. Aspectual Caml technique also has high impact, because this technique does not support Haskell language. Referring to the maturity level of the techniques, they all satisfy because there are several works already implemented. Finally, almost all presented techniques support the Haskell functional language where it was implemented the existing Hephaestus code, except the Aspectual Caml technique which supports only the Objective Caml functional language.

Overall, Aspectual Caml and Type Parameterization techniques were the techniques with more unfavorable measures (in this case, two measures), regarding the requirements for the design and development of Hephaestus-PL. On the other hand, we identified that CIDE and Transkell techniques are best suited to the development of Hephaestus-PL from the existing Hephaestus code. CIDE, despite being annotative and not allowing the modularization of variability of new assets, is still appropriate because the level of scattering and tangling of the variabilities is low in Hephaestus. The Transkell technique, despite having been evaluated with high impact (M4) to the existing Hephaestus code, according to section IV.C.2, Marcos and Borba report on the maturity of Stratego/XT tools that make easier the construction of the Transkell language and the source to source transformations between Transkell and Haskell.

Regarding completeness, we note that no single technique was able to address positively all metrics. Therefore, a possible strategy is combining approaches to address all such metrics. In particular, combining the two best ranked approaches, i.e., CIDE and Transkell achieves completeness. This is viable because of the immediate availability of these tools for Haskell. Additionally, in order to address CIDE's shortcoming in modularity, artifacts with tangling and scattering could be handled by Transkell transformations; there are few of such artifacts, so that the use of Transkell is not too demanding, thus minimizing the impact of its downside regarding M4. For other such artifacts, e.g., with medium- or coarse-grained granularity, CIDE or CIDE+ could be applied, relying on their low effort of application without compromising modularity. The implementation of this strategy could be bootstrapped from Hephaestus itself, leveraging the *build* and supporting functions and data types as well some transformations including those dealing with annotative variability and selection of artifacts. We regard the implementation of this strategy as future work.

V. Related Work

Different SPL design and implementation techniques have been compared [23], [24], [25]. Nevertheless, these have not been accomplished in the particular context of SPL tool development as in this work. Previous work [26] has considered how a compositional technique, feature-oriented programming implemented in AHEAD, could bootstrap the tool itself, thus potentially leading to a family of AHEAD-like tools. However, this has not been explored to handle variability in different artifacts as presented here. On the other hand, CIDE and CIDE+, annotative approaches, can handle variability in different artifacts. However, differently from our work, neither this nor none of the previously mentioned approaches handle combination of different artifacts to address or-features (Figure 5). Similarly to this work, Apel [27] et al. have compared an annotative versus compositional approach for modularizing features. According to their study, each brings relative benefits

Approaches			Annotative	Transformational	Compositional	Refactoring
Techniques			CIDE	Transkell	Aspectual Caml	Type Classes
Goal	Question	Metric				
G1	Q1	M1.1	Yes	Yes	Yes	Yes
		M1.2	Yes	Yes	Yes	No
		M1.3	Yes	Yes	Yes	Yes
		M1.4	Yes	Yes	Yes	No
	Q2	M2	Fine and coarse-granularity	Fine and coarse-granularity	Fine-granularity	Fine-granularity
	Q3	M3	No	Yes	Yes	Yes
	Q4	M4	Medium / Low (CIDEPlus)	High	High	High
	Q5	M5.1	DB; OS; CASE tools (ArgoUML); CIDEPlus	several	compiler	several
		M5.2	software engineering	several	programming languages	several
	Q6	M6	Yes	Yes	No	Yes

TABLE II
SUMMARY OF THE EVALUATION OF EACH TECHNIQUE

and liabilities and they have concluded that a combination of the approaches is best suited for feature modularization. The synergetic approach is similar to our conclusion, but we have considered additional approaches (transformational and parametric) and we concluded that the annotative approach is best combined with the transformational one. In addition to Hephaestus, there is a plethora of other SPL derivation tools (e.g., pure::variants, GenARch, and Gears), but as pointed out elsewhere [4] most tools lack flexibility and adaptability.

VI. Final remarks

We have conducted a comparative analysis of variability management techniques for SPL tool development in the context of the Hephaestus tool. The analysis reveals that two techniques, one annotative (CIDE) and another transformational (Transkell), are most suitable to variability management in Hephaestus, and that their combination is a feasible strategy to improve such management. Although the comparative study was conceived in the context of only one tool, we believe that the analysis could be useful for the development of other tools, since these should have similar abstractions to Hephaestus' data types and functions with have similar variability issues, to be addressed by the discussed techniques. Nevertheless, further empirical work is necessary to address this external validity threat. As future work, we are planning to implement the strategy described in Section IV-E for bootstrapping Hephaestus-PL. The implementation will focus on reuse and generalization of the transformations and on assuring type safety of the instantiated products in a scalable way. We also plan to conduct further empirical studies assessing Hephaestus-PL's evolution to handle variability in different kind of artifacts.

References

[1] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[2] M. L. Griss, "Implementing product-line features with component reuse," in *ICSR*, London, UK, 2000, pp. 137–152.

[3] S. Deelstra, M. Sinnema, and J. Bosch, "Product derivation in software product families: a case study," *JSS*, vol. 74, no. 2, pp. 173–194, 2005.

[4] R. Rabiser, P. Grünbacher, and D. Dhungana, "Requirements for product derivation support: Results from a systematic literature review and an expert survey," *Inf. Softw. Technol.*, vol. 52, pp. 324–346, March 2010.

[5] P. Grünbacher, R. Rabiser, and D. Dhungana, "Product line tools are product lines too: Lessons learned from developing a tool suite," in *ASE*, 2008, pp. 351–354.

[6] R. Bonifácio, L. Teixeira, and P. Borba, "Hephaestus: A tool for managing SPL variabilities," in *SBCARS Tools Session*, 2009, a draft copy is available at: http://bit.ly/notgBg.

[7] R. Bonifácio and P. Borba, "Modeling scenario variability as crosscutting mechanisms," in *AOSD '09*. ACM, 2009, pp. 125–136.

[8] F. Ferreira, L. Neves, M. Silva, and B. Paulo, "Target: a model based product line testing tool," in *CBSoft 2010 Tools Session*, 2010, pp. 1–4.

[9] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[10] I. Machado, R. Bonifácio, V. Alves, L. Turnes, and G. Machado, "Managing variability in business processes: an aspect-oriented approach," in *EA '11*. ACM, 2011, pp. 25–30.

[11] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*. Wiley, 1994.

[12] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *ICSE '08*, 2008, pp. 311–320.

[13] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. S. Batory, "Guaranteeing syntactic correctness for all product line variants: A language-independent approach," in *TOOLS (47)*, 2009, pp. 175–194.

[14] V. Borges, R. Garcia, and M. T. Valente, "Cide +: A tool for semi-automatic extraction of software product lines using coloring code (in portuguese)," in *SBSOFT 2010 Tools Session*, 2010, pp. 73–78.

[15] E. Visser, "Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9," in *LNCS*, vol. 3016. Springer, 2003, pp. 216–238.

[16] M. A. Xavier and P. Borba, "A DSL specification for implementing SPL variability," Recife, PE, Brazil, 2010.

[17] M. Mezini and K. Ostermann, "Variability management with feature-oriented programming and aspects," in *FSE*, 2004, pp. 127–136.

[18] H. Masuhara, H. Tatsuzawa, and A. Yonezawa, "Aspectual caml: an aspect-oriented functional language," *SIGPLAN Not.*, vol. 40, pp. 320–330, September 2005.

[19] M. Wang and B. C. d. S. Oliveira, "What does aspect-oriented programming mean for functional programmers?" in *WGP '09*. ACM, 2009, pp. 37–48.

[20] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *CSMR*, 2011, pp. 191–200.

[21] M. Jones, "Functional programming with overloading and higher-order polymorphism," *Advanced Functional Programming*, pp. 97–136, 1995.

[22] R. Lämmel and K. Ostermann, "Software extension and integration with type classes," in *GPCE '06*, 2006, pp. 161–170.

[23] M. Anastasopoulos and C. Gacek, "Implementing product line variabilities," in *Symposium on Software Reusability*. ACM Press, May 2001.

[24] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques," *Softw., Pract. Exper.*, vol. 35, no. 8, pp. 705–754, 2005.

[25] R. E. Lopez-Herrejon, D. S. Batory, and W. R. Cook, "Evaluating support for features in advanced modularization technologies," in *ECOOP*, 2005, pp. 169–194.

[26] D. S. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," in *ICSE*, 2003, pp. 187–197.

[27] S. Apel *et al.*, "Feature (de)composition in functional programming," in *SC '09*, 2009, pp. 9–26.