Configurability and Extensibility of Tools for Software Product Lines

Lucinéia Turnes\textsuperscript{a}, Vander Alves\textsuperscript{a,*}, Rodrigo Bonifácio\textsuperscript{a}, Ralf Lämmel\textsuperscript{b},

\textsuperscript{a}Departamento de Ciência da Computação, Universidade de Brasília, Campus Darcy Ribeiro, Brasília, Brazil, 70910-900
\textsuperscript{b}Software Languages Team, Universität Koblenz-Landau, Institut für Informatik, Universitätsstraße 1, D-56070 Koblenz, Germany

Abstract

Tool support is essential for application engineering in software product lines for the derivation of products from reusable assets. Existing tools are insufficiently configurable and extensible with regard to the type of artifacts and associated variability mechanisms that they target. This paper addresses the problem by exploring the underlying commonality and adequately managing variability of tools for product derivation. More specifically, the paper describes domain analysis, design, implementation of Hephaestus-PL—a proper product line for such tools. To this end, an extractive strategy was applied to the existing Haskell-based Hephaestus tool. A transformational approach to variability management is designed and implemented by metaprogramming in Haskell. We also describe a reactive process for evolving Hephaestus-PL to address new types of artifacts. Improved configurability and extensibility is substantiated by an assessment.

Keywords: Software Product Line Tools, Variability Management, Hephaestus, Configurability, Extensibility, Bootstrapping, Expression Problem, Metaprogramming, Haskell

*Corresponding author. Telephone: +55 61 8157-1464. Fax: +55 61 3107-7414.
Email addresses: lucineiaturnes@gmail.com (Lucinéia Turnes), rbonifacio@cic.unb.br (Rodrigo Bonifácio), valves@unb.br (Rodrigo Bonifácio), rlaemmel@acm.org (Ralf Lämmel)
1. 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 benefits, tool support for the 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 parts of different components are combined. Given the inherent complexity of SPLs and the coordination required for instantiation, the process of derivation of individual products from shared software assets should be supported by tools: \textit{product derivation tools}. Without such tools, the derivation process is a laborious, error-prone, and thus, expensive activity [2, 3].

The changing needs of users and the continuous evolution of product lines motivates configurability and extensibility of product derivation tools for addressing future needs. As reported by a contemporary Systematic Literature Review and expert survey [4], key requirements of product derivation tools are indeed configurability and extensibility (in fact, ‘flexibility’ in the terminology of [4]) which the same study identifies as shortcoming of most existing tools. The present paper addresses such configurability and extensibility. As an illustration of insufficient configurability, consider a derivation tool that targets different types of artifacts without supporting, though, the selection of types for a specific use case of the tool. For instance, different variants of the Haskell-based \textit{Hephaestus} tool [5] for product derivation target use case models, requirement models, source code, and yet other types of artifacts, but the selection of only some of the types is not possible for a given variant of \textit{Hephaestus}. The present paper studies \textit{Hephaestus} in detail and improves it ultimately. As an illustration of insufficient extensibility, consider a derivation tool that needs to target an additional type of artifact without providing, though, any extension scheme and process to this end. For instance, \textit{Hephaestus} had to target architectural models at some point, but this extension could only be completed in a low-level (i.e., laborious and error-prone) manner.

The requested configurability and extensibility may be obtainable, if the variability within tools for product derivation is managed by architecting the tools themselves as SPLs, as suggested by Grünbacher et al. [6] and also
exemplified by some efforts [7, 8]. Overall, the major contribution of the present paper is the description of the details of the engineering process for obtaining and enhancing such a meta-product line, while using the Haskell-based Hephaestus tool for product derivation for a comprehensive study.

Contributions of the Paper

- We describe an extractive process that enhances the existing Haskell-based tool Hephaestus for deriving products from a SPL into a Haskell-based proper product line, Hephaestus-PL, for product derivation tools. In particular, we describe domain analysis, design, and implementation [9] of Hephaestus-PL. Configuration is expressive; specifically, or-features (allowing for combinations) are supported as opposed to just alternative features (allowing only for exclusive choices). Hephaestus-PL employs a general transformational approach to variable management. In this manner, the present paper provides evidence for the feasibility of configurable and extensible tools for product derivation.

- We describe the technicalities and implications of providing a meta-SPL in Haskell. Hephaestus-PL’s transformational approach is implemented through metaprogramming in Haskell. Technically, we had to solve an Hephaestus-specific variation on the ‘Expression Problem’ [10, 11] within Haskell. The approach enables modular code organization for all components of Hephaestus-PL, separate compilation within limits, and bootstrapping of Hephaestus-PL, thereby achieving high uniformity across the regular SPL level and the meta-SPL level.

- We describe a reactive process for evolving Hephaestus-PL for modeling variability for new types of artifacts so that product line tools can be derived for these new types. An assessment reveals that Hephaestus-PL has improved configurability and extensibility when compared to previous variants of Hephaestus. In this manner, the present paper provides evidence for the effectiveness of addressing the problem of configurable and extensible tools for product derivation.

We believe that these contributions and experiences could be leveraged in other contexts in which configurability and extensibility of tools for product derivation are key requirements.
Road-map of the Paper

The remainder of this paper is organized as follows. First, Section 2 briefly describes Hephaestus and it analyzes a representative evolution scenario for addressing an additional type of artifacts. Next, based on experience with variants and evolution history of Hephaestus, Section 3–5 describe domain analysis, domain design, and implementation of Hephaestus-PL. Section 6 describes a reactive process for extending Hephaestus-PL. Section 7 provides an assessment and discussion of Hephaestus-PL. Related work is addressed in Section 8. The paper is concluded in Section 9.

The Haskell source code of Hephaestus-PL is publically available. For the rest of the paper, ‘Haskell’ refers to standardized Haskell in sense of Haskell 2012, unless noted otherwise.

2. Hephaestus

Hephaestus [5] is a publicly 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 [12], Hephaestus provides a declarative and executable specification in Haskell of a compositional [13] and parametric approach for managing variability in use case scenarios. Currently, Hephaestus supports variability in different types of artifacts, ranging from business processes and Simulink models to source code; and it has been used as the derivation tool in a industrial-strength product line [14].

2.1. Initial Hephaestus

For the initial purpose of the tool, the implementation contained the following entities:

- Specific data types representing use case models (UCM), feature models (FM), and configuration knowledge (CK) model [9], which relates feature expressions in propositional logic to transformations that deal with variability in use cases.

1https://gitorious.org/hephaestus-pl/hephaestus-pl
2http://www.haskell.org/onlinereport/haskell2010/
3https://gitorious.org/hephaestus/pages/Home
Specific functions that solve SPL variabilities in use case scenarios, by selecting use cases or scenarios from an SPL model, composing them, and binding parameters according to specific feature configurations.

A build function that serves as an interpreter for the configuration knowledge and is responsible for building a specific product from a given selection of features, i.e., a feature configuration (FC).

Specific functions providing functionality on use case models.

Figure 1 presents a code snippet of the initial implementation of Hephaestus including data types related to configuration knowledge (lines 1-5) and a corresponding interpreter (the build function, in lines 7-17), and the signature of the transformation functions (line 22). The configuration knowledge relates the problem space (through a feature expression) to the solution space (through a list of asset transformations). Each transformation solves some variation point in the SPL asset base. The build function has four input artifacts (FM, FC, CK, and SPL) and generates a product, i.e., an instance of the SPL. The build process evaluates the rows of the CK, validating each feature expression according to the FC—the set of features that characterizes a given product. If a feature expression of the CK is true for the given FC, then the corresponding transformations are applied to the emerging instance. In addition, the code snippet also shows the initial representation of the data types SPLModel (lines 24-27) and InstanceModel (lines 29-32). Both types collect assets and group them with the feature model or the feature configuration, respectively. The build process eliminates all variability from the SPLModel value and computes the InstanceModel value. The exportProduct function (lines 34-36) generates a \LaTeX\ representation of a product-specific use case model.

2.2. Evolution of Hephaestus
In the context of a R&D project, Hephaestus was used as a replacement for a proprietary tool that was used to manage variabilities in an industrial-strength product line [14]. In this context, Hephaestus had to be evolved so that it could manage variability not only in use case scenarios, but also in higher level requirements and source code. Variability in source code means here that source files can be selected for inclusion into the product, triggering compilation and preprocessing, thereby solve yet more variability within the
```haskell

```ConfigurationKnowledge = [ConfigurationItem]

data ConfigurationItem = ConfigurationItem {
    expression = FeatureExpression,
    transformations = [Transformation]
}

build :: FeatureModel -> ConfigurationKnowledge -> SPLModel -> InstanceModel
build fm fc ck spl = derive ts spl emptyInstance

where
    ts = concat [transformations c | c <- ck, eval fc (expression c)]
    ucmodel = ucmodel { useCases = [], aspects = [] }
    emptyInstance = InstanceModel fc emptyUCM

derive [] spl product = product
derive (t:ts) spl product = derive ts spl (t spl product)

has exported the source code. We show the configurations for both the initial release and the evolved releases in Figure 2 and Figure 3.
Evolution required that new data types and transformations were added and some of the existing code had to be changed. Precisely, to introduce support for variability in high-level requirements (or requirements for short) and source code, the following implementation and evolution steps were necessary:

(a) Implementation of new data types for representing requirements and references to source-code assets.
(b) Implementation of new parsing and output functions for reading/writing requirements and source-code assets into/from Hephaestus.
(c) Implementation of new transformations for resolving variabilities in requirements and source code.
(d) Evolution of the configuration knowledge XML parser so that it could recognize the concrete syntax of the new transformations.
(e) Evolution of the base product instance used by the `build` function.
(f) Evolution of both `SPLModel` and `InstanceModel` data types, to collect new assets.

The evolution of data types and functions of Hephaestus, as described above, is illustrated in Figure 4 and Figure 5. To introduce support for new
**Figure 4:** SPLModel and InstanceModel data types, emptyInstance definition and exportProduct function 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" → ...
    _ → Fail "..."

Figure 5: Code snippet for the XML parser for configuration knowledge

definition (lines 23-25), the exportProduct function (lines 27-31), and the xml2Transformation function of the XML parser for configuration knowledge (Figure 5).

In contrast, the data types ConfigurationKnowledge and FeatureModel as well as the build function do not require any revision, when we introduce variability support for new types of artifacts.

In particular, evolving Hephaestus to support source code variability (Figure 4) required a new type of asset in the SPLModel (splComponents, line 5). This asset is a list of pairs that relate a name to the relative path of a source code file. The same type of asset was also introduced into the InstanceModel (line 12). Besides that, two other fields were required in the InstanceModel: (a) buildEntries (line 13), which declares pre-processing directives, and (b) preProcessFiles (line 14), 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.

The code snippet in Figure 5 shows the revised XML parser for configuration knowledge. The initial version of Hephaestus declared just the first four case statements on the xml2Transformation function (lines 8-11). The added selectRequirements transformation (line 12) deals with variability in the requirements models, whereas the remaining transformations (lines
13-16) deal with variability in source code.

Likewise, *Hephaestus* also provides limited configurability: to obtain a new version of *Hephaestus* managing variability in only a proper subset of the types of artifacts currently supported (use cases, requirements, source code), e.g., a version supporting only source code and use cases, the change impact is similar to what has been described previously when extending *Hephaestus* to target new types of artifacts.

### 2.3. *Hephaestus*’s ‘Expression Problem’

Technically, the evolution scenario demonstrated an extensibility problem. We had to extend data types and functions to incorporate new types of artifacts, transformations, and support functionality (e.g., for exporting). We failed to provide a modular extension; instead, we ended up revising existing code.

The well-known ‘Expression Problem’ (EP [10, 11]) captures this sort of challenge in a principle manner: *Given a family of data variants and a family of operations on the data, how can we design an implementation of data and operations such that new variants and new operations can be added without revising existing code, while possibly also providing some degree of separate compilation, modular type checking, and modular reasoning?*

Programming language support for the EP, as discussed below, helps with modularization so that evolution scenarios, like the one discussed above, can be potentially carried out without revising existing code.

Hinze and Löh proposed a Haskell extension for open data types and functions for Haskell [15], which would provide a solution to the EP in Haskell. The extension allows adding new cases to data types (i.e., adding cases to ‘sums’, algebraically speaking) and new equations to functions (i.e., adding cases to a discrimination over a sum), but it does not deal with additional adaptation patterns encountered in Figure 4. Specifically, the evolution of *Hephaestus* involved extending ‘products’ as opposed to just ‘sums’ (see the data types *SPLModel* and *InstanceModel*) and ‘function compositions’ as opposed to just ‘case discriminations’ (see the the function *exportProduct*). Arguably, these adaptation patterns could be eliminated by using a different data and program design that treats the products instead as more generic containers, as we discuss in Section 5. In either case, open data types and functions are not available in actual Haskell systems.

Lammel and Ostermann proposed a type class-based encoding for solving the EP in Haskell with relatively established extensions (in fact, in Haskell 98
for the narrow EP) [16]. The encoding overhead is substantial. Data types like SPLModel and InstanceModel and functions like exportProduct and xml2Transformation would need to be represented as type classes, their constructors and equations would give rise to extra data types and type-class instances, leading to a substantial increase in code size and a negative impact on comprehensibility, e.g., due to more complex type-error messages. The approach also fails at the aforementioned adaptation patterns, which go beyond the EP. More subtly, the approach also fails at function extension scenarios that are not readily based on different constructors; see xml2Transformation, where all equations match on strings.

Ultimately, the domain design of Hephaestus-PL (see Section 4) leverages a transformational approach to managing variability with an implementation (see Section 5) based on Haskell metaprogramming. The approach readily covers all adaptation patterns, it directly works with existing Haskell systems, it easily integrates with feature modeling and configuration, and it enables bootstrapping of Hephaestus-PL.

3. Domain Analysis of Hephaestus-PL

This section introduces the domain analysis [9] of Hephaestus-PL. There exist different Hephaestus variants to provide variability for different types of artifacts. These variants were created by different parties in response to the needs for different combinations of types or additional types to be targeted. The Target variant of Hephaestus [14] deals use cases (relying on structured language and supporting test generation), requirements, and source code. Newer variants of Hephaestus have targeted business process models [17] and Simulink assets [18].

All these variants share the same configurability and extensibility issues explained in Section 2. To address these issues, we adopt a SPL perspective to Hephaestus itself, thus leveraging the commonality in these variants and systematically managing the incurred variability. We adopt the extractive strategy [19], i.e., we extract Hephaestus-PL from the existing Hephaestus variants. Correspondingly, we analyzed the existing variants of Hephaestus and manually identified the common and variable features, architectural and implementation elements, as explained in rest of this section. In Section 4, we describe how Hephaestus-PL’s domain design leverages the domain analysis of the present section and how it addresses configurability and extensibility at the design level. The details of implementation are presented in Section 5.
Section 6 describes a reactive process needed to introduce support for managing variabilities for new types of artifacts.

### 3.1. **Hephaestus-PL’s Feature Model**

In terms of the problem space, **Hephaestus-PL**’s feature model is represented in Figure 6. As the diagram shows, the features *SPLAsset* and *OutputFormat* are mandatory. Further, these features are parents of *or-features* so that combination of models and output formats are supported, e.g., a given instance might comprise business processes and use cases and export both assets as XML files. Managing variabilities in such a combination of features is essential in **Hephaestus-PL** and has not been addressed elsewhere.
In addition to the feature diagram, Figure 6 shows some cross-tree constraints that must be satisfied for any valid feature configuration of Hephaestus-PL. To illustrate, Figure 2 and Figure 3 show two valid configurations of Hephaestus-PL, whereas Figure 7 shows an invalid configuration of Hephaestus-PL, in which the UcmToXML feature is selected, but the UseCase feature is not selected. In this case, the $UcmToXML \lor UcmToLatex \rightarrow UseCase$ constraint was violated, leading to an invalid feature configuration of Hephaestus-PL.

Thus, in the problem space, a specific configuration of Hephaestus is conceptually simple in that it is represented by a combination of a few or-features. However, in the solution space, a specific configuration implies more complexity because it require management of variability in several artifacts (algebraic data types, functions of different kinds, and models) at different levels of granularity (i.e., course-grained and fine-grained). As the change impact of the evolution scenario in Section 2.2 showed, the implementation of the features is somewhat scattered in Hephaestus’ source code.

3.2. Identification of Commonality

In the solution space, there exists significant amount of commonality among configurations. The domain analysis revealed commonality for these abstractions across the variants of the evolution history:

**Feature Model Representation** An algebraic data type represents the feature model of a product line.

**Product Configuration Representation** An algebraic data type represents a valid feature configuration.

**CK Representation** An algebraic data type represents the product line’s configuration knowledge.

**Interpreter for Product Instantiation** The build function performs the SPL instantiation by generating a product corresponding to a specific product line configuration.

3.3. Identification of Variability

The domain analysis revealed variability for these abstractions across the variants of the evolution history:
**Asset Representations**  Algebraic data types represent the abstract syntax of different SPL assets (such as use cases, business processes, and code).

**Asset Transformations**  Functions manipulate such artifacts, thereby resolving variability of SPL assets. Some transformations basically select a specific asset from the product line so that it is included into the product during derivation. Other transformations change the structure of an asset of the SPL in the final product.

**Asset Containers**  The `SPLModel` and `InstanceModel` algebraic data types comprise the set of SPL assets and group it with the feature model or feature configuration, respectively, of a given Hephaestus-PL instance.

**Asset Input**  Input functions for reading assets, possibly parsing the concrete syntax of an asset to the corresponding abstract syntax of Hephaestus-PL, i.e., the asset abstract data types. Input functions are typically invoked upon starting product derivation so that all assets are made available to the process through a `SPLModel` value.

**Asset Output**  Output functions for writing artifacts in different output formats. Output functions are typically invoked upon finishing product derivation so that the derived product is persisted.

**Empty Instance**  An expression defines the initial representation of a product during the product derivation activity. It is an instance of the `InstanceModel` data type and serves as a baseline that is successively refined by the `build` function.

**CK Parser**  The `xml2Transformation` function parses the XML representation of transformations into actual transformations (functions) on instance models for use in the configuration knowledge.

Domain analysis also substantiated that variability has a regular form that may be understood linguistically in terms of introduction of new abstractions (‘modules’) and extension of existing abstractions (data types and functions). For example, to introduce variability support for one new asset in Hephaestus, one needs to implement the data types for representing the abstract syntax of these models, implement the transformations for solving variability, implement the parser and output functions for reading/writing
these assets into/from Hephaestus; one also needs to extend some data types and functions of Hephaestus, as it was illustrated in Section 2.2.

4. Domain Design of Hephaestus-PL

This section introduces the domain design [9] of Hephaestus-PL. We begin with an architectural view (Section 4.1). Then, we describe Hephaestus-PL’s transformational approach to variability management (Section 4.2), which also relies on designated configuration knowledge (Section 4.3), and a derivation process for SPL tools (Section 4.4). Finally, bootstrapping of Hephaestus-PL is described (Section 4.5).

4.1. Hephaestus-PL’s Architecture

Hephaestus-PL’s architecture is depicted in Figure 8. Overall, Hephaestus-PL is componentized in a way that most parts are separately compilable and
unaffected by configuration or extension.

*Kernel* represents the basic abstractions necessary for the generation of Hephaestus-PL instances including the generation of Hephaestus-PL itself in a bootstrapping process. Within the kernel, component *Hephaestus Base* represents the commonality among all Hephaestus-PL instances (as identified in Section 3.2); it serves as a base for deriving all Hephaestus-PL instances. To this end, Hephaestus Base has variability points (as identified in Section 3.3) which are to be resolved by transformations defined in component *Hephaestus SPL Asset* in a derivation process driven by component *Hephaestus Product*. These components of the kernel had to be specifically designed and implemented for Hephaestus-PL. The kernel also hosts Feature Modeling and Configuration Knowledge: these components provide basic abstractions for the representation of feature models, feature configurations, configuration knowledge as well as associated support functionality, e.g., for verifying validity of feature configurations relative to a given feature model. These components of the kernel could be reused from Hephaestus and all Hephaestus-PL instances share them, as is.

*SPL Assets* represents the assets of Hephaestus-PL. Each such meta-level asset essentially corresponds to a type of artifact that can be targeted with an Hephaestus-PL instance including the corresponding (non-meta-level) asset base for the type and designated support for variability management and product derivation in the domain of the asset. Each meta-level asset is essentially packaged in a component that exposes abstractions corresponding to the variation points of Section 3.3: datatypes for the representation of assets and their transformation, functions for the interpretation of transformations, and other functionality related to Hephaestus-PL’s feature model, notably export functions.

4.2. Hephaestus-PL’s Transformational Approach

We adopt a transformational approach [20] to variability management. That is, the derivation of an Hephaestus-PL instance involves source-code transformation. The approach provides transformations to address the heterogeneity of the variability patterns observed in Section 2.2 without compromising modularity and comprehensibility of Hephaestus-PL, and thus configurability and extensibility, which may be issues for annotative approaches [13]. The approach is also designed for uniformity. That is, the Hephaestus approach to feature modeling, feature configuration, declaration and interpretation of configuration knowledge is adopted also at the meta-level. Hephaestus
often delegates some part of variability management to external tool support (e.g., for pre-processing or aspect weaving), in which case, the transformation in the configuration knowledge essentially trigger those external transformations. In contrast, Hephaestus-PL fully implements the meta-level transformations as metaprograms in Haskell on top of object programs in Haskell.

Configuration of Hephaestus-PL relies on transformations provided by a high-level API, which are implemented in terms of transformations of a low-level API, which, in turn, are implemented in terms of metaprogramming operations (with Haskell used both for object- and metaprograms). The design of the high-level API captures the variability as identified during the domain analysis (Section 3.3). The low-level API captures adaptation patterns as identified during the examination of the evolution scenario (Section 2.2).

Figure 9 illustrates the API layers, as they are part of Hephaestus-PL’s component Hephaestus SPL Asset in the Kernel. We refer to Section 5.4 for the description of the actual implementation.

The transformations of the high-level API are of specific interest from the point of view of domain design, as these transformations are directly used in the configuration of Hephaestus-PL. A description follows. (Some details specific to bootstrapping are deferred until Section 4.5.)

SelectBase selects Hephaestus Base, which represents the commonality of all Hephaestus-PL instances. Typically, this is the first transformation to be executed in the process of deriving an instance. The implementation of Hephaestus Base is given in Section 5.2.

BindProductName sets the module name of the Hephaestus-PL instance.

SelectAsset refines the product being derived with support for the selected asset, thereby enabling variability management for the associated type.
<table>
<thead>
<tr>
<th>Feature Expressions</th>
<th>Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>SelectBase</td>
</tr>
<tr>
<td>UseCase</td>
<td>SelectAsset &quot;Use Case&quot;</td>
</tr>
<tr>
<td>UseCase AND UcmToXML</td>
<td>SelectExport &quot;UcmToXML&quot;</td>
</tr>
<tr>
<td>UseCase AND UcmToLatex</td>
<td>SelectExport &quot;UcmToLatex&quot;</td>
</tr>
<tr>
<td>BusinessProcess</td>
<td>SelectAsset &quot;Business Process&quot;</td>
</tr>
<tr>
<td>BusinessProcess AND BpmToXML</td>
<td>SelectExport &quot;BpmToXML&quot;</td>
</tr>
<tr>
<td>Requirement</td>
<td>SelectAsset &quot;Requirement&quot;</td>
</tr>
<tr>
<td>Requirement AND ReqToLatex</td>
<td>SelectExport &quot;ReqToLatex&quot;</td>
</tr>
<tr>
<td>Code</td>
<td>SelectAsset &quot;Code&quot;</td>
</tr>
<tr>
<td>Code AND BuildFile</td>
<td>SelectExport &quot;BuildFile&quot;</td>
</tr>
</tbody>
</table>

Table 1: Hephaestus-PL’s configuration knowledge

of artifacts. Technically, the transformation extends datatypes and functions for Asset Representations, Asset Transformations, Asset Input, Asset Containers, Empty Instance, and CK Parser. To this end, existing extension points are connected with asset-specific abstractions and the underlying modules are added to the product.

SelectExport refines the product being derived with support for the selected couple of output format and asset. To this end, the transformation extends datatypes and functions for Asset Output in a manner very similar to SelectAsset.

4.3. Hephaestus-PL’s Configuration Knowledge

Table 1 shows Hephaestus-PL’s configuration knowledge, without some details related to bootstrapping, which are deferred to Section 4.5.

4.4. Hephaestus-PL’s Derivation Process

The derivation process evaluates the condition for each line of CK for the given feature configuration. In this manner, transformations with true conditions are collected; they are applied consecutively in the order, as they appear in the table, with the Empty Instance serving as input for the first transformation and the final product corresponding to the output of the last transformation.
Figure 10: An illustrative Hephaestus-PL’s feature configuration

Figure 10 shows a feature configuration which we use for the illustration of the derivation process. The configuration selects four features: use case models (UseCase), business process models (BusinessProcess), and both kinds models in XML format (UcmToXML and BpmToXML).

Figure 11 abstractly depicts the steps taken in Hephaestus-PL’s product derivation process for the feature configuration of Figure 10. Hephaestus-PL’s CK guides this transformational process in five steps towards the generation of a Hephaestus-PL instance from the base product.

Hephaestus-PL’s product derivation process begins with the execution of the SelectBase transformation associated with the feature expression True in the first line of Hephaestus-PL’s CK (Table ??); see the first step in Figure 11. Next, the UseCase feature expression in the second CK line evaluates to true for the feature configuration at hand and thus the SelectAsset "UseCase" transformation is executed; see the second step in Figure 11. In this manner, the UCM Asset is incorporated into the product. Next, the UseCase AND UcmToXML feature expression in the third CK line evaluates to true and thus the SelectExport "UcmToXML" transformation is executed; see the third step in Figure 11. In this manner, Asset Output for the UCM Asset is incorporated into the product. Another two steps handle the BPM Asset very similar to the UCM Asset.

4.5. Bootstrapping of Hephaestus-PL

Consider again Hephaestus-PL’s architecture in Figure 8. Hephaestus SPL Asset in the Kernel provides datatypes and functions for representing and transforming Hephaestus-PL products, i.e., tools for product derivation—in the same way as the components in the Asset SPL provide datatypes and
Figure 11: Derivation of an Hephaestus-PL instance supporting the UCM, BPM, Ucm-ToXML and BpmToXML features

<table>
<thead>
<tr>
<th>Feature Expressions</th>
<th>Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hephaestus</td>
<td>BindProductName &quot;Hephaestus&quot;</td>
</tr>
<tr>
<td>Hephaestus</td>
<td>SelectAsset &quot;Hephaestus&quot;</td>
</tr>
<tr>
<td>Hephaestus</td>
<td>RemoveProductMainFunction</td>
</tr>
<tr>
<td>NOT Hephaestus</td>
<td>SelectCKParser</td>
</tr>
</tbody>
</table>

Table 2: Lines of Hephaestus-PL’s CK related to Hephaestus feature

functions for regular assets. Thus, bootstrapping of Hephaestus-PL, specifically derivation of Hephaestus Product, is relatively straightforward, as we discuss now.

Hephaestus-PL’s feature model contains a designated feature Hephaestus,
which is selected for bootstrapping. Hephaestus-PL’s configuration knowledge handles the feature in the manner shown in Table 2. The first two lines of CK directly model that bootstrapping builds Hephaestus Product from Hephaestus SPL Asset. That is, when the Hephaestus feature is selected, the product name is set to be “Hephaestus” and the “Hephaestus” asset (i.e., Hephaestus SPL Asset in the architecture) is selected.

The last two lines are somewhat more idiosyncratic. The third lines models that Hephaestus Product uses a different (in fact, a much simpler) main function than other products. To this end, an extra transformation RemoveProductMainFunction is invoked, with the simple intended semantics of removing the main for the product being derived. The fourth line models that CK parser is only needed if features other than Hephaestus are selected. To this end, an extra transformation SelectCKParser is invoked, with the intended semantics of enhancing the product being derived such configuration knowledge is parsed, and passed as an argument to the build function. Hephaestus Product does not need CK Parser because its configuration knowledge is defined as an abstraction as part of Hephaestus SPL Asset; thus, parsing is not needed.

5. Implementation of Hephaestus-PL

In this section, we present the important details of the implementation of Hephaestus-PL’s architecture. That is, we focus on implementation aspects of managing variability without discussing the modules specific to the different kinds of artifacts (use cases, requirements, etc.). We also describe Hephaestus-PL’s metaprogramming support in detail.

5.1. Modules and Dependencies

The modular structure of the implementation is shown in Figure ??; module dependencies are shown as well. The dotted line comprises the modules which compose each of the main elements in the Hephaestus-PL’s architecture, e.g., the HephaestusBase.hs and BaseType.hs modules constitute Hephaestus Base of Hephaestus-PL’s Kernel.

At the domain level, the IO.hs module contains the buildHpl function which represents the interface to the Hephaestus-PL’s instance level. This function is the initial point for the generation of a new Hephaestus-PL instance by the Hephaestus-PL’s product derivation process. The buildHpl function receives a feature configuration of Hephaestus-PL’s FM as input and prepares
the environment to execute the generation of Hephaestus-PL instance (guided by build function of Hephaestus Product). The inputs to the derivation process of Hephaestus-PL are four: a instance of the SPLModel data type, a valid feature configuration and the Hephaestus-PL’s FM and CK. In this context, the instance of the SPLModel data type wraps the Hephaestus asset, i.e., the physical modules of the Hephaestus-PL base product are inserted into an instance of HephaestusModel data type that represents the abstract type of Hephaestus asset and it is wrapped into an SPLModel data type. Besides, the derivation process of Hephaestus-PL needs the Hephaestus-PL’s feature modeling and configuration knowledge to execute the build function of the Hephaestus Product that controls the generation of the new Hephaestus-PL’s instance. The Hephaestus-PL’s FM and CK are contained into MetaData.hs module. Then, the build function contained into Hephaestus.hs module

Figure 12: Hephaestus-PL’s modules dependency
(Hephaestus Product) is executed and returns the new Hephaestus-PL tool instance. Therefore, the I0.hs module depends of Base Product’s modules, Hephaestus Product’s modules and MetaData.hs module, as shows the Figure 12.

The Hephaestus Product’s modules depend strongly on Hephaestus SPL Asset’s modules that implement the HephaestusModel and HephaestusTransformation data types and transformHpl function, i.e., the build function evaluates the CK and calls the transform function defined in the same Hephaestus.hs module into Hephaestus Product that executes the transformHpl function into Hephaestus.hs module of Hephaestus SPL Asset to solve the Hephaestus-PL transformation.

The MetaData.hs, MetaDataTypes.hs and MetaProgramming.hs modules comprise the group called Low Level API Support that implement the metaprogramming operations to solve the variation points in Base Product’s modules. The MetaProgramming.hs module depends the Language.Haskell.Syntax package.

The SPL Assets are composed by pairs of modules: the Types.hs module which defines the algebraic data types of the asset to insert into SPLModel and InstanceModel data types of Hephaestus-PL instance and defines the data types which are used as constructors into TransformationModel and ExportModel data types of BaseTypes.hs module. Besides, another module (assetName.hs) of each group in SPL Assets defines the functions which are used into transform, mkEmptyInstance and export functions of BaseProduct.hs module. Thus, the SPL Assets modules depend the Base Product’s modules.

Finally, we have the Core Assets’ modules (FM and CK) that are imported by Base Product’s modules.

5.2. Hephaestus Base

Figure 13 presents a snippet of the Haskell module for the Hephaestus Base with explanations for the definitions as follows:

- declares basic representations for the SPLModel (lines 3-5) and InstanceModel (lines 7-9) data types, where the first has just the featureModel data field and the InstanceModel data type has just the featureConfiguration data field,
- declares TransformationModel data type (line 11) which just supports the UndefinedTransformation constructor,
module HephaestusBase where

data SPLModel = SPLModel {
  featureModel :: FeatureModel
} }

data InstanceModel = InstanceModel {
  featureConfiguration :: FeatureConfiguration
} deriving (Data, Typeable)

data TransformationModel = UndefinedTransformation

transform :: TransformationModel -> SPLModel -> InstanceModel -> InstanceModel
transform UndefinedTransformation _ _ = undefined

data ExportModel = UndefinedExport

lstExport :: [ExportModel]
lstExport = []

export :: ExportModel -> FilePath -> InstanceModel -> IO()
export UndefinedExport _ _ = undefined

mkEmptyInstance :: FeatureConfiguration -> SPLModel -> InstanceModel
mkEmptyInstance fc spl = InstanceModel {
  featureConfiguration = fc
}

xml2Transformation :: String -> [String] -> ParserResult TransformationModel
xml2Transformation ”Undefined” _ = undefined

main :: IO ()
main = do
  ...
  let spl = SPLModel { featureModel = fm }
  let product = undefined
  let out = (outputFile (snd t) (snd n))
  sequence_ [export x out product | x <- lstExport]
  ...

Figure 13: Code snippet of the Haskell module for the Hephaestus Base
• defines one case for the `transform` function (lines 13-14), which just supports the `UndefinedTransformation` transformation,

• declares `ExportModel` data type (line 16) which just supports the `UndefinedExport` constructor,

• defines an empty list (`lstExport`) (lines 18-19) of export data types (used to export a product in different output formats),

• defines one case for the `export` function (lines 21-22), which also just supports the `UndefinedExport` export format,

• defines a `mkEmptyInstance` function (lines 24-27) that returns an `InstanceModel` having only the `featureConfiguration` data field,

• defines a `xml2Transformation` function (lines 29-30) which comprises the CK XML parser process performing the recognition of the concrete syntax of asset transformations of the Hephaestus-PL instance, and

• defines a `main` function (lines 32-39) to run the Hephaestus instance.

The `TransformationModel` and `ExportModel` data types could be understood as variation points which will be resolved by introducing new values that will replace the `UndefinedTransformation` and `UndefinedExport` definitions, respectively, in the refinement of Hephaestus-PL instance. Instances of Hephaestus-PL must provide one definition for the `transform` function for each supported SPL asset, even though the product derivation process of Hephaestus-PL refines the `transform` function introducing those new definitions through metaprogramming.

Line 36 of the Figure 4 represents an undefined Hephaestus-PL's instance (`product` variable) and this line will be replaced in emerging product by the line `let product = build fm fc cm spl` that executes the `build` function to generation of a product Hephaestus-PL instance from feature configuration.

5.2.1. **Hephaestus Product**

The module provides the `build` function, which controls Hephaestus-PL's product derivation process by progressive refinement of the base product. The module also provides a simple definition for the `transform` function (code snippet in the Figure 14). In this particular instance, the `transform` function only deals with the transformations of the Hephaestus SPL Asset, which are detailed in Section ??.
build :: FeatureModel
  -> FeatureConfiguration
  -> ConfigurationKnowledge
  -> SPLModel
  -> InstanceModel

build fm fc ck spl = stepRefinement ts spl emptyInstance
  where emptyInstance = mkEmptyInstance fc spl
        ts = tasks ck fc

tasks :: ConfigurationKnowledge -> FeatureConfiguration -> [TransformationModel]
tasks ck fc = concat [transformations c | c <- ck, eval fc (expression c)]

transform :: TransformationModel -> SPLModel -> InstanceModel -> InstanceModel
transform (HephaestusTransformation t) s i = transformHpl t s i

Figure 14: Code snippet of the Hephaestus product

5.3. Description of the Transformational Approach

The underlying hypothesis is that certain key types and functionality of any specific Hephaestus instance can be derived by metaprogramming. To this end, we consider an Base Product from which to build “Hephaestus-PL products”. All products are expected to define SPLModel, InstanceModel, ConfigurationKnowledge, TransformationModel, ExportModel, a transformation function transform, a output format function export and main function that controls the generation the product instances.

All forms of assets are hence provided in a form that they can contribution to the above-mentioned entities as presented in SPL Asset block in Figure 12. We use metaprogramming operators that build the product’s entities (say, types and functions) from the corresponding parts of the asset-related modules by adding parts incrementally to the base product.

Because of the objective of self-application of Hephaestus (bootstrapping design principle), the above-mentioned transformational approach is actually packaged as another kind of asset: Hephaestus SPL Asset. That is, the construction of any specific Hephaestus instance is controlled by a feature configuration relative to Hephaestus-PL’s feature model and the corresponding configuration knowledge as presented in module MetaData.hs.
5.4. Metaprogramming Operations

The required metaprogramming operations are of different complexity as implemented in MetaProgramming.hs module. We begin with the simpler operations. There is an operation setModuleName to modify the module name so that the module of the base product can be renamed into a module for the emerging product, say Hephaestus instance. This operation is performed on the SelectBaseProduct and BindProductName transformations. There is an operation addImportDecl to add an import declaration to the main module of the emerging product; this operation is needed to incorporate any additional kind of asset. This operation is performed on the SelectCKParser transformation to add an import declaration to CK parser (CK.Parsers.XML.XmlConfigurationParser module); on the SelectAsset transformation to add import declarations to incorporate the asset modules (algebraic data types, transformations, parser and output format) and the

<table>
<thead>
<tr>
<th>Hephaestus-PL Transformations</th>
<th>Metaprogramming Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectAsset assetName</td>
<td>addUpdateCase</td>
</tr>
<tr>
<td></td>
<td>initializeFieldWithFun</td>
</tr>
<tr>
<td></td>
<td>addImportDecl (4x)</td>
</tr>
<tr>
<td></td>
<td>addLetInstruction</td>
</tr>
<tr>
<td></td>
<td>addGeneratorInstruction</td>
</tr>
<tr>
<td></td>
<td>initializeField</td>
</tr>
<tr>
<td></td>
<td>addField (2x)</td>
</tr>
<tr>
<td></td>
<td>addConstructor</td>
</tr>
<tr>
<td></td>
<td>addUpdateCaseList</td>
</tr>
<tr>
<td>SelectExport assetFormat</td>
<td>addUpdateCase</td>
</tr>
<tr>
<td></td>
<td>addImportDecl</td>
</tr>
<tr>
<td></td>
<td>addConstructor</td>
</tr>
<tr>
<td></td>
<td>addListElem</td>
</tr>
<tr>
<td>SelectBaseProduct</td>
<td>setModuleName (2x)</td>
</tr>
<tr>
<td></td>
<td>removeImportDecl</td>
</tr>
<tr>
<td>BindProductName</td>
<td>setModuleName (2x)</td>
</tr>
<tr>
<td>RemoveProductMainFunction</td>
<td>removeFunction</td>
</tr>
<tr>
<td>SelectCKParser</td>
<td>addImportDecl</td>
</tr>
<tr>
<td></td>
<td>addGeneratorInstruction</td>
</tr>
<tr>
<td></td>
<td>addLetInstruction (2x)</td>
</tr>
</tbody>
</table>

Table 3: Mapping of Hephaestus-PL transformations into metaprogramming operations
data types module of emerging product. There is an operation `addField` to extend a given record type with a field; this operation is needed in the extension of Hephaestus’ data types `SPLModel` and `InstanceModel`. There is also an operation `addConstructor` to extend a given algebraic data type with a constructor declaration besides to remove the constructor declaration not used, i.e., `UndefinedTransformation` constructor; this operation is needed for assembling Hephaestus’ data type `TransformationModel` for transformations on possibly different forms of assets. We note here that `addConstructor` is deliberately limited to only support the addition of a constructor with a single constructor component and to actually reuse that component’s specified type for the constructor’s name. Therefore, we defined a similar operation `addConstructorWithoutArgs` to extend the Hephaestus’ data type `ExportModel` for different asset output formats. In this case, the `addConstructorWithoutArgs` only supports the addition of a constructor’s name without components. We also remove the variation point declaration `UndefinedExport` in this operation. The operation `addListElem` adds into `lstExport` list the constructors of the Hephaestus’ data type `ExportModel`.

The addition of fields and constructors is relatively straightforward at the type level, but we need additional, non-trivial operations that transform functions that readily use the affected types. There are the operations `initializeField` and `initializeFieldWithFun` which modify all expressions for record construction for a given record type such that in the first case a given field is initialized by a constant (say, a variable name or a function name with assumed zero arity) and in the second case a given field is initialized by a function name with assumed one arity, i.e., the empty asset function that receives the asset data type as input parameter. Other forms of reaction to added fields are conceivable, but the given form turned out to be sufficient in the experiment. There is also an operation `addUpdateCase` which extends function definitions by a case in reply to a previously added constructor. Different forms of adding cases are conceivable. The following, non-trivial form was required in the experiment.

Addition of a constructor is needed for `TransformationModel`, which in turn is to be used in a `transform` function that essentially interprets transformation models (‘terms’). The type of the function is this:

```haskell
transform :: TransformationModel
          -> SPLModel
          -> InstanceModel
          -> InstanceModel
```
The idea is here that the function perhaps case discrimination on the first argument and essentially delegates to a more specific transformation function that readily handles the given transformation on the further arguments at hand. An additional complication arises from the fact that the more specific transformation function would not be able to operate on the composed types SPLModel and InstanceModel. For example, consider a Hephaestus instance that should support use cases and business processes. Then, the following transform function has to be synthesized:

1. transform (UseCaseTransformation x0) x1 x2 = transformUcm x0 x1 x2
2. transform (BusinessProcessTransformation x0) x1 x2 = transformBpm x0 x1 x2

Hence, the operation addUpdateCase must add cases such that indeed case discrimination is performed on the first argument and the last two arguments are defined by x1 and x2 variables when being passed to the function on the RHS. A similar operation to addUpdateCase was defined to extend the export function definition by a case in reply to a previously added constructor into ExportModel data type. The type of the function is this:

1. export :: ExportModel -> FilePath -> InstanceModel -> IO ()

For example, consider a Hephaestus instance that should support use cases and business processes in xml format. Then, the following export function has to be synthesized:

1. export :: ExportModel -> FilePath -> InstanceModel -> IO ()
2. export (ExportUcmXML) x1 x2 = exportUcmToXML (x1 ++ ".xml") (ucm x2)
3. export (ExportBpmXML) x1 x2 = exportBpmToXML (x1 ++ ".xml") (bpm x2)

The extension of the xml2Transformation function is also done with the addUpdateCase operation applied in a list of cases of transformations supported by CK for the emerging Hephaestus-PL product.

Moreover, we defined two new operations to extend the main function into a module for the emerging product, they are addLetInstruction and addGeneratorInstruction. The first is used to add let sentences in the main function to recover the asset spl and to execute the parser function and moving the asset spl into Hephaestus-PL’s instance. The second operation addGeneratorInstruction is used to add instructions about asset and CK parsers into module for the emerging product. The operation addLetInstruction adds the instruction let product = build fm fc cm spl responsible for generation of new Hephaestus-PL’s instances. To ensure the compilation of BaseProduct and Hephaestus-PL modules, we add
the instructions related to CK (import and parser) and the calling of build function only in emerging product.

There is also the addUpdateCaseList operation based on addUpdateCase operation and using a list of cases to extend the xml2Transformation function that comprises the CK parser process performing the recognition of the concrete syntax of asset transformations of the Hephaestus-PL instance.

Finally, the operation removeImportDecl removes an import declaration of the main module of the emerging product. This operation is performed on the SelectBaseProduct transformation to remove the import declaration of the module with algebraic data types of base product. This is replaced by import declaration of module with algebraic data types of emerging Hephaestus-PL product.

The operation removeFunction removes the main function of the emerging product. It is necessary in the generation of Hephaestus product because its main function is the buildHpl function located into IO.hs module. This operation is executed by RemoveProductMainFunction transformation assigned to Hephaestus expression feature.

Table 3 summarizes the mapping of Hephaestus-PL transformations into metaprogramming operations.

5.4.1. Hephaestus UCM and BPM instance’s Haskell module

From the Hephaestus product presented in Section ??, the product derivation process generates a new Haskell module that refines the BaseProduct module with the selected features of the product configuration. For instance, Figure 15 shows the Hephaestus-PL instance source code generated by selecting the features Use Case, Business Process, UcmToXML and BpmToXML. New cases related to these assets and output formats will be introduced into the definition of the transform (lines 17-18) and export (lines 30-31) functions, as well as new data fields are introduced into the definitions of the SPLModel (lines 3-4), InstanceModel (lines 9-10), TransformationModel (lines 13-14) and ExportModel (line 27) data types, the mkEmptyInstance (lines 23-24) function returns an instance model with new data fields and, finally, new elements corresponding to ExportModel data type are introduced into the lstExport (line 34) list.

Moreover, the Hephaestus-PL product derivation process introduces sentences in the main function for Use Case and Business Process asset parser (lines 39-42) and updates the fields of the SPLModel instance into spl (line 43) declaration and the product (line 44) declaration with calling build func-
tion. Besides the mentioned transformations applied to the BaseProduct module during the product derivation, the resulting module also has to import declarations that are specific for a given selection of features.

6. Reactive Process

As discussed earlier, the design of Hephaestus-PL should improve the flexibility to introduce support for managing variabilities on new SPL assets, and thus evolve the configurability space of Hephaestus-PL. In this section we describe a process (Figure 20 shows the corresponding BPMN diagram) that could guide domain engineers to proceed in this task. It is important to note that we have designed this process based on our experience for evolving the versions of Hephaestus-PL to support several assets (use cases, business processes, requirements and code). Therefore, it focuses on the reactive process to introduce new assets into Hephaestus-PL.

Three roles contribute to this process, which are represented as the lanes in Figure 20. Customers start the reactive process in Hephaestus-PL demanding a Hephaestus tool with new requirements. Application engineers consider the new requirements, specify and map the requirements for the features of Hephaestus-PL, and evaluate whether the requirements are already supported by Hephaestus-PL. If so, they generate the Hephaestus-PL product with the desired configuration, testing and delivering it to the customer. If Hephaestus-PL does not support the customer tool’s requirements yet, the application engineers submit to the domain engineer the demand for evolving Hephaestus-PL to support the new requirements (assets).

The domain engineers here comprise two distinct roles: asset domain expert that defines and implements the artifacts of the new assets; and domain engineer of Hephaestus-PL that evaluates the impact and integrates the artifacts of the new asset into Hephaestus-PL. The asset domain expert executes activities concerning the new artifact, by providing its implementation in terms of data types, transformations, parsers, and output format functions. It corresponds the Define asset type structures, Define asset transformations, Implement asset parser and Implement asset output format activities in Figure 20. No order is specified among these tasks because they are normally carried out iteratively. The domain engineer of Hephaestus-PL assesses whether the introduction of a new asset needs to update the Hephaestus-PL Kernel’s APIs (high and low level transformations); integrates the new asset’s artifacts into Hephaestus-PL infrastructure; updates the metadata structures inserting
data SPLModel = SPLModel {
  featureModel :: FeatureModel,  
  splUcm :: UseCaseModel,  
  splBpm :: BusinessProcessModel
}

data InstanceModel = InstanceModel {
  featureConfiguration :: FeatureConfiguration,  
  ucm :: UseCaseModel,  
  bpm :: BusinessProcessModel
} deriving (Data, Typeable)

data TransformationModel = UseCaseTransformation UseCaseTransformation  

transform :: TransformationModel -> SPLModel -> InstanceModel -> InstanceModel
transform (UseCaseTransformation x0) x1 x2 = transformUcm x0 x1 x2
transform (BusinessProcessTransformation x0) x1 x2 = transformBpm x0 x1 x2

mkEmptyInstance :: FeatureConfiguration -> SPLModel -> InstanceModel
mkEmptyInstance fc spl = InstanceModel {
  featureConfiguration = fc,  
  ucm = emptyUcm (splUcm spl),  
  bpm = emptyBpm (splBpm spl)
}

data ExportModel = ExportUcmXML | ExportBpmXML

export :: ExportModel -> FilePath -> InstanceModel -> IO ()
export (ExportUcmXML) x1 x2 = exportUcmToXML (x1 ++ ".xml") (ucm x2)
export (ExportBpmXML) x1 x2 = exportBpmToXML (x1 ++ ".xml") (bpm x2)

lstExport :: [ExportModel]
lstExport = [ExportUcmXML, ExportBpmXML]

main :: IO ()
main = do
  ...
  let bModel = fromJust (findPropertyValue "businessprocess\-model" ps)
  let uModel = fromJust (findPropertyValue "usecase\-model" ps)
  (Core.Success bppl) <- parseBusinessProcess (ns bpSchema) (snd bModel)
  (Core.Success ucpl) <- parseUseCaseFile (ns ucSchema) (snd uModel)
  let spl = SPLModel{featureModel = fm, splUcm = ucpl, splBpm = bppl}
  let product = build fm fc cm spl
  let out = (outputFile (snd targetDir) (snd name))
  sequence_ [export x out product | x <- lstExport]
references to the new asset; updates Hephaestus-PL’s FM and CK using the guidance of safe evolution templates of software product lines defined [21]; and validates the new asset by generating a new Hephaestus-PL product with the selected new asset and checking the correctness of the generated source code. If the generated product is a valid Hephaestus product then the domain engineer’s activities are finalized.

6.1. Evolution of Hephaestus-PL to support the Requirement Model asset

To illustrate the execution of the reactive process we present the evolution of Hephaestus-PL’s to support the management of variabilities in the Requirement Model asset. The current version of Hephaestus-PL supports Use Case Model and Business Process Model assets. Figure 16 summarizes this scenario of evolution of Hephaestus-PL which we describe below.

![Hephaestus-PL's Evolution to introduce the Requirement asset.](image)

The reactive process to introduce new assets into Hephaestus-PL begins with the domain engineer assessing the impacts in Kernel’s APIs of
data Requirement = Requirement {
  reqId :: Id,
  reqName :: String,
  reqDescription :: String
} deriving (Show, Data, Typeable)

data RequirementModel = RM {
  reqs :: [Requirement]
} deriving (Show, Eq, Data, Typeable)

Figure 17: Definition of RequirementModel data type

Hephaestus-PL to support the Requirement asset. It’s necessary to analyze whether the high level APIs – corresponding to Hephaestus-PL’s transformations (SelectAsset and SelectExport) – and low level APIs – corresponding to the metaprogramming operations – support the specification of new Requirement asset. This reactive process considers that the transformations of Hephaestus-PL (SelectAsset and SelectExport) support new assets, i.e., based on our experience, introducing support for variation in a new asset in Hephaestus-PL does not impact the high level APIs.

Then, the asset domain expert has to implement the four artifacts of Requirement asset to evolve Hephaestus-PL supporting this asset. The items below describe what is necessary to define about the new Requirement asset:

(i) to define the RequirementModel data type and other auxiliary types that represent a set of requirements with their variabilities. The RequirementModel data type is composed by id, name, and description fields (see code snippet in Figure 17) and is located in a specific module, i.e., HplAssets.ReqModel.Types.hs module;

(ii) to define the transformations and empty instance function of Requirement asset. The transformations specified that managing variabilities in Requirements are SelectAllRequirements, SelectRequirements and RemoveRequirements. The emptyReq is the function which defines a Requirement asset empty instance. It is also necessary to define a data type and a function to comprise all the transformations of Requirement asset to introduce them in a product Hephaestus-PL instance. For example, we defined the RequirementTransformation data type and the transformReq function, respectively. The definition RequirementTransformation must have the clause deriving (Show, Eq, Ord) to allow the ordering and viewing
The Requirement asset's transformations. All definitions of data types to Requirement asset must be in `HplAssets.ReqModel.Types.hs` module. The `transformReq` function must have a similar signature of the `transform` function into `BaseProduct.hs` module that is a Hephaestus-PL base instance. Besides, a `transformReq` function must be defined by pattern matching for each constructor declared in the `RequirementTransformation` data type (see code snippet in Figure 18). Both the `emptyReq` and `transformReq` functions are implemented in a specific module, i.e., `HplAssets.Requirements.hs` module. These functions must be visible by the Hephaestus-PL kernel to be able to generate a new Hephaestus-PL instance with selected Requirement asset;

(iii) to implement a new Requirement asset parser function to reading the artifacts of Requirement product line from a public representation format such as XML into RequirementModel data type in Hephaestus-PL.

(iv) to implement a new Requirement asset output format function, i.e., `exportReqToLatex` function that enables the exportation of Requirement product instance in the Latex output format out Hephaestus’s tool.

After setting the four artifacts of Requirement asset, the domain engineer performs the integration of the modules of the new Requirement asset into Hephaestus-PL. She creates a new directory to the Requirement asset below the `HplAssets` directory, i.e., `ReqModel` directory, and inserts the new asset’s
modules there, except the `HplAssets.Requirements.hs` module with the definitions of `emptyReq` and `transformReq` functions which is placed at the same level of `ReqModel` directory (below the `HplAssets` directory).

Then, the domain engineer picks up the information of the `Requirement` asset modules and setting the data sets that supporting the execution of low-level APIs of Hephaestus-PL Kernel. This represents to extend the `AssetMetaData` and `ExportMetaData` metadata structures defined in `HplAssets.Hephaestus.MetaData.hs` module to support the new `Requirement` asset. Some informations defined in the modules of `Requirement` asset, such as identifiers of data types, fields, functions and modules are inserted in the metadata structures. These informations will be used by metaprogramming support to extend the open data types and open functions in the Hephaestus-PL’s base product and generating an instance of Hephaestus-PL with the selected `Requirement` asset. For example, informations about `Requirement` asset such as the data field name and data type to extend the `SPLModel` and `InstanceModel` data types, i.e., ("splReq","RequirementModel") and ("req","RequirementModel") respectively; the function name that defines a `Requirement` asset empty instance, i.e., `emptyReq`; the function name that implements the parser of `Requirement` asset, i.e., `parseRequirementModel`; the name of data type that defines all the `Requirement` asset transformations, i.e., `RequirementTransformation`.

After that, the domain engineer updates the Hephaestus-PL’s FM and CK defined into `HplAssets.Hephaestus.MetaData.hs` module using the safe evolution templates of software product lines defined in [21]. The new or-features, `Requirement` asset and its Latex output format, and a new cross tree constraint `ReqToLatex ⇒ Requirement` are inserted into Hephaestus-PL’s FM. We introduce the new `Requirement` OR-feature below the `SPLModel` mandatory feature and introduce the new `ReqToLatex` OR-feature below the `Output Format` optional feature (see Figure 19). The Hephaestus-PL’s CK is updated with the new mapping of feature expressions to transformations about `Requirement` asset (see Table 4).

Finally, the domain engineer generates a product Hephaestus-PL instance by selecting a product configuration only with the new `Requirement` asset integrated into Hephaestus-PL. She validates if the generated instance contains the correct definitions of the `Requirement` asset into points of variability of the Hephaestus-PL base product and reported the new Hephaestus-PL version to application engineer. Suppose that the validation of the integration of `Requirement` asset into Hephaestus-PL has not been successfully, then it is
necessary to return to the assessment of impact in the Hephaestus-PL Kernel’s APIs until the validation correct of Hephaestus-PL instance.

7. Results and Discussion

In order to guide the discussion of the results, we apply the Goal Question Metric (GQM) method [22], so that it helps structuring 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 the goals, questions and metrics of our investigation (Subsection 7.1), then evaluate our empirical study accordingly (Subsection 7.2), and finally discuss its limitations (Section 7.3).

7.1. Goals, Questions, and Metrics

Our goal is to assess the configurability and flexibility of Hephaestus-PL (Sections 4 and 5), regarding the application engineering phase for software product development and within the context of the different artifacts. Since Hephaestus-PL emerged from a systematic evolution of Hephaestus, this later is also assessed as a baseline. According to GQM, Table 5 summarizes the general evaluation goal of our work.

In Section 2, we characterized the context, the viewpoint, and part of the object 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 Sections 4 and 5, we detailed part of the object, i.e., Hephaestus-PL. This section proceeds with the analysis of Hephaestus-PL and Hephaestus, using a qualitative and quantitative evaluation that answers our GQM metrics, thus meeting the purpose and issue components of the study’s goal.
<table>
<thead>
<tr>
<th>Feature Expressions</th>
<th>Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>SelectBaseProduct</td>
</tr>
<tr>
<td>Hephaestus</td>
<td>BindProductName &quot;Hephaestus&quot;</td>
</tr>
<tr>
<td></td>
<td>SelectAsset &quot;Hephaestus&quot;</td>
</tr>
<tr>
<td></td>
<td>RemoveProductMainFunction</td>
</tr>
<tr>
<td>NOT Hephaestus</td>
<td>SelectCKParser</td>
</tr>
<tr>
<td>UseCase</td>
<td>SelectAsset &quot;Use Case&quot;</td>
</tr>
<tr>
<td>UseCase AND UcmToXML</td>
<td>SelectExport &quot;UcmToXML&quot;</td>
</tr>
<tr>
<td>UseCase AND UcmToLatex</td>
<td>SelectExport &quot;UcmToLatex&quot;</td>
</tr>
<tr>
<td>BusinessProcess</td>
<td>SelectAsset &quot;Business Process&quot;</td>
</tr>
<tr>
<td>BusinessProcess AND BpmToXML</td>
<td>SelectExport &quot;BpmToXML&quot;</td>
</tr>
<tr>
<td>Requirement</td>
<td>SelectAsset &quot;Requirement&quot;</td>
</tr>
<tr>
<td>Requirement AND ReqToLatex</td>
<td>SelectExport &quot;ReqToLatex&quot;</td>
</tr>
</tbody>
</table>

Table 4: Hephaestus-PL’s CK after introducing Requirement asset.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>assess</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue</td>
<td>the configurability and flexibility of</td>
</tr>
<tr>
<td>Object</td>
<td>Hephaestus-PL and Hephaestus</td>
</tr>
<tr>
<td>Viewpoint</td>
<td>application engineering perspective</td>
</tr>
<tr>
<td>Context</td>
<td>different artifacts</td>
</tr>
</tbody>
</table>

Table 5: GQM goal of this research.

From the study’s goal presented in Table 5, we derive questions that best characterize our study. Moreover, related to each question, we use one or more quantitative or 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 [22]. In what follows, we present the questions (Q) and the metrics (M) of our GQM model and explain how they trace to the study’s goal.

Q1 Is the variability mechanism sufficiently expressive?

- Metric M1.1: Is there support for open data types?
- Metric M1.2: Is there support for open functions?
- Metric M1.3: Is there support for the instantiation of one SPL
Questions Q1-Q4 represent relevant characteristics of the development of Hephaestus-PL when compared to Hephaestus. Q1 traces to the configurability issue, addressing the required expressiveness of the underlying variability management mechanism. Correspondingly, Metrics M1.1, M1.2, M1.3 and M1.4 investigate whether there is support for the types of variability in Hephaestus-PL, classified as open data types, open functions, single asset instantiation, and assets composition, as explained in Section 3. In particular,
metric M1.4 focuses on a current issue of Hephaestus: the need to instantiate more than one SPL asset defined in SPLModel and InstanceModel algebraic types. Likewise, Q2 also deals with the configurability issue, but addresses it within a predefined scope of assets. Its metrics then address the effort to add a new configuration, from different granularity perspectives (modules and lines of code), automation support, and analytical complexity, the latter being relevant for scalability. Differently, Q3 traces to the flexibility issue, by considering the necessary evolution effort to address variability in a new asset. This question is refined by metrics from different granularity perspectives (code artifacts represent data types and functions, except modules) and also considering automation support. Finally, Q4 also refers to the flexibility issue and assesses whether modularity in handling asset related variability is supported or not, an important property to the reactive approach for SPL development since it potentially supports the introduction of new features. The homogeneity (metric 4.1) refers to the kinds of Haskell syntactic structures (data types, functions, classes) associated with the changes. We consider homogeneous when there is only one type of syntactic structure to change and heterogeneous otherwise. The location of the changes (metric 4.2) is assessed on the modules with tangled code of features and can be defined as localized when it involves only one module or scattered otherwise. At the end, the metric M4.3 is defined as a result of metric M4.1 and M4.2. Thus, M4.3 can be high when we observe the values homogeneous to M4.1 and located to M4.2 otherwise we define M4.3 as low.

7.2. Assessment

Given the GQM setup from Subsection 7.1, we now proceed to evaluating the design and implementation techniques for managing variability (Sections 4 and 5). Table 6 summarizes the assessment to Hephaestus and Hephaestus-PL of the metrics associated to questions defined in the our GQM model.

Regarding Q1, when evaluated in Hephaestus-PL, there is sufficient expressiveness to support all variability types in Hephaestus. The support for variability in data types and functions (metrics M1.1 and M1.2) is guaranteed by the transformational approach of variability management used in Hephaestus-PL, i.e., using metaprogramming operations to extend the data types and functions. The support for the composition of assets, i.e., instantiation of a product with one or more assets (metrics M1.3 and M1.4) is
<table>
<thead>
<tr>
<th>Metric</th>
<th>Hephaestus</th>
<th>Hephaestus-PL</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.1</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>M1.2</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>M1.3</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>M1.4</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>M2.1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>M2.2</td>
<td>9n</td>
<td>0</td>
</tr>
<tr>
<td>M2.3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>M2.4</td>
<td>8n</td>
<td>0</td>
</tr>
<tr>
<td>M2.5</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>M2.6</td>
<td>$O(n)$</td>
<td>$O(k)$</td>
</tr>
<tr>
<td>M3.1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>M3.2</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>M3.3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>M3.4</td>
<td>low, depends of new asset</td>
<td>low, depends of new asset</td>
</tr>
<tr>
<td>M3.5</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>M4.1</td>
<td>heterogeneous</td>
<td>homogeneous</td>
</tr>
<tr>
<td>M4.2</td>
<td>scattered</td>
<td>localized</td>
</tr>
<tr>
<td>M4.3</td>
<td>low</td>
<td>high</td>
</tr>
</tbody>
</table>

Table 6: Summary of the assessment of the metrics of GQM model

guaranteed by the build process of the Hephaestus-PL’s kernel and metaprogramming operations generating a Hephaestus-PL variant that represents an instance of product which combines any assets of the Hephaestus-PL’s Feature Model. When evaluated in Hephaestus, that is a variant manually generated that supports derivation of products using a compositional approach of artifacts and transformations into CK to generate a new product, we observe there is not support for variability management in data types and functions by the introduction of new assets and for the composition of any assets. Related to M1.4, the generation of Hephaestus variant that supports a composition of assets without having to instantiate all assets requires noticiable effort because it is necessary to change several code artifacts (modules, data types and functions). This ad hoc change in different parts of the code is an error-prone activity and it is not usually done. Alternatively, we could use the concept of Over Feature to generate new Hephaestus variants because the

41
impact is considered small and limited into main function.

Regarding Q2, when evaluated in Hephaestus-PL, the effort to address a new configuration from a fixed number of assets is constant and small and represents the effort to specify the new product configuration for Hephaestus-PL with the selected assets, whereas the assets of the desired configuration are already integrated into Hephaestus-PL. Therefore, a measure for the metric M2.6 is constant and close to zero (≃ O(k)). Likewise, the measure for the metrics M2.1, M2.2, M2.3 and M2.4 is zero, i.e., there are not created and modified modules. Regarding the metric M2.5, there is an automation support in Hephaestus-PL to generate new asset configurations based on the build process, Hephaestus-PL’s transformations and operations of metaprogramming which treat the variability in Hephaestus-PL.

When evaluated in Hephaestus, the effort to address a new configuration of assets can be seen in two ways: by using or not the concept of Over Feature. First, using the concept of Over Feature and a Hephaestus product that supports a set of assets where the desired configuration is a subset of these assets, the effort is considered small with impact only on the Main.hs module that needs some changes in the code. Therefore, the measure of the metric 2.1 is one module and the measure of the metric 2.3 is zero module (i.e., no module is created). Lines of code related to selected assets are changed and created in the module Main.hs, i.e., importing the modules of algebraic data types, parser and output format, reading the SPL asset file in the input Hephaestus properties file, executing the asset parser, updating the createSPL function after execute the asset parser and executing the call of the export function of the asset in the output format. This represents about nine changed lines (metric M2.2) and eight created lines (metric M2.4) in Main.hs module. With respect to the metric M2.5 there is not automation support in Hephaestus and analytical complexity of the effort (metric M2.6) using the concept of Over Feature in Hephaestus is the sum of the number of changed and created lines (metric M2.2 and M2.4) multiplied by the number of assets (n) of the desired configuration, i.e., (M2.2 + M2.4) * n ≃ O(n). Therefore, the effort is proportionally linear to the number of assets of the configuration.

Analyzing now the effort to address a new configuration of assets when we did not use Over Feature in Hephaestus, we observe that the impact is quite high because it is necessary to introduce into the Hephaestus variant the code related to selected assets and this requires a good understanding the code and changes in several modules, data types and functions. Moreover,
being a manual activity is more error-prone and to check invalid settings is not an easy task to be done manually. Thus, only to use code of the assets of the selected product configuration would be necessary to copy the source code of Hephaestus variant to a new directory and remove the non-selected features. Currently Hephaestus does not work like this.

Regarding Q3 refers to reactive process, both Hephaestus-PL and Hephaestus tools, the effort to address a new asset considers the major and initial effort to generate the elements of the new asset, such as the abstract data types, the transformations, the parser function and the output format function. This effort is the same in both tools and assessing it is outside the scope of this paper. After generation of asset’s elements, there is the effort to integrate these elements of new asset into Hephaestus-PL and Hephaestus. We evaluate this later effort. When evaluated in Hephaestus-PL, the number of changed modules and code artifacts (metrics M3.1 and M3.2) is only one changed module (MetaData.hs) and four changed code artifacts. In MetaData.hs module we need to change the featuremodel and configurationKnowledge functions that build the Hephaestus-PL’s FM and CK and to change the assetMetaData and exportMetaData lists that contain the meta data structures that support metaprogramming operations. The number of created modules and code artifacts (metrics M3.3 and M3.4) are four created modules (Types.hs contains the algebraic data types, <NewAsset>.hs contains the transformations that manage asset’s variabilities, <NewAssetFomartParser>.hs contains the asset parser function and <NewAssetOutputFormat>.hs contains a asset output format function) where the sum of created code artifacts (data types and functions) varies depending on the asset. Currently, there is not automation support in Hephaestus-PL to address a new asset (metric M3.5) but it is possible in the future using a minimum set of variables with the automatic or semi-automatic derivation of the asset meta data informations into asset elements since the meta data structures have a regular structure and the implementation of asset follow some design rules to enable this automatic or semi-automatic derivation. When evaluated in Hephaestus, the number of changed modules and code artifacts (metrics M3.1 and M3.2) are five changed modules (ConfigurationKnowledge/Types.lhs, ConfigurationKnowledge/Interpreter.hs, Parsers/XML/XmlConfigurationKnowledge.hs, ExportProduct.hs and Main.hs) and six changed code artifacts that are SPLModel and InstanceModel data types and build, xml2Transformation,
The number of created modules and code artifacts (metrics M3.3 and M3.4) are also four created modules (Transformations/<newAsset>/Types.hs, <newAsset>/Parsers/<newAssetParser>.hs and <newAsset>/PrettyPrinter/<newAssetOutputFormat>.hs) and the number of created code artifacts to represent the elements of the new asset (abstract data types, transformations, parser function and output format function) varies depending on the asset. In both Hephaestus-PL and Hephaestus the number of changed and created code artifacts is small because Haskell is a declarative (functional) language. In Hephaestus there is not automation support to address a new asset (metric M3.5) and it is more difficult than Hephaestus-PL to implement some automation because in Hephaestus asset code is scattered in five modules and its format is heterogeneous.

In terms of feature modularization (Q4), we observe that both Hephaestus-PL and Hephaestus support modularity in the elements of SPL asset which are defined in independent modules and have their implementation entirely confined to specific modules, like the four modules that define respectively the algebraic data types, transformations, parser and output format of the asset. However, there is code associated with SPL asset that is scattered and tangled with code from other SPL asset in some modules. Therefore, to assess if the management of SPL asset related variability is modular we define three metrics. We evaluate the homogeneity (metric M4.1), the location (metric M4.2) and the possibility of automation (metric M4.3) of the changes related to variation points of SPL asset in the Hephaestus-PL and Hephaestus objects as qualitative criteria for the Q4 question. In Hephaestus we define the changes to address an SPL asset as heterogeneous, scattered and therefore with lower possibility of automation. They are heterogeneous because they refer to different types of changes such as insertion of fields in data types, defining new class instances, defining new functions and introducing new sentences in functions. The changes are scattered because they occur in five different Hephaestus modules (as presented in Hephaestus’s metric M3.1) and the Hephaestus modules that address variability of SPL asset need to be updated manually and it is a error-prone activity (i.e., changes are heterogeneous, scattered in various modules and tangled with other SPL assets). On the other hand, in Hephaestus-PL the changes to address new assets are classified as homogeneous and localized since they only refer the definition of SPL asset meta data informations in a single MetaData.hs module. Thus, in Hephaestus-PL there is more possibility of automation from a process of
inference into the SPL asset modules implemented according to design rules previously defined. We had a initial greater effort in the development of Hephaestus-PL with the participation of a Haskell expert creating an infrastructure that contributes to a reduced effort to address the changes related to new assets when compared to the effort for the same purpose observed in Hephaestus.

Furthermore, in Hephaestus-PL some degree of modularity was obtained by the mapping of Hephaestus-PL’s product configurations to metaprogramming transformations where the Hephaestus-PL’s variability related to the assets’ configuration is handled dynamically and automatically in the generation of a Hephaestus-PL variant. This is allowed by the transformational process of generating products in Hephaestus-PL with the support of the metaprogramming operations that meet the needs of managing the variability of assets in the artifacts (open data types and open functions) of the base module of Hephaestus-PL.

7.3. Threats to Validity

The threats to validity of our study can be addressed as follows:

- **Construct validity** concerns establishing correct operational measures for the concepts being studied. The main constructs in our study are the concepts of “configurability” and “flexibility”. Regarding the first, we defined metrics considering the underlying variability and effort of adding a configuration. As for the second construct, we defined metrics considering the evolution effort to address a new asset as well as key internal characteristic such as modularity.

- **Internal validity** concerns establishing a causal relationship, whereby certain conditions are shown to lead to other conditions. The design decisions behind the architecture definition trace to solving variability issues demanded by enhanced configurability support. The proposed reactive process was conceived to address the flexibility issue.

- **External validity** concerns establishing the domain to which a study’s findings can be generalized. Although our study focuses on a single tool, we believe that its design and supporting reactive process could be used to improve configurability and flexibility in other SPL product derivation tools.
• **Reliability** concerns demonstrating that the operations of a study can be repeated with the same results. Given the design, implementation, and reactive process descriptions, we expect that replications of our study should offer results similar to ours.

8. Related Work

Previous work discusses the need to address SPL tool development as SPL development itself [6]. Nevertheless, detailed guidelines on domain and application engineering were not explored.

As discussed in Section 3, a key requirement of Hephaestus-PL is to support or-configurability of product artifacts, which has not been explicitly addressed elsewhere.

In [8] was bootstrapped AHEAD from AHEAD. This is similar to our work, but they do not focus on addressing different artifacts and no explicit support for or-feature is provided. Furthermore, the programming language and paradigm is different from ours.

Transkell [23] is a domain specific language (DSL) developed to extract and modularize the variabilities of Hephaestus tool and makes it a product line. It is a technique of transformation approach similar to our work. The Hephaestuss feature model represents the transformations of Hephaestus and the syntax and semantics of the Transkell language were implemented in the environment Stratego/XT [24]. In our work, we focus on assets variabilities that represent the different domain artifacts of Hephaestus and we present the details of Hephaestus-PL developed as a product line.

The **DOPLER (Decision-Oriented Product Line Engineering for effective Reuse)** approach [25] represents a tool suite to SPL tool development as SPL development itself [6]. It comprises DoplerVML, a variability modeling language to define product lines based decision models with emphasis on the derivation of products. **DOPLER** was initially developed to support the domain of industrial automation (Siemens VAI), but the proposal to be extensible and customizable to different contexts to meet the needs of different users and organizations. However, the approach **DOPLER** does not support configurability of any combination of assets as we propose in the solution of our work.

Some other comparative studies with product derivation tools were conducted [26, 27]. [26] has analyzed six modern product derivation tools
(Captor, CIDE, GenArch, Hephaestus, pure::variants e XVCL) in the context of evolution scenarios of a software product line. The study analyzed the modularity, complexity and stability of product derivation artifacts along evolution of a mobile product line. The evaluation showed that modularity and stability requirements in software product lines are favorable to the flexibility of the tool. In [27] the flexibility and extensibility of product line development tools that support product derivation based on DDM and AOSD approaches were evaluated to support the addition of new features.

9. Conclusion

Hephaestus-PL is a product line resulting from evolution of the Hephaestus tool. Hephaestus [5] was initially designed to support application engineering and variability management in a product line of use case scenarios. Over time, Hephaestus evolved to manage variability in other assets such as requirements, source code, and business processes. However, this tool was not designed with flexibility and configurability in mind to allow its customization to address variability in a new specific asset nor in any desired combination of assets.

To address this shortcoming, Hephaestus-PL was developed. Hephaestus-PL increases the configurability of Hephaestus by allowing the derivation of instance tools managing variability in any combination of artifacts; additionally, its flexibility allows for further systematic extension to add new assets and their combination. Once Hephaestus-PL was bootstrapped, we defined a reactive approach to increase its configurability and to reach the goal of enabling the generation of different instances of Hephaestus.

An assessment reveals that Hephaestus-PL has improved configurability and flexibility when compared to previous evolution of Hephaestus. The proposed Hephaestus-PL’s architecture and the approach of variability management used in Hephaestus-PL address configurability and flexibility in the development of SPL derivation tools. Thus, we using a transformational approach with metaprogramming operations to extend the variation points of the base product of Hephaestus-PL instance.

In Hephaestus-PL some degree of modularity was obtained by the mapping of Hephaestus-PL’s product configurations to metaprogramming transformations where the Hephaestus-PL’s variability related to the assets’ configuration is handled dynamically and automatically in the generation of a Hephaestus-PL variant. This is allowed by the transformational process of generating
products in Hephaestus-PL with the support of the metaprogramming operations that meet the needs of managing the variability of assets in the artifacts (open data types and open functions) of the base module of Hephaestus-PL. Besides, the reactive process defined in Hephaestus-PL to introduce support for managing variabilities in new assets and it contributes to the flexibility of Hephaestus-PL.

Although our study focuses on a single tool, we believe that its design and supporting reactive process could be used to improve configurability and flexibility in other SPL product derivation tools. Nevertheless, further empirical work is necessary to address the external validity threat. We also plan to conduct further empirical studies assessing Hephaestus-PL’s evolution to handle variability in different kinds of artifacts.

As future work, we propose the definition of Design Rules that represent a mechanism that would allow reduction of the size of the asset metadata structures in Hephaestus-PL. Using an inference process into the asset modules, especially the modules that define the data types and the transformations of the asset, it would be possible to extract information, currently contained in the asset metadata structures, to extend the variability points of a Hephaestus-PL base product. In this case, we could reduce the size of the asset metadata structures. Another intermediate solution that would bring a good reduction of the asset metadata structures would work with two pieces of information, a acronym and a name of asset, and deriving most of the other pieces to extend the variation points of Hephaestus-PL.

References


Figure 20: Reactive Process for introducing new artifacts in Hephaestus-PL.