Variability Modeling for Distributed Development – A Comparison with Established Practice

Klaus Schmid

Institut für Informatik, Universität Hildesheim
Marienburger Platz 22, D-31141 Hildesheim
schmid@sse.uni-hildesheim.de

Abstract. The variability model is a central artifact in product line engineering. Existing approaches typically treat this as a single centralized artifact which describes the configuration of other artifacts. This approach is very problematic in distributed development as a monolithic variability model requires significant coordination among the involved development teams. This holds in particular if multiple independent organizations are involved.

At this point very little work exists that explicitly supports variability modeling in a distributed setting. In this paper we address the question how existing, real-world, large-scale projects deal with this problem as a source of inspiration on how to deal with this in variability management.

Keywords: Software product lines, variability modeling, eclipse, debian linux, distributed modeling, software ecosystems, global development.

1 Introduction

An increasing amount of software engineering is done in a (globally) distributed way. This has multiple reasons and takes multiple forms. In particular, we see three major forms of distributed development:

1. Distributed by discipline: different parts of a development are done by different (sub-)organizations. These organizational units may be distributed on a world-wide scale.
2. Distributed along a software supply chain: some components are developed by one organization and other components are developed on top of this by a different organization. Such a supply chain may exist either within a single company or across companies. This can in particular be a software ecosystem, i.e., the companies involved are independent on an organizational level and only coupled through the software product (line) [3].
3. Unstructured distributed development: the development distribution structure is not matched to the software structure, i.e., people at different locations work on the same parts of the software.

These forms of software development can be well combined with software product line engineering [14, 5, 12], leading to development organizations of high complexity and the need to synchronize variability across the involved organizations. Distribution type 1 is a rather common organization scheme for large scale development – and thus also for large scale product line development. For example, Nokia develops certain parts of its phone software in labs around the world. Distribution type 2 is actually well known under the name of product populations [22]. Distribution type 3 has also been applied in combination with software product line engineering. It should be noted that while many case studies of product line engineering exist [12, 4] and a significant number of them even deals with a distributed stetting, very little work explicitly addresses the issue of distributed variability modeling.

In this paper, we focus on the question how distributed development impacts variability management, respectively what characteristics make a variability management approach particularly suited for distributed development. As only distribution types 1 and 2 relate development structure and distribution, we will focus on those approaches to distributed development. We will also address in particular the situation of software ecosystems, i.e., development is distributed across multiple organizations which are only loosely coupled [3].

The remainder of this paper is structured as follows: in Section 1.1 we will introduce the key research questions of this paper and in Section 1.2 we will discuss related work. Section 2 will then introduce the case studies that we chose to analyze, Debian Linux and Eclipse. In Section 3 we will discuss to what extent our case studies are reasonable cases of variability management. On this basis we will discuss in Section 4 the main concepts that can be taken from these case studies to support distribution and in particular distributed variability management. Finally, in Section 5 we will summarize and conclude.

1.1 Research Questions

The main goal of this paper is to improve the understanding of how variability management can be effectively supported in the context of distributed development. As a basis for answering this question, we decided to analyze some existing highly configurable infrastructures in order to identify what works in practice. The examples we draw upon are Debian Linux [1] and Eclipse [9], respectively their package management. Both are large-scale, well-known projects, which do heavily rely on highly distributed development. Moreover, many independent organizations can contribute packages, realizing a rather low level of interdependence among the organizations. Thus, they show that these approaches can in particular be applied in the context of a software supply chain (respectively a software ecosystem).

Of course, it is non-trivial that the configuration approach which is used in the package management systems of Eclipse and Debian Linux can be compared to variability management approaches at all. In order to address this concern we will make an explicit comparison of these approaches with variability management techniques like feature modeling [11].

In summary, we will address the following research questions in this paper:

(RQ1) Can existing package management approaches like those of Debian Linux and Eclipse be regarded as a form of variability management?

(RQ2) Which concepts in these package management approaches can prove useful to handle distributed development, if any?

The main focus of our work will be on the aspect of distribution. Thus, while we will address the aspect of expressiveness, it is our main goal to identify concepts that might help variability management approaches to better cope with distribution.

1.2 Related Work

In this paper we analyze two existing software configuration systems and compare them with variability management approaches. So far such analysis of existing software configuration systems have been performed surprisingly few. A notable exception is the analysis of the Kconfig system used in Linux kernel development by She et al. [20].

A difference between the case studies we use here and case studies like [20] is that here configuration is performed rather late, i.e., at installation time. However, this is in accordance with recent developments in the product line community where later binding of variability is increasingly common (e.g., [23]). This is sometimes termed dynamic software product lines [10] and seems to be very relevant to software ecosystems [3].

A major aspect of distributed development is that the variability management needs to be decomposed as well. This has so far received very little attention. A notable exception is the work by Dhungana et al. [8]. They introduce the notion of decision model fragments which can be manipulated independently and integrated at a later point. They explicitly mention distributed development (type 1) as a motivation for their work. The feature diagram references mentioned by Czarnecki et al. [6] are also related as it seems their approach can also be used to support distributed development, though this is not made explicit.

Recently, Rosenmüller et al. also addressed the issue of integrating multiple product lines into a single, virtual variability model under the name of multiple product lines [16, 15]. A major disadvantage of their approach from the perspective of our discussion here is that it requires a single central point of control (the integration model). We expect this to problematic, in particular in the context of open variability, e.g., in software ecosystems [3].

In this paper, we do not aim at introducing a new approach. Rather we focus on how existing approaches from industrial practice deal with the problem of distributed development of a software product line, respectively a software ecosystem [3]. We will analyze these approaches, show that they are comparable to existing variability management approaches, while offering at the same time new insights that should be taken into account for designing future distributed variability management approaches.

2 Analysis of Case Studies

In this section, we provide an overview of the two case studies we will use as a basis for our analysis: the Debian Package Manager [1] and the Eclipse Package Manager [9]. While these are certainly not the only relevant, distributed software configuration systems, we restricted our analysis to these two as they are well-known, widely used and documentation of them is easily available. Also they are clearly examples of the problem of distributed and rather independent development. In particular they are used as a means to realize a software ecosystem [3]. We also considered to include Kconfig in our analysis, but decided against this for multiple reasons: first of all a good analysis like [20] already exists, although the focus was in this analysis not on distribution aspects. Second Kconfig supports the configuration of the Linux Kernel, which is released in a centralized way. Thus, we expected to learn less from this approach compared to the ones we selected for more general distribution models like software supply chains (cf. distribution 2) or software ecosystems.

Both the set of all Linux installations, as well as the set of possible Eclipse configurations can be regarded as a rather open form of product line respectively a software ecosystem. It is a product line, as the customer has even with a standard distribution of Linux such a large range of possible configurations, that actually each installed system can have a unique configuration. Further, it can be regarded as an open product line, as additional systems can refer to and extend the existing base distribution enlarging the overall capabilities and configuration space. It is this form of openness of the variability model which is particularly interesting to us from the perspective of distributed development.

It is important to note – and we will see this in more detail below – that the approach used by these management systems is completely declarative. This makes them rather easy to compare to variability management and is different from approaches like makefiles, which contain procedural elements and can thus not be directly compared to declarative variability management approaches.

Some concepts are common to both the Eclipse and the Debian Linux package management approach, thus, we will first discuss their commonalities, before we discuss the specifics of both systems in individual subsections.

Both Linux and Eclipse are actually aggregates that consist of a number of packages. So-called distributions are formed, which are collections of packages which are guaranteed to work well together. In the case of Linux the user can select a subset of these packages for installation, while in the case of Eclipse a specific distribution forms a set which is installed as is. The packages in a distribution are selected (and if necessary adapted) to work well together. This is the main task of the organization that maintains a distribution. It should be noted, however, that this organization is not necessarily responsible for the actual development. It is thus more appropriate to compare it to an integration department in traditional software development. There exists a configuration model as a basis for defining a specific installation. This is included in a distribution and is stored in a distributed (per-package) fashion, as we will discuss later.

As the various packages that belong to a distribution can be developed by a large number of different development organizations, we have already here the situation of distributed development, although there is an explicit harmonization step in the form of the maintenance of the distribution. It should be noted, however, that this is harmonization is not needed per se, as both approaches are open to the inclusion of arbitrary third party packages. The step of distribution formation mainly plays the role of quality assurance.

In both cases a package manager exists, which uses the descriptive information from the packages to provide the user[1] with configuration capabilities. Users can also install additional packages (e.g., from the internet) which are not part of the initial

[1] We will use the term *user* to denote any individual who performs a configuration process. In practice the user will typically fill the role of an administrator.

distribution. As these packages can be developed completely independently, this further enforces the notion of distributed development.

The user can use the package manager to create the final configuration. This step results in a specialized installation. It is peculiar of this approach that the binding time is rather late (i.e., the individual packages already contain compiled code), however, this is not a must, as even source code can be part of packages. This is in particular the case in the Linux environment. This difference in binding time may seem unusual, as typically product line engineering is equated with development time binding. However, already in the past different approaches have been described that go beyond this restriction (e.g., [23, 21, 17]). This is actually most pronounced in the context of dynamic software product lines (DSPL) [10].

2.1 Debian Linux Package Management

The installation packages for the Debian package manager consist of the following parts [1]:

- *Control-File:* the control file contains the necessary dependency information as we will discuss below.
- *Data File:* the data file contains the actual data to be installed. This can be source code, binary files, etc.
- *Maintenance scripts:* these are used to perform maintenance actions before and after the reconfiguration takes place.

The key information we are interested in here is contained in the *control file* as this contains all relevant dependency and configuration information. The control file provides administrative information like the name and the version of the package, but also dependency information. It also defines the required disk space and the supported hardware architecture. This constrains the situations when it can be installed.

For the dependency information in a package seven different keywords are defined: *depends, recommends, suggests, enhances, pre-depends, conflicts,* and *replaces*. The semantics of these keywords overlaps as discussed below:

- *Depends:* this keyword expresses that the following text provides information on other packages that are required for the proper use of this package. This can be combined with a version qualifier with the implication that the package requires this version (or a later one) of the package.
- *Pre-Depends:* similar to depends it defines that another package is needed *and* must be fully installed prior to installation of the current one. Thus in addition to the dependency information it provides execution information for the package manager.
- *Recommends:* this expresses that the package is typically installed together with the recommended packages. However, this is not a strict dependency, but rather a hint for the configuration, that the recommended packages should as well be installed (but a configuration will also be successful without them).
- *Suggests:* this expresses that the suggested packages could be installed together with the current one. This should be interpreted as a hint, it is similar to recommends, but should be interpreted in a much weaker form.

- *Enhances:* this defines that the package can be used together with the enhanced packages. This should be interpreted as the inverse relationship to *suggests*.
- *Conflicts:* this expresses that the current package cannot be used in combination with the mentioned packages.
- *Replaces:* this expresses that installation of files in the package will actually replace or overwrite information from the referenced packages.

As we can see the different keywords actually combine different aspects in a non-systematic may. These aspects are:

- *Dependency and conflict information:* this can be compared to the information typically contained in a variability model.
- *User guidance:* some information is only meant as a hint to the user of the configuration system (e.g., suggests, enhances). This is actually ignored by some package management tools [1].
- *Execution information:* information like pre-depends actually influences how the package manager performs the actual installation process.

Our main interest is of course on the dependency and conflict information. All of the relationships (except for *replaces*) that are introduced by the control files have some aspect of dependency and conflict information, although in some cases (e.g., suggests) this is not strict in the sense that the given guidance can be ignored by the user, respectively the installation system.

The remaining two parts of a package are the data file and the maintenance scripts. The *data file* is basically a packed archive. There are actually two slightly different formats, depending on whether the package contains source code or binaries (e.g., executables). Unpacking the archive generates all the necessary files, including directories, etc. This implies an all or nothing semantics, i.e., either the whole data contained is added to the installation or the package is not installed.

Finally, there are *maintenance scripts.* These are mainly important as the package manager may run while the system is actually running. The scripts are then used to start and stop services to allow their (re-)installation. Another application of these scripts is to customize the configuration process. For our analysis these scripts are not of further interest.

2.2 Eclipse Package Management

The Eclipse package management provides two concepts that are relevant to our analysis here: *feature* and *plug-in* [9]. The feature in the Eclipse terminology is a coarse-grained capability. Actually the typical user installation consists of only three major features [9]: platform, java development tooling, plug-in development tooling.

A feature by itself does not contain any functionality, rather it describes a set of plug-ins that provide the functionality. In addition to acting as a sort of container for the actual plug-ins it provides management information like where to get updates for the feature.

A plug-in consists of a so-called *manifest* which provides a declarative description of the plug-in and the relevant resources. The resources contain a set of java classes, which implement the functionality, but may contain also other resources like scripts, templates, or documentation.

The main part, we are interested in here, is the manifest. It declares plugin name, id, version, provider, etc. It also defines the dependency information for the plug-in. Dependencies are defined in the *requires*-section of the plug-in manifest [9]. This section declares other plug-ins that are needed for successful usage of the current one. The plug-ins can be further annotated with version information.[2] In addition the *requires*-information can be further refined as *optional*. The semantics of an optional requires is that the referenced plug-in should be integrated into the platform if it is found, but if it is not found the installation of the current plug-in is still possible (as opposed to a pure requires). On the other hand, Eclipse does not provide any way to express that a plug-in is mutually exclusive (conflicts) with another plug-in. The requires-information can be further refined by making restrictions with respect to the versioning information. This is supported by a proposal for the semantics of the version numbering by Eclipse. Specific relations on the versions include: perfect match, equivalent, compatible, greaterOrEqual.

The Eclipse feature and plug-in mechanisms also support some sort of modularization. This is expressed by exports and extension points.

The classes that make up the plug-in can also be exported, enabling other plug-ins to explicitly refer to them.

In addition the manifest may declare extension points and extensions. The extension point architecture of Eclipse is a core part of its extensibility. Any plug-in may define extension-points. This means it will allow explicit, external customization of its functionality. A plug-in may also refer to an extension point and extend it. Typical examples for extension points within the Eclipse-IDE are menu entries or additional views.

3 Package Managers as a Form of Variability Management

In this section, we will focus on the question of whether the package managers, described above can be seen as a form of variability management (RQ1). In order to characterize variability management several formalizations of variability modeling, like [19, 7, 2] have been developed. As we will see, the package managers only support rather simple concepts of variability management, thus a simplified treatment of variability management is sufficient in this context.

3.1 Variability Management Concepts

As a first step, we need to establish a mapping between the typical concepts used in package managers and in variability management.

If we take as a basis for variability management the concepts from feature modeling, as they are described in a formal definition like [19], we find that a feature model (represented by a feature diagram) can be characterized in the following way (we use here the terminology introduced in [19]):

- *Graph Type:* this can either be a graph or a tree.
- *Node Type:* possible node types; these are Boolean functions like *and*, *or*, *xor*. This describes how nodes can be decomposed. Also cardinalities *(card)* are classified as node types.

[2] For identifying required versions it is possible to define constraints as exact matches, compatible, etc. This is implemented using a specific version naming scheme.

- *Graphical Constraint Type:* is a binary Boolean operator, e.g., *requires* (⇒) or *mutex* (|).
- *Textual Constraint Language:* is a subset of the language of Boolean formula and is used to describe the allowed compositions.

In addition, it should be noted that edges within a feature diagram denote a decomposition. The allowed forms of decomposition are expressed by the node type.

The mapping of the main concepts in package managers to such a variability modeling language are not straight-forward as the relationships are somewhat different and the package managers do not support a diagram notation. Thus, we will discuss this mapping here explicitly. In support of the discussion Figure 1 illustrates the main concepts we will deal with.

The first major difference between the package management approaches and feature diagrams is that the approaches are textual not graphical. We can thus ignore the difference between the graphical constraint type and the textual constraint language. The nodes in the package management approaches correspond to packages (in Debian Linux), respectively plug-ins in Eclipse. It should be noted that the concept of features as it is introduced in Eclipse is rather coarse-grained and describes actually a conglomerate of plug-ins while the basic level on which dependencies are expressed are on the level of individual plug-ins. This is shown in Figure 1 by illustrating packages, but having the relations on the level of the individual contained units.[3]

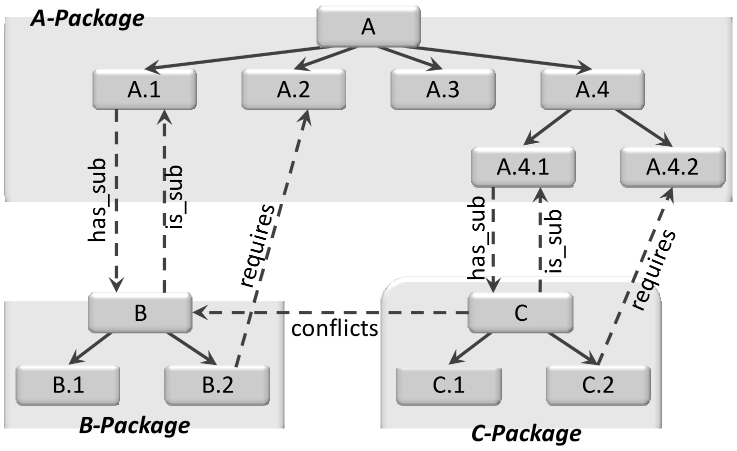

Fig. 1. Main concepts of variability in package managers

A major concept in most feature modeling approaches is that the features are decomposed in the form of a graph or tree. This is shown in Figure 1 as *has_sub* and is described by the node-type as defined above. It should be noted, however, that such a decomposition approach is not part of all forms of variability management approaches.

[3] This corresponds closely to the situation in Eclipse, in the Debian Linux situation the packages shown in Figure 1 have no correspondence.

For example, some forms of decision modeling [18] or the approach described in [14] do not rely on decomposition. Both package managers do also not have the concept of decomposition. However, still often a tree-like or graph-like dependency structure is introduced. This can be done using the *requires*-links. This can be seen as *has_sub*-relation shown in Figure 1. On the other hand, as the various packages can be selected individually and are interrelated mainly by requires-relations, this relation can also be regarded as the inversion of the *has_sub*-relation, i.e., the *is_sub*-relation. Thus, in both approaches, both relations *has_sub* and *is_sub* are replaced by requires, if they are represented at all.

3.2 Analyzing Package Management as Variability Management

We will summarize the expressiveness of the two package management approaches and will use this as a basis to compare them with variability management as described by the characterization from [19].

As discussed in Section 2.1 the Debian Linux package management has as main concept the package. We can equate this with a node. As a basis for dependency management the relations *depends* and *conflicts* can be regarded. The *depends*-relation can be equated with requires. The *conflicts*-relation is not exactly the same as the *mutex*-relation, defined in [19], as it is not symmetrical, thus we keep the name *conflicts* in Figure 1. However, *conflicts* effectively emulates the *mutex*-relation as it does not allow a configuration in which both packages participate. Noteworthy is also the *recommends*-relation as it has the same direction as the *has_sub*-relation. However, it is weak in the sense that it does not require the installation. The other relations defined by the Debian Linux package management approach (*pre-depends, suggests, enhances, replaces*) are variations of the mentioned relations, but augment it with additional user advice or execution information. They do not provide any new information from a logical point of view.

Below, we discuss the comparison in more detail:

- The most striking difference is probably that the decomposition hierarchy which is typical for feature diagrams is not directly a part of the dependency management defined by the package management approaches. As a consequence the resulting structure is not one continuous graph, but is rather a set of graphs. Further, the hierarchy in the sense of decomposition cannot be directly expressed, but can only be simulated using a requires-relationship. While both aspects seem unusual for feature-based variability management, they exist in other variability management approaches like [18, 14, 7] as well.
- The *and*-, *or*-node types can be expressed as described in Table 1. The *xor*-node type can only be represented for the Debian Linux package manager by means of the conflict relationship. This is not available for Eclipse and can thus not be simulated there. More advanced concepts like cardinality do not exist in either approach.
- Constraints are always textual constraints, as there is no graphical notation for both approaches, thus we discuss Graphic Constraint Type and Textual Constraint Type

together.[4] Again the Debian Linux approach allows the representation of *mutex*- and *requires*-constraints (using *conflicts* and *depends* relations, respectively). The Eclipse package manager falls short as it can represent the *requires*-relation, but not the *mutex*-relation. More complex combinations (in the sense of complex formula) are not available in either approach.

Table 1. Comparison of package management and a characterization of feature models ([19])

Concept [19]	Debian Linux Package Manager	Eclipse Package Manager
Graph Type	Graph*	Graph*
Node Type	and, or – all elements that are required must be installed, but weaker versions like suggests actually provide optionality xor – can be simulated by using the conflicts relation	and, or – the requires relationship can be augmented with an optional modifier
Graphical Constraint Type (GCT)	requires – is supported using depends mutex – the conflicts relationship is a directed variant of mutex	requires – exists as a relation mutex – does not exist nor can be simulated
Textual Constraint Language	From a logical point of view only the concepts mentioned under GCT are supported, although the language is completely textual. There exist extensions like references to specific version, which do not exist in variability modeling techniques for product lines.	

*: As there need not be connections between nodes that are installed, either induced by requires or any other relations, it might actually be more appropriately regarded as a set of graphs.

If we accept the use of the requires-relation to describe the decomposition, we can deduce from [19] that the expressiveness of the Debian Linux approach is at least similar to FODA [11].[5] More problematic is the Eclipse package manager, which does not support a form of exclusion (similar to alternatives, mutex-relations, conflicts-relation, etc.). As a consequence, we need to accept that this approach is truly weaker than other variability management approaches.

According to the above comparison we can deduce several findings. The first and most important is that we can answer (RQ1) with yes for the Debian Linux approach,

[4] Again, it should be noted that this also exists in other variability modeling approaches. For example the approach described in [18] only provides a semantic approach without prescribing a specific representation (graphical or otherwise).

[5] This is not fully correct, as FODA allows parameterization, which is not present in the Debian Linux model. However, this is also not part of the analysis given in [19].

albeit it truly provides only minimal expressiveness (comparable to FODA). For Eclipse, answering (RQ1) is not easy, as the expressiveness is less powerful than any of the variability management approaches given in the literature, due to the lack of the *mutex*-relation. We will thus answer (RQ1) for Eclipse only as partially, however, many elements of a variability management approach exist. Thus, it forms a valid basis for our comparison in this paper.

In summary, we can say that both approaches lack in comparison with modern variability management approaches significant expressiveness. Examples for this are cardinalities, complex constraint conditions, etc. From this perspective it is surprising that both approaches work very successful in practice. One reason for this is certainly their limited application domain.

4 Concepts in Package Management That Support Distribution

After discussing whether the two package managers can actually be regarded as a form of variability management, we will now turn to the question what concepts exist in the two package managers which may prove useful for variability management, in particular in a distributed setting (RQ2).

We will structure our discussion along the following topics:

- Decomposition
- Version-based dependency
- Information Hiding
- Variability Interfaces

Decomposition: Probably the most immediate observation one can make when analyzing the package managers is that they take a decomposition of what shall be managed for granted. Of course this flows well with distribution. In Debian Linux the basic concept is a package, but it should be noted that the contribution of a distributed part of the development is often contained in several packages that are related with each other (e.g., an implementation and a source package). In Eclipse the feature concepts defines such a unit of distribution and may contain several plug-ins. However, also in the Eclipse case sometimes several features are developed and distributed together, as the feature is also a configuration unit.

From the perspective of distributed variability management this decomposition also leads to a decomposition of the variability model as this enables to assign responsibility for different of the variability model to different teams. We believe this is one major characteristic to support distributed variability management. Further, if we look at the way the relations are typically used within the package management approaches, we see that the *has_sub*-relation is typically not used across packages that build on each other, but rather *requires* is used in the form of the *is_sub*-relation in Figure 1 (or across the hierarchy as also shown in Figure 1). This leads to the package that builds on top of another one to know the package on which it is building, but not vice versa. Thus a platform can be built without the need to know everything which will build on top of it at a later time. We regard this as an important difference to the decomposition hierarchy in feature models and term it *inversion of dependency*. It should be noted, however, that for other variability management approaches that do not have a decomposition hierarchy this is straightforward. We regard this *inversion of dependency* as very important for developing future software ecosystems, using product line technologies [3].

Version-based dependency: the capability that all packages may have versions and that the dependency management may use the version information to define what combinations are acceptable is very useful to reduce the coupling between packages. Thus, packages may still work together, even if their content changes and this may be explicitly expressed in the relations. This is particularly prominent with the Eclipse package manager, which even defines different forms of changes (and can put constraints on the acceptance of the different compatibility levels). This enables the specification of the degree of decoupling of the development of the different packages that is possible.

Information Hiding: Of course explicit variability management always leads to some form of information hiding as the potential configurations only depend on the features defined in the variability model, not on the base artifacts. However, Eclipse goes one step further by explicitly defining what parts of the packages will be visible to other packages. This concept does not exist in classical variability modeling, as there no information hiding is induced on the basic artifacts. It is interesting to note that this is similar to packages in the UML which also allow to restrict the visibility of their content. Also information hiding in Eclipse goes hand in hand with explicit variability interfaces.

Variability Interfaces: A very interesting mechanism within the Eclipse package management approach is the extension point approach. Plug-ins can announce specific extension points where further variability is possible and can be resolved by other packages. Typical examples of this are menus, views, etc. that can be augmented by further plug-ins. Eclipse even introduces a schema definition for extension points. While this would allow the definition and verification of the parameterization of the extension point, this schema is currently not yet used in the Eclipse implementation according to the latest description [9].

The concepts of *decomposition*, *information hiding* and *interfaces* are all well-known. They are typical of what we call today *modularization* [13]. However, if we compare this with almost all existing variability modeling approaches, we have to recognize that they are monolithic in nature (for a discussion of the exceptions see Section 1.2). It seems reasonable to assume that the modularization concepts that are useful in the construction of every software systems, in particular in a distributed manner are as well useful in distributed variability modeling. This is emphasized by the success that the discussed package management systems already have in industrial practice today were they provide a foundation for respective software ecosystems.

The concept of *version-based dependency* further supports the decoupling of the variability packages. Both discussed approaches posses this capability in some form. Finally, the *inversion of dependency*, i.e., that only the refining variability package needs to know the refined package, but not vice versa seems rather useful, as it further decouples distributed development.

Thus, we can answer (RQ2) positively: there are certain concepts that have been introduced in package management approaches that are helpful for distributed variability management in software product line engineering. The key concepts we could identify are: decomposition, version-based dependency, information hiding, variability interfaces, and inversion of dependency. Out of these only decomposition has to our knowledge been applied so far [15, 8, 6]. The other concepts introduce some form of modularization to provide a better basis for decoupling in distributed product line development.

5 Conclusions

In this paper, we analyzed existing, large-scale, real-world package management approaches to identify concepts they can offer for distributed variability management. We established (RQ1) that these approaches can indeed be interpreted as a form of variability management as they support the declarative definition of variation constraints. The concepts they support for dependency management can be mapped onto existing variability management approaches. However, we had to recognize that only very fundamental concepts are realized. Thus, these tools may significantly profit by integrating more advanced capabilities from modern variability management approaches.

On the other hand these package management approaches have been developed from the beginning to support distributed development and integration. Both provide today the basis of a software ecosystem of their own. As a result they offer a slightly different approach and concepts that can be well integrated into product line engineering to support improved modularity of variability management. We can thus answer our second research question (RQ2) positively. The concepts we identified are:

- Decomposition
- Version-based dependency
- Information hiding
- Variability interfaces
- Inversion of dependency

However, it should be recognized that these concepts, while certainly useful for augmenting existing variability management techniques, they can still be further improved. For example, it would be useful to provide a formal definition of variability interfaces. As a result of the introduction of the above concepts together with further work, we expect that in the future we will arrive at a meaningful theory on the modularization of variability models [13]. We expect that such a theory will be particularly relevant in the context of open and distributed variability as is required for software ecosystems [3].

As a next step we plan to extend our work and will address in particular software ecosystems more in depth. We will also extend our basis of analysis further to cover a larger range of existing approaches. The results of this analysis will then drive the development of an integrated approach for dependency management that combines the strengths of established practical approaches with the best of existing, sophisticated variability management techniques.

References

[1] Aoki, O.: Debian Reference, (2007), http://qref.sourceforge.net/Debian/reference/reference.en.pdf (last verified: 13.3.2009)

[2] Benavides, D.: On the automated analysis of software product lines using feature models. A framework for developing automated tool support. PhD thesis, University of Seville, Spain (2007)

[3] Bosch, J.: From software product lines to software ecosystems. In: Proceedings of the 13th Software Product Line Conference, pp. 111–119 (2009)

[4] Catalog of software product lines, http://www.sei.cmu.edu/productlines/casestudies/catalog (last verified: 13.03.2010)

[5] Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston (2002)

[6] Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multi-level configuration of feature models. Software Process Improvement and Practice 10(2), 143–169 (2005); Special Issue on Software Product Lines

[7] Dhungana, D., Heymans, P., Rabiser, R.: A formal semantics for decision-oriented variability modeling with dopler. In: Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2010), pp. 29–35 (2010)

[8] Dhungana, D., Neumayer, T., Grünbacher, P., Rabiser, R.: Supporting the evolution of product line architectures with variability model fragments. In: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture, pp. 327–330 (2008)

[9] The Eclipse Foundation. Eclipse 3.1 Documentation: Platform Plug-in Developer Guide (2005), http://www.eclipse.org/documentation (checked: 13.3.2009)

[10] Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. Computer 41(4), 93–95 (2008)

[11] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21 ESD-90-TR-222, Software Engineering Institute Carnegie Mellon University (1990)

[12] van der Linden, F., Schmid, K., Rommes, E.: Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering. Springer, Heidelberg (2007)

[13] Parnas, D.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)

[14] Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)

[15] Rosenmüller, M., Siegmund, N.: Automating the configuration of multi software product lines. In: Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2010), pp. 123–130 (2010)

[16] Rosenmüller, M., Siegmund, N., Kästner, C., ur Rahman, S.S.: Modeling dependent software product lines. In: GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE), number MIP-0802, pp. 13–18. University of Passau (2008)

[17] Schmid, K., Eichelberger, H.: Model-based implementation of meta-variability constructs: A case study using aspects. In: Proceedings of VAMOS 2008, pp. 63–71 (2008)

[18] Schmid, K., John, I.: A customizable approach to full-life cycle variability management. Science of Computer Programming 53(3), 259–284 (2004)

[19] Schobbens, P.-Y., Heymans, P., Trigaux, J.-C.: Feature diagrams: A survey and a formal semantics. In: Proceedings of the 14th IEEE Requirements Engineering Conference (RE 2006), pp. 139–148 (2006)
[20] She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: The variability model of the linux kernel. In: Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2010), pp. 45–51 (2010)
[21] van der Hoek, A.: Design-time product line architectures for any-time variability. Science of Computer Programming 53(30), 285–304 (2004); Special issue on Software Variability Management
[22] van Ommering, R.: Software reuse in product populations. IEEE Transactions on Software Engineering 31(7), 537–550 (2005)
[23] White, J., Schmidt, D., Wuchner, E., Nechypurenko, A.: Optimizing and automating product-line variant selection for mobile devices. In: Proceedings of the 11th Annual Software Product Line Conference (SPLC), pp. 129–140 (2007)