Configuring Software Products with Traditional Methods – Case Linux Familiar

Katariina Ylinen[1], **Tomi Männistö**[1] and **Timo Soininen**[1]

Abstract. Recently, the software industry has begun to adopt a product family approach. Support tools for tailoring configurable product families have been developed for some time for mechanical and electronics products. However, it is not clear that the modeling languages designed for these configurators are suitable for software product configuration. In this paper, we investigate this problem by modeling a Linux Familiar operating system distribution with a configuration modeling language aimed at representing the structure of physical products. Findings from this case study suggest that the language is largely suitable for software product configuration. However, some phenomena in the product strongly suggest that modeling them as functions, features or resources and optimality criteria familiar from the configuration domain would be useful. In addition, there is some evidence that deeper modeling of versions of components and reconfiguration knowledge, not usually covered by configuration models, should be supported.

1 INTRODUCTION

Configurable product families have been an important phenomenon in the mechanical and electronics product domains for some time. In these domains, called *traditional products* for short, it has been noted that it is possible to satisfy a wide range of customer requirements with relatively low costs by designing the products to be routinely configurable. Systems that support tailoring such products, *configurators*, have been extensively developed and studied [4]. Recently, the software industry has to some extent adopted the point of view that designing software product families and delivering variations of them rather than customizing individual products can be far more cost-effective [1]. However, it is not clear that configurators supporting traditional products are suitable for software product configuration. The software community has approached product families from a different perspective and developed its own modeling methods and methods for tailoring the products [15,16] .

In this paper, we investigate whether there are any differences between the two domains of traditional products and software by means of a concrete case study. We try to model a Linux Familiar operating system distribution [13] with a configuration modeling language aimed at representing the structure of physical products [11, 14]. The three research questions we ask are: Is the configuration modeling language adequate for representing the Familiar configuration knowledge? If not, what are the most important missing elements? And finally, which of the common concepts used for modeling configurable traditional products [5] could be used to make the models more useful?

The rest of the paper is organized as follows: We first briefly review and compare the two domains of product configuration and software families and analyze their similarities and differences in section 2. We then present the case product and the related configuration problem and present an analysis of the configuration knowledge for the product in section 3. We present a way to model the configuration knowledge of the case product using a language for modeling traditional products in section 4. The findings of our case study are presented and discussed in section 5. Finally, in section 6 we present our conclusions and identify some topics for further research.

2 BACKGROUND

This section briefly describes the domains of product configuration in both traditional industry and in the software industry and then compares the frameworks used in them to find their similarities and differences.

2.1 Physical Product Configuration

For a *configurable product family,* each *product individual* is adapted to the *requirements* of a particular customer order on the basis of a predefined *configuration model,* which describes the set of legal *product variants* [6,7]. A *configuration*, that is, a specification of a product individual is produced from the configuration model and particular customer requirements in a *configuration task*.

Knowledge based systems for configuration tasks, product *configurators,* are an important application area of artificial intelligence techniques for companies selling products adapted to customer needs [9,10]. Product configuration tasks and configurators have been investigated for at least two decades [11].

Conceptualization of configuration knowledge synthesizing these approaches is reported in detail in [7,17].

2.2 Software Product Configuration

The software professionals often claim that the methods and processes of traditional industry do not apply to software development and products. It has been suggested that the product configuration concepts and methods used for the physical products are not directly applicable in the software industry [2]. According to Brooks [6], software has four special features that make it special: complexity, conformity, changeability and invisibility.

Even though modern physical products may also be complex and invisible in nature, they are constructed of a limited number of parts that are replicated. In a software product variant, there are usually no two parts equal to another.

In the past, much of the configuration-related software management research has focused on the software configuration management (SCM). In SCM, the focus has primarily, although not completely, been on managing the evolution of the source code files

[1] Helsinki University of Technology, Software Business and Engineering Institute SoberIT, P.O. Box 9600, FIN-02015 HUT, Finland. Email: {Katariina.Ylinen, Tomi.Mannisto, Timo.Soininen}@hut.fi

[7]. Most SCM systems today only manage the software systems as a set of files instead of as configurable product components [7].

The research of configuring final software products has only recently become an increasingly important challenge while the industry has slowly adopted a goal of producing product families instead of single products. A related field, which aims at modeling product families, is the research on architectural descriptions [17]. Even though the primary goal of ADLs has not been the product configuration task, it can be noted that they have adopted significantly similar concepts as the traditional configuration research.

While formal methods for software product configuration have been absent, there has been separate efforts for configuring single products. Most of the software companies implement configurability by producing one big product with all variants included. The configuration task is then done using, for example, preprocessor directives or makefiles to define the specific product variant [2].

Another result of software invisibility is that in software products, it is often impossible to know the correct order of installation, if there are limitations to this. This may often be as complex in the physical products as well. The difference lays in the usage – software products are often installed at the same time as they are configured. The installation and configuration process is, especially in consumer products, assumed to happen at the same time and at least semi-automatically. This has led to a situation where installation programs do the configuration and vice versa.

The adequacy of the configuration knowledge present in the packages of Debian distribution has been a subject for study before [3, 12]. The approach has been developing a new rule-based language for modeling the packages and dependencies, and then implementing this using a logic-program-like rule based system. We, on the other hand, aim at applying the high-level modeling concepts of physical product configuration to modeling Linux Familiar.

3 FAMILIAR CONFIGURATION PROBLEM

Linux Familiar was chosen as the case since it is developed for a handheld computer, Compaq iPAQ, and therefore faces resource limitations, such as the amount of memory available. Whereas operating systems on PCs can be installed with all bells and whistles included, the handheld devices make prioritization of installed components necessary. There are currently about 1700 Familiar packages available in the distribution, only a subset of which fits into a device at the same time. The packages vary in nature from necessary Linux kernel packages to application packages. Linux was chosen instead of other handheld operating systems due to its openness, as it was easier to study.

Linux is a monolithic operating system. The part implementing the core functionality of the operating system such as the thread management, file system etc. is called the Linux kernel [9]. The kernel itself is an interesting entity from the configuration perspective, since it has built-in support for dynamic reconfiguration. However, this feature has been implemented for mainly memory management use and is left out of the scope of this paper. We focus our attention on package management, that is, a higher-level configuration management of the whole distribution.

The Familiar Distribution is derived from the Debian distribution, which means the configuration attributes share the same concepts. Both distributions consist of a large amount of software components, called packages. Packages consist of several files, which can be either executables or other files as well. In Debian, as well as in most widespread Linux distributions, there is a package management system, which manages the installation and removal of packages in the system. In Debian, this program is called dpkg and it manages configuration constraints like dependencies and conflicts between packages, virtual packages and installation order. It can also be used to create and purge packages.

Due to the size constraints dpkg is not a part of Familiar but a lighter version of it called ipkg has been developed. Ipkg shares the basic functionality in package installation and removal but lacks most of the configuration validity functions. Therefore, the comparison with the traditional configurator is made against dpkg features. Familiar packages are equipped with configuration information of the same syntax, and they can still be used as an example of the input data.

The configuration language of Familiar / Debian is fairly simple. It is a list of the package information in pure text format. There are two fields attached to all packages: version and the package name. In addition, it lists all constraints known for the package. The constraint clauses also refer to a package name and optionally to its version. An example of the fields concerning configuration in a Familiar package description:

```
Package: xlibs
Priority: optional
Version: 4.0.2-13
Replaces: xlib, xbase (<< 3.3.2.3a-2), xlib6 (<< 3.3.2.3-2)
Provides: libxpm4
Depends: xfree86-common (>> 4.0), libc6 (>= 2.2.1-2)
Conflicts: xlib, xlib6 (<< 3.3.2.3-2), xlib6g (<< 4.0)
```

Figure 1

The different types of these packages is explained in more detail in section 3.1. The nature of the different relationships between the packages is analyzed in section 3.2.

3.1 Package Types

Currently, three types of packages can be identified in the system. The package types are *virtual, concrete* and *task* packages.

A virtual package is an abstract package that does not actually exist and, the functionality of which can be provided with one or more concrete packages. The virtual packages do not appear on the package lists as package definitions and therefore cannot have relationships to other packages. Instead, some of the concrete packages refer to a virtual package as they provide its functionality.

In Debian, the developer community strictly controls the virtual package names and the full list of available virtual packages can be found at the developer web site [10]. The same virtual package names are used in Familiar, although it seems that the control is a bit lighter and there are some additional, non-documented ones as well. An example of a typical definition of virtual package (x-terminal-emulator) in a Familiar package description looks like this:

The concrete packages are the actual packages that can be installed or uninstalled on the device and that have relationships to other packages on the system. The concrete packages consist of one or more files to be saved on the file system. These files can be either executables or other files, for example text files.

The third package type, task package, is a package that consists of several other packages. It has no own separate functionality but just the collection of the packages it contains. The package does not withhold any important functionality itself but has several dependencies to other packages, so that installing the package requires then installing the whole set. The task packages seem to consist of a set of different packages that together implement certain functionality. The task packages in Familiar are separated from concrete packages

with a naming convention. The package names have prefix "task-". The task packages are made to make basic installations easier for the end user by collecting a potentially important set of related packages into one package.

3.2 Relationships

As mentioned earlier, the package descriptions include constraint clauses. Every package has a listing of these as presented in example in figure 1. The clause first describes the nature of the relationship and then a list of package names and their versions.

There are seven different kinds of constraint types:

- Depends <*package B*> - this package requires package B to be installed on the device to function correctly.
- Pre-depends <*package B*> - it is required that package B is installed on the device before this package can be installed.
- Conflicts <*package B*> - this package should not be present in the same configuration with package B.
- Provides <*package B*> - this package provides all the functionality and files present in package B.
- Replaces <*package B*> - the installation of this package removes or overwrites files of package B.
- Recommends <*package B*> - the package B is recommended when it is presumable that the users would like to have it in the configuration with this package.
- Suggests <*package B*> - the package B is suggested to get better use of the package.

Of these, the constraints recommends and suggests are not relevant for checking the configuration validity but only useful hints for the user for configuration optimization. It can be also noted that the *pre*-depends clause is used to gain correct installation order but does not differ from depends when used to check the configuration validity.

For all the different relationships, there can be many packages listed and in that case the relationship holds for all the packages in the list. That is, the commas between the package names can be interpreted as Boolean AND. For depends, there can also be a Boolean OR, which is indicated by " | " in the list. The precedence rules are not very clear and we are not sure for which elements the OR statement refers to in some occasions. It is not, however, a big problem since there are not many OR statements currently in the package descriptions.

Depends is a simple relationship. When a package depends on another, the other must also be installed on the system. When checking the configuration validity, pre-depends is also treated the same way.

There are also some interesting exceptions in the use of "Depends" relation, which we consider more or less misuse and not usage rules. For example, the package *xlibs* depends on its own older version. This means there are also some incremental packages – the newer version is not actually a new version of the package but some additional features for it.

Conflicts is just as simple. When a package is in conflict with another, they should not be installed on the same system. There is, however, a minor problem in the language and the configurator, dpkg, considering this. As a package is installed in the system, only the relationships of the installed package are checked. In case there is a package in the system conflicting this new package, it may not get noticed. This is due to the fact that as the conflict has been noticed, it is highly probable that it only has been declared in the description of one of the packages but not in both. This means that either the conflict information must be duplicated to both of the package descriptions or the functionality of dpkg should be changed so that it would check all package descriptions of the packages installed on the system to check the configuration validity.

Provides has been designed for managing packages, which in some way implement functionality available in some other package. Unlike the other configuration clauses, the provides has been used quite systematically but it has two different tasks. One of them is the usage for version control when package naming has changed during versions. The newer version provides the older one, and it seems that the potential dependencies from other packages to the older version will not get broken. Provides allows the coexistence of the provided and the providing package in a configuration. In the case of renaming an existing package when making a new version, it must be separately specified with a conflict clause that the versions should not be present in a configuration at the same time.

Provides is also used for virtual packages – several concrete packages can provide some virtual package functionality, and many of them can coexist on the configuration [3]. For example, a concrete package *rxvt*, which is an emulator program emulating the x-terminal, has a row: "Provides: x-terminal-emulator" in its package definition. The *x-terminal-emulator* is a virtual package, and packages like *rxvt* (a basic x-terminal emulator) or *rxvt-aa* (an x-terminal emulator with anti-aliased fonts support) could coexist on the system both implementing the virtual x-terminal emulator package.

An example of the virtual package use of Provides in Familiar package description is presented in picture 2.

```
Package: rxvt
Version: 1:2.6.3-8-fam6
Provides: x-terminal-emulator
Depends: libc6 (>= 2.1.97), xlibs (>= 4.0.1-11)
Conflicts: suidmanager (<< 0.50)

Package: rxvt-aa
Version: 1:2.6.3-8-fam6
Provides: x-terminal-emulator
Depends: libc6 (>= 2.1.97), xlibs (>= 4.0.1-11), libxft, libxrender
Conflicts: suidmanager (<< 0.50)
```

Figure 2

Replaces is quite ambiguous and is therefore used in various ways, some of which can be clearly interpreted as designer errors. It is also the most dangerous one in use, since there is no such system functionality that would remove the replaced package beforehand. As there is no proper information on which way the replacement exactly takes place, there is no guarantee on what will happen to the files installed originally with the package being replaced. In practice, the replacement is therefore only used in some cases when user tries to install two conflicting packages. In these cases, when one of these packages is marked as replacing the other, the installer in Debian prefers the replacing one [3].

Replaces is used, as already stated, very differently in different packages. In some packages, it is used similarly as provision clause. As the replacing package should replace, by definition, some of the files of the replaced package, it is also dangerous that there is no solution so far for the case when a user would try installing the replaced package back to the system. The replacement clause is again only written in the other package and thus overwriting the same files back again would most probably break the system. This may be one reason for the fact that the replacement has not been

often used in actual replacing of packages. One example of an unexplainable use of this relationship is the package set util-linux (2.11b-2-fam2), fileutils (4.0.43-1) and shellutils (2.0.11-5). Both fileutils and shellutils replace an older version of util-linux. Still, they seem not to provide the functionality specified in the util-linux description. There are no other relationships between these packages.

We conclude that the most common use for replacement is that the replacing package provides at least some of the functionality of the replaced one, but it is not promised that it works the same way. This means the dependencies from other packages to the replaced package will get broken when it gets replaced, unlike in the case of provision. This cannot, however, be generalized for all packages due to the very varying use of the concept.

4 FAMILIAR AFTER MAPPING

In this section, the new configuration model for Linux Familiar is introduced. The syntax of the used language is presented in section 4.1 and the mapping of components and relationships are presented in sections 4.2 and 4.3, respectively.

4.1 Modeling Language

We use a language based on a subset of a general configuration ontology [5] to try to model the Familiar configuration information to gain understanding of how suitable the concepts of the language are for modeling software.

The used language, called *PCML,* is introduced in [11]. The main concepts of PCML are *component types*, their *compositional structure*, *properties* of components, and *constraints*. Component types define intensionally the characteristics (such as parts) of their *individuals* that can appear in a configuration. A component type is either *abstract* or *concrete*. Only an individual directly of a concrete type is specific enough to be used in an unambiguous configuration. A component type defines its direct parts through a set *of part definitions*. A part definition specifies a *part name*, a set of *possible part types* and a *cardinality*. A component type may define properties that parameterize or otherwise characterize the type. A *property definition* consists of a *property name*, a *property value type* and *a necessity definition.* Component types are organized in a *taxonomy* or class hierarchy where a *subtype* inherits the property and part definitions of its *supertypes* in the usual manner.

Constraints associated with component types define conditions that a correct configuration must satisfy. The first level building blocks of the constraint language are *references* to access parts and properties of components, and constants such as integers. References can be used in succession, e.g. to access a property of a part. Boolean returning *tests* are constructed out of references, constants, and arithmetic expressions. Tests also include predicates that allow checking if a particular referenced individual exists or is of a given type. Property references can be used with constants in arithmetic expressions that can be compared with the usual relational operators to create a test. Test can be further combined into arbitrarily complex *Boolean expressions* using the standard Boolean connectives.

4.2 Components

The component types we define for Linux Familiar configuration are the package types we identified in section 3.1. In addition, we define a component type defining the whole system, of which the packages are parts. We define a type hierarchy, in which there is one supertype *package*, of which all packages are subtypes. Package is an abstract component type with one property, version, which is of value type string. Both concrete and virtual packages are subtypes of this component type. It is a subtype of the root component type of the language, Component.

The virtual component types are defined as abstract, as there should be no occurrences of component individuals of them in a valid configuration. The conceptual meaning of the abstract component is seen as the same as that of the virtual package. All the virtual packages are modeled as subtypes of the Package component type. The concrete packages implementing a virtual package, are subtypes of the virtual package in question. Other concrete packages will all be subtypes of the component type Package.

Some of the data on the packages that is needed on configuration time are modeled as properties for concrete packages. These are priority, size, installed-size, and version. Priority will be modeled with the concept of cardinality. When a package is obligatory (priority: required), its cardinality is 1 and when optional, it is 0 to 1. Virtual packages will not have these properties since this information is not available for them.

4.3 Relationships

Relationships between the packages are modeled as constraints in the configuration model. The relationships are translated as follows:

A Depends B	constraint <constraint_name> A implies B
A Pre-Depends B	constraint <constraint_name> A implies B
A Conflicts B	constraint < constraint_name> not (A and B)
A Provides B (virtual packages)	subtyping, A is a subtype of an abstract component type B
A Provides B (change of name between versions)	subtyping and conflict, A is a subtype of B and constraint <constraint_name> not (A and B)

As many of the relationships also involve version numbers, they must be taken into account in the constraints. Constraints involving version numbers are modeled as the following example demonstrates (conflict):

```
A Conflicts B (>= 2.2.1)   ⇨
constraint <constraint name>
not(A and B and
<part name for B>.B:version >= “2.2.1”)
```

4.4 A Sample

Figure 3 presents a sample of the model created with PCML:

```
configuration model TestModel

 # The concrete package type
 component type Package
  abstract
  subtype of component
     property version value type string

 # The virtual package x_terminal_emulator
 component type x_terminal_emulator
  abstract
  subtype of Package
```

```
# Package rxvt implementing x-terminal-emulator
 component type rxvt
  subtype of x_terminal_emulator
     property version
        value type string constrained by $
        in list ("1:2.6.1", "1:2.6.3-8-fam6")
  . . .
component type OSystem

  part x_terminal_emulator
    allowed types x_terminal_emulator
    cardinality 0 to 1

  constraint only_one_version_rxvt
    present(x_terminal_emulator) and
    x_terminal_emulator individual of rxvt
    and x_terminal_emulator.rxvt:version
        = "1:2.6.1"
```

Figure 3

In this example, we use the same virtual package as in figure 2, 'x_terminal_emulator. Two concrete packages, rxvt and rxvt-aa, implement its functionality. There are two versions of each available, but for a valid configuration, only the older version of rxvt is accepted.

5 DISCUSSION AND FUTURE WORK

Linux Familiar does not represent the configuration aspects of all software products. However, it is an easily studied real-world example of configurable software. On the basis of the modeling approach defined earlier, it seems that the methods of physical product configuration are at least partially applicable to software products as well. The differences between these domains are not so remarkable that entirely new methods and modeling languages need to be developed for software configuration.

Using a modeling language like PCML seemed adequate for modeling the most important aspects of Linux configuration. The ability to model virtual packages as entities on their own with relationships to one another can be considered a good additional feature, as long as we have correctly understood the concept. If, however, the virtual package concept has been developed for describing the functionality offered by different packages, modeling them as features or functions [5] could be more useful. Those concepts correspond semantically better to what seems to be modeled with virtual packages and provide more flexibility in defining relationships between virtual and concrete packages. For now, as each virtual package is implemented by one package only (although there are several alternatives), it can be stated that the use of abstract component types and subtyping is a reasonable choice. This should, however, be studied further.

On the basis of this case study, there is no indication that the differences between the traditional industry and software domains, referred to by Brooks [6], are of big relevance with respect to product configuration. Invisibility, conformity and complexity did not have an impact in the case product. On the basis of this single case study, it seems that changeability may be more important and a difference between the domains. A single software component can easily have quite a large number of versions. However, Linux is somewhat a special case in this respect as more functional versions are released for users than in a typical commercial software product. Therefore, it should be studied further if the number of versions really is big for software products in general and what sort of challenges this may present for the modeling task. In our case, there were no complex and large version spaces, which means that no conclusions could be done of the issue.

Modeling conflicts was easier using PCML. For example, the modeling of conflicts was simplified. The model then became more manageable as the problems presented in section 3.2 ceased to exist.

The usefulness of cardinality of packages in software configuration was questioned when the mapping of the concepts was made. In this case, there were only optional and required packages and therefore a concept of optionality could be used instead of cardinalities. On the other hand, modeling distributed systems may require a more elaborate cardinality, as a component type may be instantiated (i.e., installed) on several different devices. Thus, we cannot conclude that cardinality would be useless for software configuration modeling.

In the case of a handheld device, a concept for modeling the configuration size would have been useful. In PCML, Such a concept is not present and some problems can be expected when the disk of the device becomes full. The size information was available for all the packages but there were no means used to calculate the configuration size in this case study. However, the resource balancing based approach to modeling incorporated into the ontology of Soininen et al [5] could be useful to capture this phenomenon.

5.1 Problems and Challenges Raised in Our Mapping

We faced a few challenges when modeling Linux Familiar package descriptions with PCML. We next discuss the challenges with the modeling and the challenges with the input data in more detail.

Dangling references and reconfiguration. In the package descriptions of Linux Familiar, there were a remarkable number of references to packages, which were not present in the package list. PCML does no allow references to non-existent parts, and thus, such constraints were simply removed. This did not produce a problem for the configuration task, as the relationships were mainly conflicts and the conflicting, lacking package was not available to be installed to the device. However, this becomes a problem for the reconfiguration task. The package list of Familiar distribution is on the Internet and it only lists the packages currently available. Packages that are no longer available are removed from the list. When reconfiguring the system, the new package to be installed may have a conflict with a package already installed on the device but no longer listed in the model. In this case, the conflict would be ignored according to our model and it would be possible to install an invalid configuration. This is an issue that needs further studies.

Feature modeling. There were some packages included in Familiar, whose names started with "task-". These packages consisted of a set of different packages to make some basic installations easier for the end user. For example, package *task-x* includes all the packages that together implement the Linux graphical user interface called X. They should be studied further, as it seems that they provide a form of feature or function modeling.

Installation order. There were no means to model this information using PCML. When modeling the Familiar package listing with PCML, the installation order part of the information of disappeared. It can be argued that the installation process is separate from the configuration process and that the information should not be in the model in the first place. However, in the case of software, it can be claimed that these processes are commonly intertwined and the separation should not be done. In addition, modeling the informa-

tion on installation order seems to be closely connected to reconfiguration, as it is essential to capture the configurations and the transitions from one configuration to another. This topic requires further work.

Configuration optimization. In section 3.2, the package relationships *recommends* and *suggests* were briefly mentioned. These can be seen as related to configuration optimization. One solution [12] would be to install the packages recommended and suggested every time a package recommending them is installed. This strategy of maximizing the configuration is not the best solution when configuring software for handheld devices. The limited size of the device does not encourage installing all recommended and suggested packages automatically without consulting the user. On the other hand, simply ignoring these relationships and thus minimizing the configuration, as was done in our approach, leaves this information totally unused. It should be studied further in which way the information should be used appropriately.

Replacement, reconfiguration. We also left out the concept of replacement from our model. The input data varied so wildly with regard to this respect and we could not reliably conclude what meaning the replaces relationship should be given. The main interpretations is, as explained in section 3.2, that the replacing package provides the functionality of the mentioned package but also overwrites it at least partially. This operation then may break existing dependencies from other packages. In addition, if the user wants to install the replaced package back on the system, again rewriting some files, she may break the whole system as the replaces relationship is only defined in one direction. In this case study, we did not research this problem further and we see this as a reconfiguration problem to be studied.

Implication and reconfiguration. Using classical implication to model the depends relationship does not seem to correspond to the way dpkg operates. When user removes a package from the system, dpkg checks that the removal does not break any existing dependencies. However, the packages installed initially just to satisfy the constraints of this package, normally due to depends- relationships, remain on the system. After a while, it is possible that the system consists of a large amount of packages, which have no justification for their existence. In a system with limited size, they easily waste resources. In order to capture this justification aspect, a new connective is needed in the constraint language that has similar properties as the rules in the weight constraint language used for formalizing configuration knowledge in [14].

Input version numbers. The version numbering of Familiar is quite versatile and due to the many different conventions in use, it is possible that in some cases a simple string comparison of version numbers may produce false results. This is, however, a problem that can only be solved by changing the version numbering conventions and cannot be simply solved by a modeling language.

6 CONCLUSIONS

We have presented a case study of modeling a configurable software product family, Linux Familiar, with a configuration modeling language designed for representing the structure of a physical product. Findings from this case study suggest that configuration modeling of a software product can be carried out to large extent using such a language. Thus it can be used as a basis for modeling and configuring software product families without a need to develop a radically new language for this purpose. However, there remain some important areas where further research is needed. In the case product, some phenomena strongly suggested that modeling them as functions or features, resources and optimality criteria, familiar from the configuration domain, would increase the understandability and usefulness of the models. In addition, there is some evidence that deeper models of versions of components and reconfiguration knowledge, not usually covered by configuration models, should be supported. In addition to researching these modeling questions, the case study should be continued by completing the model and by empirically testing its validity and the efficiency of the configurator support. In order to investigate whether the findings of this case study can be generalized, more software products from different application domains should also be investigated.

ACKNOWLEDGEMENTS

We gratefully acknowledge the financial support of Technology Development Centre of Finland and Academy of Finland (grant number 51394). In addition, we thank Juha Tiihonen and Andreas Anderson for providing the configurator used in this research and for their help in using it.

REFERENCES

[1] J.Bosch, *Design and use of software architectures - adopting and evolving a product-line approsch,* Addison-Wesley, 2000.

[2] T.Männistö, T.Soininen, and R.Sulonen, 'Modelling configurable products and software product families', *in: IJCAI'01 Workshop on configuration,* 2001.

[3] T.Syrjänen, A rule-based formal model for software configuration. Master's thesis.(2000). Helsinki University of Technology:

[4] D.Sabin and R.Weigel, 'Product configuration Frameworks—A survey', *IEEE intelligent systems & their applications,* **13**, 42-49, (1998).

[5] T.Soininen, J.Tiihonen, T.Männistö, and R.Sulonen, 'Towards a General Ontology of Configuration', *AI EDAM,* **12**, 357-372, (1998).

[6] F.P.Brooks, *No silver bullet -- Essence and accident in software development,* IFIP, 1986.

[7] J.Estublier, 'Software configuration management: A roadmap', *in: Proceedings of 22nd International Conference on Software Engineering (ICSE00), The future of software engineering,* ACM Press, 2000.

[8] T.Syrjänen, 'Version spaces and rule-based configuration management', *in: IJCAI'01 Workshop on configuration,* 2001.

[9] I.T.Bowman, R.C.Holt, and N.V.Brewster, 'Linux as a case study: Its extracted software architecture', *in: Proceedings of ICSE'99,* 1999.

[10] http://www.debian.org/doc/packaging-manuals/virtual-package-names-list.txt

[11] J.Tiihonen, T. Soininen, I. Niemelä and R. Sulonen, 'Empirical testing of a weight constraint rule based configurator' *In Proceedings of the ECAI Workshop W02 on Configuration,* 2002

[12] T. Syrjänen: 'Optimizing Configurations' *In Proceedings of the ECAI Workshop W02 on Configuration,* 2000

[13] http://handhelds.org/familiar/

[14] T. Soininen, I. Niemelä, J. Tiihonen and R. Sulonen. 'Representing Configuration Knowledge With Weight Constraint Rules'. *In Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, 2001.

[15] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, 'The Koala Component Model for Consumer Electronics Software', IEEE Computer, 33, 78-85, 2000

[16] K. Czarnecki and U. W. Eisenecker, Generative Programming, Addison-Wesley, 2000

[17] N.Medvidovic and R.N.Taylor. 'A classification and comparison framework for software architecture description languages', *IEEE Transactions on software engineering,* **26**, 70-93, 2000