Towards Intelligent Support for Managing Evolution of Configurable Software Product Families

Tero Kojo, Tomi Männistö, and Timo Soininen

Software Business and Engineering Institute (SoberIT)
Helsinki University of Technology
P.O. Box 9600, FIN-02015 HUT, Finland
{Tero.Kojo, Tomi.Mannisto, Timo.Soininen}@hut.fi

Abstract. Software product families are a means for increasing the efficiency of software development. We propose a conceptualisation for modelling the evolution and variability of configurable software product families. We describe a first prototype of an intelligent tool that allows modelling a software product family on the basis of the conceptualisation and supports the user in interactively producing correct configurations with respect to the model. The implementation is based on an existing general purpose configurator and thus is not application domain specific. We use the Debian Familiar Linux package configuration task over many releases and package versions as an example. Preliminary results show that the conceptualisation can be used to model evolution of such a software product family relatively easily and the implementation performs acceptably.

1 Introduction

Software product families (SPF) (or lines, as they are also known) have been proposed as a means for increasing the efficiency of software development and to control complexity and variability of products [1, 2]. A SPF can be defined to consist of a common architecture, a set of reusable assets used in systematically producing, i.e. *deploying*, products, and the set of products thus produced.

Software product families are subject to evolution, similarly as other software [3]. This leads to a need for practical solutions for controlling evolution. Configuration management tools keep evolving software products under control during their development [4]. However, the configuration management of large and complex software and the post installation evolution of software products present challenges [5, 6].

In this paper we propose an approach to the modelling of evolving software product families based on viewing them as *configurable software product families*. A *configurable product* is such that each product individual is adapted to the requirements of a particular customer order on the basis of a predefined *configuration model* [7]. Such a model explicitly and declaratively describes the

set of legal product individuals by defining the components out of which an individual can be constructed and the dependencies of components to each other. A specification of a product individual, i.e., a *configuration*, is produced based on the configuration model and particular customer requirements in a *configuration task*. Efficient knowledge based systems for configuration tasks, *product configurators*, have recently become an important application of artificial intelligence techniques for companies selling products adapted to customer needs [8, 9]. They are based on declarative, unambiguous knowledge representation methods and sound inference algorithms. A configuration model traditionally captures the versioning of an SPF in space, i.e., it describes all the different configurations the SPF consists of. Evolution, i.e., versioning in time, is typically not supported.

The main contribution of this paper is a conceptual foundation for modelling evolution of configurable SPFs with the main concern being the deployment phase and generation of valid configurations. The conceptual foundation is based on a subset of a de-facto standard ontology of product configuration knowledge [10] and extended with concepts for modelling evolution.

We describe a prototype implementation of an intelligent tool for modelling configurable SPFs on the basis of the conceptualisation and supports the user in interactively producing a correct configuration with respect to the model. In this we use a state-of-the-art prototype product configurator [11] as an implementation platform. Our implementation is not product or application domain specific. It is enough to change the model to use it for another SPF.

To show the feasibility of both the modelling method and its implementation, we use the Debian Familiar Linux package configuration as an example. It is an appropriate software product family for the purposes of this work as it has a component structure consisting of hundreds of packages, resulting to well over 2^{100} potential configurations. Furthermore, there are multiple subsequent releases of the same package.

The remainder of this paper is structured as follows. Section 2 describes Debian Familiar Linux, its package model and how evolution is currently handled. Section 3 defines the conceptual foundation for representing the evolution of configurable SPFs. In Section 4 the implementation is described and some preliminary results on its feasibility are given. After that the modelling method and implementation are discussed and compared to related work in Section 5. Finally, some conclusions and topics for further research are presented in Section 6.

2 Debian Familiar Linux Case

Debian Linux Familiar is an open source operating system distribution developed for the Hewlett-Packard iPAQ handheld computer. Familiar is distributed as packages, each of which provides a piece of software, such as an application or device driver. A Familiar release is a collection of packages, which make up a complete Linux environment for the iPAQ. Therefore a release allways includes a Linux kernel, device drivers and essential user software such as a shell and editor. The packages of a Familiar release are described in a package description file

distributed with the release. Each release has a release date, on which that release becomes the official current Familiar distribution. The releases are stored in separate folders on the ftp.handhelds.org FTP server. Figure 1 shows an example of a single package description.

```
Package: ash
Essential: yes
Priority: required
Section: shells
Installed-Size: 152
Maintainer: Carl Worth <cworth@handhelds.org>
Architecture: arm
Version: 0.3.7-16-fam1
Pre-Depends: libc6 (>= 2.2.1-2)
Filename: ./ash_0.3.7-16-fam1_arm.ipk
Description: NetBSD /bin/sh
```

Fig. 1. Example Debian Familiar Linux package description

The package description provides information on the different properties of the package it represents. The description provides the package name, essentiality, version information and dependencies. The version information contains the revision information and the status of the package. If a revision number is followed by "a" or "alpha" or "pre", the package is an alpha or pre-release version. The possible dependencies of a package are `Depends, Pre-depends, Conflicts, Provides, Replaces, Recommends` and `Suggests`. The meanings of these are:

- Depends - the package requires another package to be installed to function correctly.
- Pre-depends - installation of the package requires another package to be installed before itself.
- Conflicts - the package should not be present in the same configuration some other package.
- Provides - the package provides the functionality of some other package.
- Replaces - the installation of the package removes or overwrites files of another package.
- Recommends - another package is recommended when it is presumable that the users would like to have it in the configuration to with the package.
- Suggests - another package is suggested to get better use of the package.

Of these dependencies Depends, Pre-depends and Conflicts directly indicate the need for an another package or conflict with another package. Provides, Recommends and Suggests can be seen as a method of providing help at installation time in the form of features. Replaces is directly associated with the installation

of the package. The dependencies are used by the Familiar package management tool when installing a package. Most packages have a different maintainer, as the packages are maintained by volunteer hackers. This brings inconsistencies, for example, in the usage of dependencies and to the version naming scheme.

The Linux Familiar distribution has had four major releases, 0.3, 0.4, 0.5 and 0.6, and several minor releases between the major releases during the two years the project has been active. The releases 0.4, 0.5, 0.5.1 and 0.5.2 were chosen for this study. Release 0.6 was not included as it has just recently come out.

The total number of packages in the four releases is 1088, making the average number of packages per release 272. Of these packages 148 have their priority set as required. The number of dependencies between packages is 1221.

Installation instructions for each release of Familiar can be found on the Familiar WWW pages [12]. Each release has separate installation instructions, even though the process is similar for each release. This is a configuration problem related to the versioning in space. Problems related to evolution are the focus of this paper and include the representation of configuration knowledge over time and (re)configuration tasks that span time. For example, updating from an older release to a newer one requires that the user installs everything from scratch onto her/his HP iPAQ, which wipes the handheld clean removing all the user data. Installing packages from multiple releases is not supported. This means that once a user has installed a certain release on her/his iPAQ she/he must use packages only from that particular release. There clearly is need for a package management utility that supports (re)configuration over several releases. To achieve this one needs to incorporate the versioning in space and versioning in time. Constructing such a solution is the topic of this paper.

3 A conceptualisation for Modelling Evolution

This section describes a conceptualisation for evolution of configurable software product families. This work is based on a de-facto configuration ontology [10], which does not contain concepts for evolution. The conceptualisation uses concepts, such as components, their properties, compositional structure and constraints, which are introduced in the following and illustrated in Figure 2. We first introduce the main concepts for modelling configurable SPFs, i.e., for capturing versioning in space, and thereafter add the concepts for modelling evolution—more detailed discussion is postponed to Section 5.

A configurable SPF fundamentally consists of a large set of potential product individuals, called *configurations*. In the conceptualisation, a configuration is represented by *component individuals*, their *properties* and *has-part* relations between component individuals. A *configuration model* is defined to describe which configurations are legal members of a configurable SPF. A configuration is related to a configuration model by a *is-configuration-of* relation.

A configuration model contains *component types*, their *part defintions* and *property definitions* and *constraints*. Component types define the characteristics (such as parts) of component individuals that can appear in a configuration. A

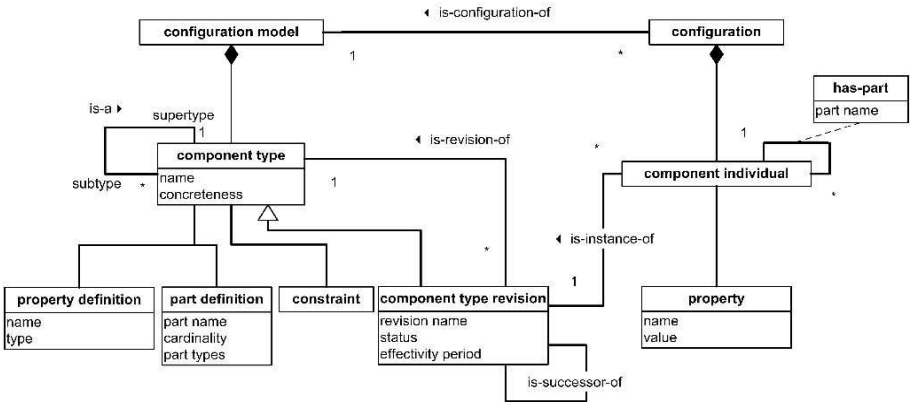

Fig. 2. Meta-model of concepts for modelling evolution of configurable SPFs

component type is either *abstract* or *concrete* (represented by the *concreteness* of the component type). 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 non-empty set of possible *part types* and a *cardinality*. A component type may also define properties that parameterise or otherwise characterise the type.

Component types are organised in a taxonomy or class hierarchy by means of a *is-a* relation where a *subtype* inherits the property and part definitions of its *supertypes* in the usual manner. Multiple inheritance is not supported.

These concepts provide the basic variation mechanisms, i.e., means for capturing the versioning in space:

1. Alternative parts are modelled by *part type* of part definitions, which defines the component types whose individuals are allowed as parts in a has-part relation with a particular (part) *name*.
2. A *cardinality* defines a range for the allowed number of component individuals for a has-part relation. Zero minimal cardinality represents an optional part and minimal cardinality of 1 a mandatory part.
3. A property definition with an enumeration type defines a set of variation choices, e.g., “UNIX”, “win2000”, “MacOs”.
4. Subtyping of component types can also be used for representing variation; the subtypes of a component type being the variants of the component type.

Constraints associated with component types define additional conditions that a *correct configuration* must satisfy. A constraint is a boolean expression with references to access the parts and properties of component individuals.

To model the evolution in configurable SPFs, additional concepts are introduced, namely *component type revision*, *status*, and *effectivity period*.

A component type has a set of revisions, called component type revisions, which are related to the component type by *is-revision-of* relation, and ordered by *is-successor-of* relation. These relations provide a simplified conceptualisation for revisioning and elaboration on them goes beyond the scope of this paper. Revisions capture the evolution of a component type in time. However, separating the versioning in time from versioning in space in this manner is a simplification we make in this paper. Ultimately, such versioning dimensions of a component type should be represented uniformly [13].

Status tells the life-cycle status of a component type revision [14]. The status is a measure of the maturity of a component type revision, e.g., "unstable", "stable" and "end of life", and can be used as additional information in configuration task. The status is a useful concept, e.g., for expressing the user requirements, but has no relevance in determining the correctness of a configuration.

Effectivity period is a time interval stating when an component type revision may legally appear in a configuration. Effectivity period is thus a new additional concept needed in determining the correctness of a configuration.

In the meta-model, component type revision is a subtype of component type to indicate that component type revisions have the same properties as component types plus the additional concepts for representing evolution. Each component individual is directly an instance of a component type revision, represented by *is-instance-of* relation. This basically means that component individuals are component type instances with additional revision information.

The is-a relation between component types is constrained more than what is visible in Figure 2. The relation is only allowed between component types, not component type revisions. Similarly, the is-revision-of relation can only be from a component type revision to component type.

4 Implementation

This section describes a prototype implementation of the modelling method and the intelligent support system, presented in Figure 3, for managing and configuring SPFs based on the models. We first describe an existing configurator prototype, called WeCoTin[1], that is used as the implementation platform. We then show how the Debian Familiar Linux package descriptions are modelled using the conceptualisation presented in the previous section. After this we show how a model based on the conceptualisation is represented using the modelling language of the prototype configurator. Finally the functionality of the modelling and support tool is presented, and some preliminary results on the feasibility of the method and implementation are provided.

The implementation consists of two new pieces of software, software that mapped the individual Familiar package description files to the conceptualisation, software that mapped the conceptualisation to a product configuration modelling language (PCML) and the existing WeCoTin configurator.

[1] Acronym from Web Configuration Technology.

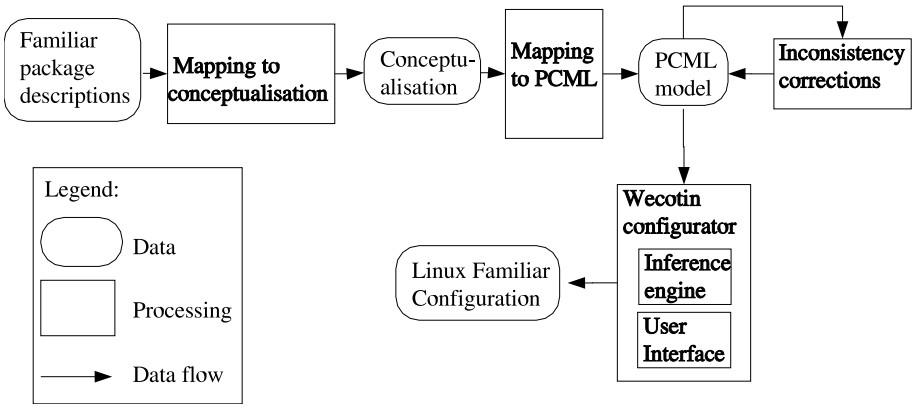

Fig. 3. The architecture of the implementation

4.1 Existing Work

An existing product configurator, WeCoTin, which is currently under development at Helsinki University of Technology, is used as the implementation platform for the conceptualisation presented in the previous section [11].

WeCoTin enables configuration over WWW and centralised configuration model management. The implementation semi-automatically generates a web-based user-interface. WeCoTin supports the user by preventing combinations of incompatible components or their versions, by making sure that all the necessary components are included, and by deducing the consequences of all selections. WeCoTin is also capable of automatically generating an entire correct configuration based on requirements. A state-of-the-art logic-based artificial intelligence knowledge representation and reasoning language and a system implementing it provide the inference mechanism for WeCoTin [15]. WeCoTin translates the configuration model presented in PCML to weight constraint rules and uses a state-of-the-art general implementation of such rules, Smodels [16], for efficiently computing configurations satisfying given requirements. PCML is based on a practically important subset of a de-facto standard ontology of configuration knowledge [10] that unifies most of the existing approaches to configuration modelling.

4.2 Mapping the Evolution of the Familiar Linux Releases

The package descriptions from the different releases of Familiar were mapped to the conceptualisation as follows (Table 1).

A root component type, the root of all configuration models was defined.

For each package a component type was defined, and made a part of the root component type by a part definition. If the package is essential the cardinality of of the part definition corresponding to it is 1, otherwise it is 0..1. The part

Table 1. The mapping of Familiar package descriptions to the conceptualisation

Familiar term	**Conceptualisation**
Package	Component type Part definition in the root component type
Name of the package	Component type name
Essentiality	Part definition cardinality
Version	Component type revision Component type revision status
Release date information	Component type revision effectivity period
Depends	Constraint
Pre-depends	Constraint
Conflicts	Constraint
Provides	Installation time activity (Not mapped)
Replaces	Installation time activity (Not mapped)
Recommends	Semantics unclear and installation time activity (Not mapped)
Suggests	Semantics unclear and installation time activity (Not mapped)

types of the part definition containt the component type that was mapped. The name of the component type and the name of the package were the same.

A component type has the revision of the package it represents as a property. The version identifier was taken directly from the package information. If the package had multiple versions, the is-successor-of relation was defined by the chronological order of the package versions.

Status information of a package is a property of the component type revision. The status information was derived from the package version information, if a version was marked with "a" or "alpha" or "pre", the status property of the component type revision was given the value unstable, otherwise a stable value was given.

The component type revisions have an effectivity period based on the releases the package is in. If a version of a package is present in only one release the component type revision has an effectivity period starting with the release date of that release and ending with the release date of the next release. If a version of a package is present in consecutive releases the effectivity period of the component type revision is a combination of the effectivity periods as if the package version were present in all the releases separately.

The package dependencies were mapped to constraints. Depends and pre-depends were mapped both to a requirement constraint that states, that the package requires another package. The additional information provided by pre-depends is used only in the instantiation phase, which was outside the scope of this case. Conflicts was mapped to a constraint that states, that the two packages may not exist together in the same configuration. Provides, replaces, recommends and suggests were not mapped, as they are seen as a method of

providing help at installation time and thereby fall outside the scope of this work.

4.3 Mapping the Conceptualisation to PCML

The conceptualisation needed to be mapped to PCML so that WeCoTin could be used to generate configurations of the Familiar system (Table 2).

Table 2. The mapping of the conceptualisation to PCML

Conceptualisation	**PCML mapping**
Component type	Component type
Property definition	Property definition
Part definition	Part definition
Constraints	Constraints
Component type revision	Property of component type as a temporally ordered list of strings
Component type revision status	Property of component type as a temporally ordered list of strings
Component type revision effectivity period	Properties of component type as integers stating end and start time of effectivity period Constraints expressing which version is effective at which part of the effectivity period

PCML provides modelling concepts for component types, their compositional structure, properties of components, and constraints. The mapping from the evolution concepts to PCML concepts was simple for these concepts. PCML also provides value types and structures such as lists, strings and integers, which were used in the mapping of the other evolution terms [11].

Component type revisions were mapped to a property of component type as a temporally ordered list of strings, i.e., ordered by the is-successor-of relation. Strings were used as the version identifiers in the Familiar package descriptions contain alphanumerical characters.

Component type revision status was mapped to a property of component type as a list of strings that was ordered in the same way as the list of revisions.

Component type revision effectivity periods were combined and mapped as two integers stating the start and end of the component type effectivity period. Constraints were used to specify which component type revision can be used in a configuration at specific parts of the effectivity period. This was used in cases where a package was present in many Familiar releases and it was necessary to identify when each revision can be used in a configuration.

Due to the fact that the inference engine in WeCoTin does not at this time support reasoning over large integer domains well, time used for effectivity peri-

ods was discretised. The times selected were those at which the Familiar releases were made and single points of time in between the release times.

4.4 Implementation of the Conceptualisation

The software for mapping the Familiar package descriptions reads the package description file for a single Familiar release and maps the package descriptions to the conceptualisation as described in Section 4.2. The software performs the mapping for each Familiar release. The software was implemented with perl and it's output is files containing the evolution models of each Familiar release that was mapped.

The software for mapping the conceptualisation to PCML reads the evolution models created by the component for mapping the Familiar package descriptions, combines them and simultaneously maps them to a PCML model as described in Section 4.3. The software searches the evolution model files for component types and collects the different component type revisions under a single component type. The constraints expressing which revision is effective at which part of the effectivity period are created as the component type revisions are collected. Identical part definitions and constraints are removed. The output from the software is a PCML model, which can be given as input to WeCoTin. The software was implemented with perl.

Figure 4 shows the final PCML presentation of the previously shown Familiar package description (Figure 1 in Section 2).

The PCML model was then input into WeCoTin which translated it into weight constraint rules. Internal inconsistencies in the model were identified and removed. The inconsistencies come from missing or erroneous package definitions, such as requirements on packages that are not in the model and inconsistent requirements, where a package at the same time depends on another package and conflicts with it. Requirements on packages that are not in the release are an implication that the package has been ported from somewhere else, like the main Debian Linux PC distribution, and the maintainer has not updated the package dependencies. After the inconsistencies were removed the PCML model was ready for use. The final PCML model file was an ASCII text file of 781 kiloBytes.

WeCoTin can semi-automatically generate a web-based user-interface for the end user based on the PCML model, presented in Figure 5. WeCoTin provides the possibility of defining default property sets, which in this implementation were used to set the default component type revisions according to the current time set by the user. WeCoTin supports the user by preventing combinations of incompatible component types or their versions, by making sure that all the necessary component types are included, and by deducing the consequences of already made selections. The implementation is also capable of automatically generating entire correct and complete configurations from the model based on user requirements. The inference engine makes inferences on the basis of the model in a sound and complete manner, meaning that e.g. the order in which

```
component type ash
subtype of concrete
property revision value type string constrained by $ in
list("0.3.7-16-fam1","0.3.7-16")
property status value type string constrained by $ in
list("stable","unstable","eol")
property refinement effectivity_start value type integer always 20010606
property refinement effectivity_end value type integer always 20020621

part ashpart
allowed types ash
cardinality 0 to 1

constraint ash_0_3_7_16_fam1_time
ashpart.ash:revision = "0.3.7-16-fam1" implies current_time > 20020514
and current_time < 20020621

constraint ash_0_3_7_16_time
ashpart.ash:revision = "0.3.7-16" implies current_time > 20010606
and current_time < 20020621

constraint ash_DEP_libc6_2_2_1_2
present(ashpart) implies present (libc6part) and
libc6part.libc6:revision >= "2.2.1-2"
```

Fig. 4. Example Debian Familiar Linux package description

the package descriptions appear in a model do not affect the set of correct configurations. The inference engine does not need to be changed if the model changes.

It took about one minute to map the individual Familiar package description files to the conceptualisation and a similar amount of time to map the conceptualisation to PCML. The translation of the configuration model by WeCoTin took 20 minutes. The removal of inconsistencies took about an hour of work, but had to be performed only once to the package descriptions. After that the model can be used to generate a correct Familiar configuration in seconds. No programming is necessary at any step of the configuration process. The configurations generated by WeCoTin are according to our initial experiences correct with respect to the package descriptions used in configuration and based on manual inspection could be loaded into an iPAQ to provide a bootable working Familiar installation.

5 Discussion and Related Work

5.1 Modelling Method

The concepts provided by PCML for configuration were used in the conceptualisation presented in this paper. In addition to these concepts PCML offers other

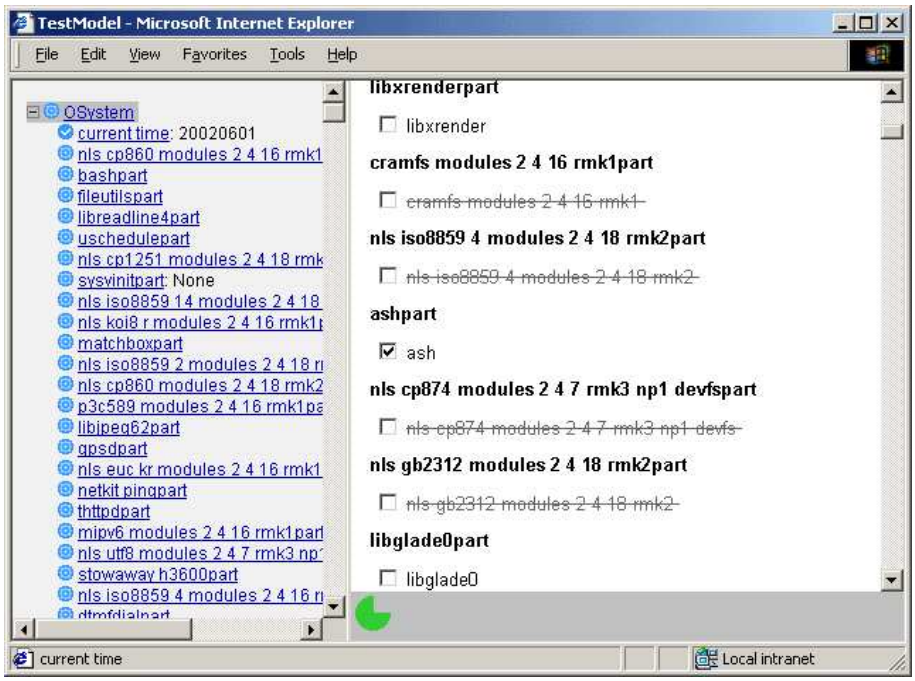

Fig. 5. WeCoTin user-interface

modelling primitives, that are not discussed here. We chose to use PCML since it provides a means for modelling the component types and their dependencies and there is a prototype configurator supporting PCML on top of which we could build the implementation of the modelling method.

The method presented in this paper was sufficient for modelling the evolution and structure of Linux Familiar package descriptions over multiple releases. However the modelling method has limitations. The separation of the versioning in time from versioning in space performed in the conceptualisation is a simplification. Ultimately, such versioning dimensions of a component type should be represented uniformly [13, **?**]. However for the prototype implementation the limited version concept was sufficient. The modelling method supports variation, e.g., through the use of alternative and optional parts of a component type. This is not a optimal method for representing component type variants, but more a way of presenting the variation in the product family structure. The relationships between versions should perhaps be modelled with a richer set of relations.

In the mapping of the Familiar package descriptions to PCML some compromises were made. Versions of a package were mapped as properties of the component type. Constraints did not have explicit effectivity periods, but were effective throughout the model, when the effectivity should be the same as the component type revisions' that they refer to. The global effectivity period of con-

straints does not impact the configuration results, as they only have meaning if the component type revisions they refer to are effective.

As the modelling method was tested on one product family, Linux Familiar, the question of whether the modelling method is suitable to other products is open. More example implementations are needed to verify that the model concepts are suitable and sufficient to model evolution in different types of SPFs.

Conradi and Westfechtel [5] present an overview of existing approaches in software configuration management for building consistent configurations of large software products. They identify version models and the constraints for combining versions as the most important parts in configuring large software products. Managing the complexity of the different versions of objects under SCM control and the close ties of the configuration rule base with the version database are seen as problems. The use of constraints for and versioning of the rule base are seen as solutions to these problems. The approach described in this paper relies on an object-oriented configuration modelling method that allows representing constraints between components types to partially solve these problems.

The use of attribute value pairs [14, 6] or attribution schemes in the selection of a configuration are used in many SCM systems [18]. By selecting attribute sets components are included in the configuration. Feature logic can be used to verify the correctness of a configuration based on the feature terms or attributes [18]. The model presented in this paper contains the concept of properties for component types, which can be used like attributes. The constraints presented in this work can be used in selecting the component types as in SCM systems using an attribution scheme or feature logic. In addition this work also provides formal semantics for the model giving a more powerfull language which can be used to generate a correct configuration.

Configuration management techniques are applied for presenting the variability, optionality and evolution of software architecture in [19]. The tool Menage can be used to graphically specify software architectures. The work of Menage has been continued in Mae [20], which added an environment for architecture based analysis and development. Their works bridges the gap between the areas of software architecture and configuration management by introducing a system model combining architectural and configuration management concepts in a way which is similar to the model presented here, with the exception that our model does not include concepts for modelling connections of components types. These were not required for Linux configuration but may well be appropriate for other SPF configuration problems.

Software release management is the process of making a working piece of software available to users [21]. Software release management provides the notion of dependency between components. The tool SRM (Software Release Manager) [21] supports the release management process. It provides developers with a tool to track and control dependencies between components and users with a single source of software. The approach described in this paper differs from SRM in that it uses formal semantics in describing the configuration model and relies on artificial intelligence methods to generate a correct configuration. However SRM

provides installation help and hides the physical distribution of of software from users.

The Koala component model provides a method for describing the product configuration through components and interfaces [22]. The Koala model does not provide a method for modelling evolution in itself, but a separate process which handles evolution is needed. The process includes rules for making changes in the interfaces and components. The approach in this paper provides support for modelling evolution on the concept level rather than on the process level, giving more freedom in choosing a development process.

5.2 Implementation

Based on the modelling method a prototype implementation of the evolution of Linux Familiar package descriptions was presented. The implementation showed that it is possible to model the evolution of a large software product using product configuration techniques. The model made from four releases of Familiar Linux can be used to configure a valid Linux environment for the HP iPAQ. Even reconfiguration of the system over time is possible. The model provides a way for creating a configuration containing any packages which have an effectivity at the current time. This overcomes the problem of needing to make a clean install when the user wishes to have packages from different Familiar releases on her/his HP iPAQ. The model supports having packages from multiple releases in a single configuration and the system checks that the generated configuration is correct based on the declarative semantics of the model. These are clear benefits compared to the current release system of Familiar Linux.

WeCoTin also support structuring the Familiar packages into categories and provides an intuitive web interface, with intelligent support for the user, such as graying out packages that may not be selected. None of these are possible with the current package management tools of Familiar. However this is only a prototype implementation and not yet capable of replacing the current package management tools of Familiar. For example the installation tools of Familiar have not been integrated to the current implementation.

Configuration problems are typically at least NP-complete [11]. However, it is not clear that the worst-case exponential computation for generating a configuration occurs in practise, since many physical products seem to be loosely constrained and can be configured efficiently [11]. This same potential for expensive computation holds for WeCoTin [15] and the implementation presented in this paper. This should be studied further by extensively testing the current implementation as well as with other software product families.

The Debian Linux PC distribution has been successfully modelled with a similar approach as presented in this paper [23]. Also a release of Familiar has been modelled as a configurable software product with PCML [24]. The two works proved that the modelling of software products with configration modelling languages is feasible. This paper brings the addition of evolution concepts to the configuration of SPFs.

6 Conclusions and Further Work

In this paper we proposed an approach to modelling the evolution and variability of software product families based on viewing them as configurable products. We presented a conceptualisation for modelling evolution of such product families that is based on a subset of a de-facto standard ontology of product configuration. This subset was extended with concepts for modelling evolution: revision, effectivity and status. With these concepts the compositional structure and evolution of a software product family can be modeled. We further described a first prototype implementation of an intelligent tool that allows modelling a configurable software product family on the basis of the conceptualisation, and, supports the user in interactively producing a correct configuration with respect to the model. The implementation is based on an existing general purpose configurator prototype and thus it is not product or application domain specific.

To show the feasibility of both the modelling method and its implementation, we use a Debian Familiar Linux package configuration task over many releases as an example. Preliminary results from the implementation show that the modelling language can be used to model evolution of such a product family relatively easily and that the implementation performs acceptably. However, several topics for further research remain. It is not clear that the modelling method is applicable for all software product families. In particular, the concepts for modelling evolution were relatively simple and may need to be strengthened to incorporate more complex semantics for example generic objects and the evolution of the entire model. Furthermore, it may be that some dependencies between components of a product family were easier to model if concepts for modelling interfaces and connections between components were included. In addition, as the configuration problem solving can be computationally very expensive, more thorough testing of the efficiency of the implementation should be carried out. These issues should be investigated by more empirical research into modelling different types of software product families and testing the implementation more thoroughly on those and also on the Linux distribution discussed in this paper. This would probably require and result in further developments to the conceptual foundation and improving the usability and efficiency of the prototype implementation.

Acknowledgements

We gratefully acknowledge the financial support of Technology Development Centre of Finland. We also thank Andreas Anderson and Juha Tiihonen for providing the configurator used in this research and for their help in using it.

References

1. Bosch, J., Evolution and Composition of Reusable Assets in Product Line Architectures: a Case Study, Proc. 1.st Working IFIP Conf. on SW Architecture, (1999)

2. Clements, P., Northrop, L., Software Product Lines: Practices and Patterns, Addison-Wesley, (2001)
3. Svahnberg, M., Bosch, J., Evolution in Software Product Lines: Two Cases, Journal of Software Maintenance - Research and Practise 11(6), (1999) 391–422
4. Estublier, J., Software Configuration Management: A Roadmap, ICSE - Future of SE Track, Ireland,(2000) 279–289
5. Conradi, R., Westfechtel, B., Configuring Versioned Software Products, in: ICSE'96, Proc., LNCS, Vol. 1167, Springer, (1996) 88–109
6. Belkhatir, N., Cunin, P.Y., Lestideau V., Sali, H., An OO framework for Configuration of Deployable Large Component based Software Products, OOPSLA 2001
7. Soininen, T., An Approach to Knowledge Representation and Reasoning for Product Configuration Tasks, PhD thesis, Acta Polytechnical Scandinavica, No. 111, (2000)
8. Faltings B, Freuder EC, editors., Special Issue on Configuration. IEEE intelligent systems & their applications; 13(4), (1998) 29–85
9. Darr T, McGuinness D, Klein M, editors., Special Issue on Configuration Design. AI EDAM; 12(4), (1998)
10. Soininen, T., Tiihonen, J., Männistö, M., Sulonen, R., Towards a General Ontology of Configuration, AI EDAM 12(4), (1998) 357–372
11. Tiihonen, J., Soininen, T., Niemelä, I., Sulonen, R., Empirical Testing of a Weight Constraint Rule Based Configurator, ECAI 2002 Configuration Workshop, (2002)
12. Hicks, J., Nelson, R., Familiar v0.6 Installation Instructions http://handhelds.org/familiar/releases/v0.6/install/install.html
13. Männistö, T., Soininen, T. and Sulonen, R., Product Configuration View to Software Product Families, SCM-10 held at ICSE 2001, Canada, (2001)
14. Mahler, A., Lampen, A., An integrated toolset for engineering software configurations, SIGPLAN Software Engineering Notes, 13(5), USA, (1988)
15. Soininen, T., Niemelä, I., Tiihonen, J., Sulonen, R., Representing Configuration Knowledge With Weight Constraint Rules, AAAI Spring 2001 Symposium, USA, (2001)
16. Simons, P., Niemelä, I., and Soininen, T., Extending and implementing the stable model semantics, Artificial Intelligence, 138(1-2), (2002) 181-234
17. Männistö, T., A Conceptual modelling Approach to Product Families and their Evolution, PhD thesis, Acta Polytechnical Scandinavica, No. 106, (2000)
18. Zeller, A., Configuration Management with Version Sets, PhD thesis, Technical University of Braunschweig, (1997)
19. van der Hoek, A., Heimbigner, D., Wolf, A.L., Capturing Architectural Configurability: Variants, Options and Evolution, CU-CS-895-99, Univ of Colorado, (1999)
20. van der Hoek, A., Mikic-Rakic, M., Roshandel, R., Medvidovic, N., Taming Architectural Evolution, ESEC/FSE 2001, (2001) 1–10
21. van der Hoek, A., Hall, R., S., Heimbigner, D., Wolf, A., L., Software Release Management, ESEC/FSE 1997, (1997) 159–175
22. van Ommering, R., van der Linden, F., Kramer, J., Magee, J., The Koala Component Model for Consumer Electronics Software, IEEE Computer 33(3), (2000)
23. Syrjänen, T., A rule-based formal model for software configuration, Master's thesis, Helsinki University of Technology, (2000)
24. Ylinen, K., Männistö, T. and Soininen, T., Configuring Software with Traditional Methods - Case Linux Familiar, ECAI 2002 Configuration Workshop, (2002)