The Linux Kernel Configurator as a Feature Modeling Tool

Julio Sincero and Wolfgang Schröder-Preikschat
Department of Computer Science 4
Friedrich-Alexander University Erlangen-Nuremberg
{sincero,wosch}@cs.fau.de

Abstract

In order to contribute to the understanding of how the SPL community and the open source community can benefit from each other, we present the Linux Kernel Configurator (LKC). We describe its capabilities and explain how it can be used for the design of feature models.

1. Introduction

A software product line (SPL) is a set of software components that can be composed in order to deliver a specific product. The scientific community has proposed a variety of techniques to support the development of SPLs, *feature modeling*, *product line scoping*, *feature implementation techniques*, *variability management*, among others. These approaches aim at producing flexible software architectures, reducing time-to-market, enabling substantial code reuse, and, consequently, providing high-quality software products. The literature has plenty of study cases showing the benefits of the adoption of such techniques.

Different sectors of the software industry have been adopting the SPL approach in their development process. However, another increasingly interest is the use of *open source* software. This can be motivated by many reasons, but mainly, due to cost factors and the attested quality of many open source projects. In addition, it has been shown that some open source projects, due to its technical implementation, can be considered to be SPLs [8], even though during its development process no methods proposed by the SPL community were used.

Therefore, it is clear that both communities share interests and goals. We believe that for a better understanding of how these communities could benefit from each other, technical issues should be discussed. This paper presents the *variability management* employed by the Linux Kernel. We present the open source tool that is used for this task, and also, how it can be adapted to be used by the SPL community as a *feature modeling* tool.

Motivation

We are specially interested in addressing the configuration of non-functional properties (NFPs) in SPLs[9, 5]. We believe that information about NFPs should be provided during feature selection so that the application engineer can be aware of the impact of a determined feature on the final product. Therefore, our idea was to extend a feature modeling tool in order to appropriately present this information that cannot be accommodated in the *feature diagram*, as it can be in the form of graphs or charts.

Nevertheless, we were not able to find any open source feature modeling tool to bring our extensions, and also, our goal was not to develop one from scratch. As the Linux Kernel configurator is open source and very flexible, we decided to test if it could meet our needs. It turned out that it could be easily used as a feature modeling tool. In this paper we demonstrate how to design features models with it. [1]

2. The Linux Kernel Configurator

The Linux Kernel Configurator (LKC) is a tool that is delivered within the Linux Kernel in order to enable its configuration (feature selection) . Its first prototype was proposed in 2002, the current version is 1.3, and as the Linux Kernel, it is released under the GNU General Public License.

2.1 A Little bit of History

In 2001 the community around the Linux Kernel started to show dissatisfaction with the kernel configuration tool, back then known as *configuration menu language* (CML1). With the growth of the kernel, the configuration process was getting very complicated. The tool was responsible for selecting the capabilities to be built into the kernel, handling dependencies and providing the user interface for feature selection. Moreover, it was comprised of a mixture of code

[1]The aforementioned extensions regarding NFPs will be subject of another publication.

written in Tcl/Tk scripts, awk scripts, pearl and C, which made it hard to understand and to maintain[1].

In order to solve this problem, a configuration menu language 2 (CML2) was proposed. It was a *mini-language* designed specifically for configuring kernels. A *ruleset* describing all the available options and their dependencies can be translated into a *rulebase* that is read by the front-end in order to configure the kernel[7]. After more then two years of development, several *flame wars* on the mailing list, and many improvements over the previous system, the project was dropped and not accepted in the official kernel tree (this fact shows how restrictive and demanding is the community regarding new code being merged in the official tree). Nevertheless, the source code is still available and it is used as the configuration tool of other projects.

A couple of months after the discussions about the CML2 had finished, the LKC was proposed aiming at addressing the shortcomings of both CLM1 and CML2. According to the author, the major advantages over CML2 are: (a) it is written in C code (CML2 is written in Python which makes a Python interpreter to be delivered with the kernel) (b) a tool for the automatic converting of the CML1 configuration into the new one is included, (c) it is less complex then CML2, it does not try to address problems like facilitating the kernel configuration for non-experts as the CML2 does.

As these three points were of great importance for the linux developers, after around one year of testing and improvements, the LKC was accepted and merged in the kernel `2.5.45`.

2.2 The Linux Kernel Configurator

The LKC is basically comprised of a *parser* and a *dependency checker* that are used as the back-end. To enable the selection of *configuration options* (as defined in a configuration database), different front-ends (graphical, text-mode, command-line interactive, etc.) are provided.

A configuration database is the collection of *configuration options* organized in a tree structure. Every entry has its own dependencies that are used to determine its visibility, any child entry is visible only if its parent entry is also visible. An entry either defines a *configuration option* or is used to organize them[10].

The configuration file is a text file containing the entries which must follow a strict syntax. The configuration database is built as a set of *entries* which define *configuration options*. Line 1 of Listing 1 shows[2] an entry definition, it starts with the keyword `config` and is followed by its name. The next lines of an entry are used to define its attributes, which can be the following:

type define the type of an entry, they can be `boolean`, `tristate`[3], `string`, `hex` and `integer`. An example is shown on line 2 of Listing 1

input prompt is the visual name of the configuration option that is displayed to the user during configuration. On line 1 the actual configuration name is defined as (`GPL`) which will be used in the generated configuration file, however, the user will see during configuration the name `ROOT` as shown on line 2.

default value is assigned to the configuration symbol if no value was set by the user. An example is given on line 17.

dependencies define the requirements of the menu entry. They can simply define the entry depending on a single configuration option, as shown on line 6, or can be in the form of logical expression using primitives like && (logical and), || (logical or), as shown on line 18.

reverse dependencies are used to force the lower limit of the value of another symbol. As shown on line 3, if the symbol `GPL` is selected, the symbol `M1` will automatically be selected as well.

numerical ranges limit the range of possible input values for `integer` and `hex` symbols.

help text defines the *configuration option* help text to be shown during configuration. Examples are shown on line 21 and 31.

Listing 1. LKC language

```
 1  config GPL
 2    boolean "ROOT"
 3    select M1
 4
 5  choice
 6    depends on GPL
 7    prompt "Graph Type"
 8
 9    config DIRECTED
10      boolean "Directed"
11
12    config UNDIRECTED
13      boolean "Undirected"
14  endchoice
15
16  config NUMBER
17    default y if GPL
18    requires (BFS || DFS)
19    boolean "Number"
20    ---help---
21    Assigns a unique number to each
22    vertex as a result of a graph
```

[2]this listing is an excerpt of the GPL [6] feature model designed with the LKC language

[3]the boolean type can be assigned to `yes` or `no`, the tristate type allows an extra value (`m`) which means that the configuration option should be included, however, as a separate module.

```
23      traversal.
24
25  config CC
26      depends on GPL
27      requires (BFS || DFS)
28      requires UNDIRECTED
29      boolean ”Connected Comp.”
30     ---help---
31     Computes the connected components
32       of an undirected graph, which are...
```

A configuration database, which defines the valid configurations that can be derived with the front-ends, is created using the *entries* and the *attributes* as described above.

Moreover, in order to provide a better organization of the entries in the configuration tree that is displayed to the user, the following constructs are allowed:

menu Entries defined between the keywords `menu` and `endmenu` are grouped together and displayed in a separate window. It may also have an attribute `prompt` to name the groups of entries. An example is given between lines 5 and 14.

choice Only one entry of those defined between the keywords `choice` and `endchoice` can be selected if its parent entry is also selected.

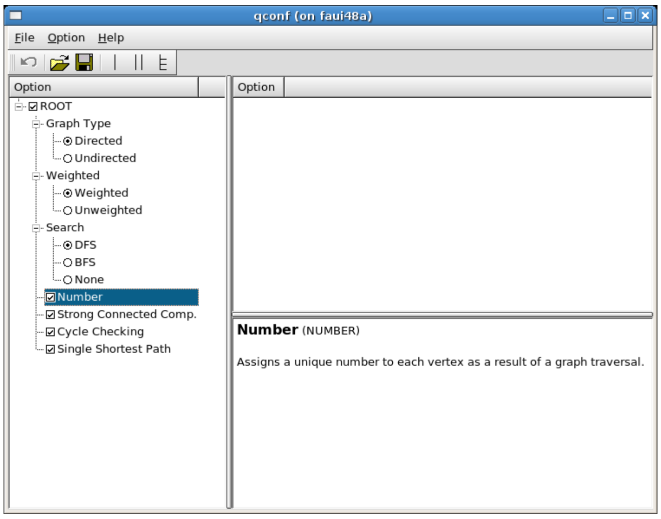
3. Feature Modeling with the LKC

The main contribution of this paper is to show how to design feature models using the LKC *configuration language* described in the previous section. After the introduction of feature models by Kang et. al.[3] many extensions were proposed. In this work we will concentrate on the basic syntax allowing *mandatory features*, *optional features or groups* and *alternative groups*. Regarding extra feature constrains, we allow *implies* and *excludes*. This decision was inspired by the work of Benavides et. al. [2], which describes the mapping from these feature model relations to representations in the form of *constraint satisfaction problem*, *boolean satisfiability problem* and *binary decision diagrams*.

Table 3 summarizes the mappings from the LKC language to feature model relations.

Most of the mappings were relatively easy to perform. For the *mandatory* relation, the parent feature forces the selection of the child by the use of a reverse dependency (`select`). The *optional* relation is described by using a dependency between the child and the parent feature (`depends on`). The *or group* is designed by creating reverse dependencies between the children and the parent, this was done inside a menu definition in order to group the children together. The *alternative group* can be described by including configuration options (the children) inside a

Figure 1. The LKC graphical interface



```
ROOT
  Graph Type
    (•) Directed
    ( ) Undirected
  Weighted
    (•) Weighted
    ( ) Unweighted
  Search
    (•) DFS
    ( ) BFS
    ( ) None
  [✓] Number
  [✓] Strong Connected Comp.
  [✓] Cycle Checking
  [✓] Single Shortest Path
```

Number (NUMBER)

Assigns a unique number to each vertex as a result of a graph traversal.

`choice` definition, which has the same semantic as of *alternative group* in feature models.

Using this mapping we were able to design several feature models. So far we did not find any feature model construction that could no be modeled with the LKC. Figure 3 depicts the screenshot of the LKC graphical front-end displaying the feature model of the Graph Product Line (GPL) [6] which was proposed as a standard problem for evaluating product lines. As we have[4] an implementation of this product line where the features are implemented by means of conditional compilation, the output of the configurator could be used (it is a set of pre-processor `defines`) in the compilation process of the GPL product line.

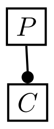
4. Future Work

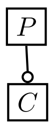
As described previously we aim at extending the LKC front-end to present information about non-functional properties and use it as our feature modeling tool, these extensions are currently being implemented. Moreover, the design of a very simple textual feature modeling language and a tool to transform it in the LKC language format is currently under development.

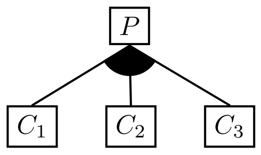
5. Conclusion

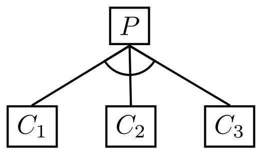
In order to contribute to the understanding of how the SPL community and the open source community can benefit from each other, we presented the configuration tool of

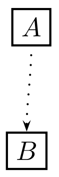
[4]generated using the Colored Integrated Development Environment (CIDE)[4]

MANDATORY		**config** P
		  boolean ”P”
		  select C
		config C
		  boolean ”C”

OPTIONAL		**config** P
		  boolean ”P”
		config C
		  depends on ”P”
		  boolean ”C”

OR		**menu** ”P”
		config P
		  boolean
		config C1
		  boolean ”C1”
		  select P
		config C2
		  boolean ”C2”
		  select P
		config C3
		  boolean ”C3”
		  select P
		endmenu

ALTERNATIVE		**choice**
		  prompt ”P”
		config C1
		  boolean ”C1”
		config C2
		  boolean ”C2”
		config C3
		  boolean ”C3”
		endchoice

IMPLIES		**config** A
		  boolean ”A”
		  requires B
		config B
		  boolean ”B”

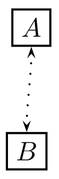
EXCLUDES		**config** A
		  boolean ”A”
		  requires !B
		config B
		  boolean ”B”
		  requires !A

Table 1. Mapping: Feature relations to LKC language

one of the most popular open source projects. We showed how it is used to describe configuration databases, and we have also explained how to describe the semantics of feature models using its language. The feasibility of the approach presented in this work, corroborates with the assumption of similarities between the technical goals that these two communities pursue.

References

[1] The linux 2.5 kernel summit. http://lwn.net/2001/features/KernelSummit/, 2005.

[2] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006.

[3] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, Nov. 1990.

[4] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, pages 311–320, 2008.

[5] D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. On the configuration of non-functional properties in operating system product lines. In *Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS ’05)*, pages 19–25, Chicago, IL, USA, Mar. 2005. Northeastern University, Boston (NU-CCIS-05-03).

[6] R. E. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. *Lecture Notes in Computer Science*, 2186:10–??, 2001.

[7] E. S. Raymond. The cml2 resources page. http://www.catb.org/ esr/cml2/.

[8] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is The Linux Kernel a Software Product Line? In F. van der Linden and B. Lundell, editors, *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, Kyoto, Japan, 2007.

[9] J. Sincero, O. Spinczyk, and W. Schröder-Preikschat. On the Configuration of Non-Functional Properties in Software Product Lines. In *Proceedings of the 11th Software Product Line Conference, Doctoral Symposium (SPLC ’07)*, 2007.

[10] R. Zippel. The linux kernel configurator. http://www.xs4all.nl/ zippel/lc/.