Grammars as Feature Diagrams

Merijn de Jonge Joost Visser
CWI, Amsterdam

Abstract

Feature Diagrams (FDs) have been proposed to describe the configuration space of a software system at the problem level. They can also be used to describe the configuration space of the various components at the solution level. We demonstrate the correspondance of FDs to grammars, and we exploit this correspondance to generate solution configurations from problem configurations. To this end, we view configurations as parse trees, and we obtain a solution configuration by flattening this tree and re-parsing it with the solution grammar. The solution configuration is then fed to the `autobundle` *tool to compose and configure a source tree from all required solution components.*

1 Feature Descriptions

Feature Diagrams (FDs) have been proposed to describe the configuration space of a software system at the level of the problem domain [1, Chapter 4]. The Feature Description Language (FDL) can be used as a textual representation for FDs [2]. Such textual representations are better suited to describe large configuration spaces, and they are amenable to automatic analysis and transformation (e.g., normalization) of configuration spaces.

Fig. 1 and 2 give an example of a feature description for a software renovation factory, as diagram and as text. Here, a renovation factory has two mandatory composite features: the language of the sources to be renovated, and the language in which the factory is implemented. Each of these composite features has several alternative languages as atomic features (mutually exclusive in the second case). Futhermore, the implementation language has support for generic syntax tree traversal as optional feature. Note that in FDL atomic features start with lower case letters and composite features start with uppercase letters and that features are composed using feature operators, such as `all`, `more-of`, and `one-of`.

A specific configuration, i.e., a point in the configuration space, corresponds to a set of atomic features. For instance, {`cobol java traversals`} describes a Cobol renovation factory implemented in Java with support for generic tree traversal.

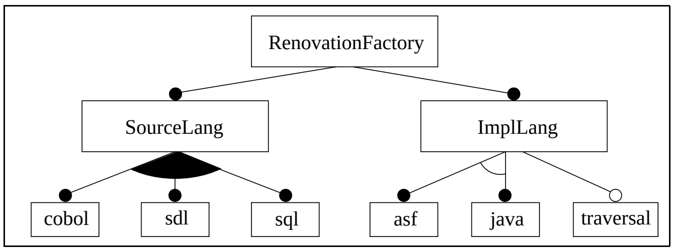

Figure 1. FD for a renovation factory.

RenovationFactory **:**
  all(SourceLang**,** ImplLang**)**
SourceLang **:**
  more-of(*cobol***,** *sdl***,** *sql***)**
ImplLang **:**
  all(one-of(*asf***,** *java***),** *traversals***?)**

Figure 2. Textual representation of Fig. 1.

SourceLang ImplLang	$\rightarrow$ RenovationFactory
(“cobol” | “sdl” | “sql” **)+**	$\rightarrow$ SourceLang
(“asf” | “java” **)** “traversals”**?**	$\rightarrow$ ImplLang

Figure 3. Generated configuration language.

2 From FD to Grammar

Feature descriptions are remarkably similar to *grammars*. For instance, Fig. 3 shows a grammar in the syntax definition language SDF that corresponds to our feature description of Fig. 2. In this analogy, atomic features are terminals, composite features are non-terminals, and feature operators map to syntax operators. The atomic features form the alphabet of a *configuration language*, and a sentence in this language specifies a particular configuration. A parse tree can be seen to model a feature selection. In fact, we implemented a generator that converts any FD to an SDF grammar, and we applied it to generate Fig. 3.

By converting an FD to a grammar, we facilitate the application of syntax tools to feature descriptions. For instance, we can generate a parser from the grammar to serve as configuration validator. To check the validity of a configuration (i.e., a set of atomic features) we simply parse it. (To check feature contraints, a type checker should complement the parser). We can feed the grammar to a structure editor to obtain an interactive configuration editor, which guides its user through the configuration space to a valid configuration.

```
package Meta
version 1.0.1
location=http://www.cwi.nl/projects/MetaEnv
realizes
   asf, traversal?
requires
   Sglr 3.5
```

Figure 4. Package description for `meta`.

```
PackageBase :
   more-of(Meta, JJForester, GrammarBase, JDK, ...)
Meta :
   all(asf, traversals?, Sglr)
JJForester :
   all(JDK, Sglr, traversals)
GrammarBase :
   more-of(cobol, sdl, sql, haskell, c, ...)
JDK :
   java
```

Figure 5. FD for factory components.

3 Solution FDs

Our example feature diagram describes the configuration space at an abstract, implementation-independent level. To bridge the gap between problem and solution, we will start at the level of the implementation components and work our way up.

The *Online Package Base* (OPB) is a repository of language tool components developed at CWI, Universiteit Utrecht, INRIA/Loria, and Vrije Universiteit Amsterdam [4]. Each component is represented in the OPB by a *package description* that states its configuration interface and requirements. Additonally, each package description states a number of *atomic features* that are realized by the corresponding component. An example is shown in Fig. 4. Here, the `meta` package (providing the ASF+SDF Meta-environment) requires the `sglr` package (Scannerless Generalized LR parser) to realize the features `asf` and `traversal`.

Interestingly, these package descriptions can straightforwardly be mapped to composite features. When taken together, the features corresponding to all packages in the OPB form an FD that describes the configuration space spanned by all available components. Fig. 5 shows an excerpt.

4 Parse to solve

A feature description at the implementation level can be converted to an SDF grammar, just like the FD at the problem level. Note that the languages described by both grammars overlap, because the sets of atomic features (alphabets) at both levels overlap. This is no coincidence. The atomic features required at the problem level must correspond to those realized at the solution level if these levels are to be bridged.

Recall that the parser generated from the 'problem grammar' can be used to validate a given configuration. From the 'solution grammar', we can generate a second parser, which we

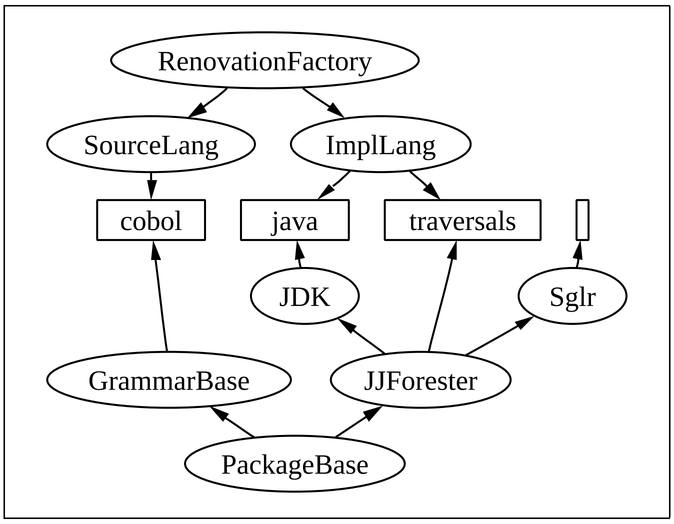

Figure 6. The configuration {`cobol java travesals`} parsed with the problem grammar (above) and the solution grammar (below).

can use to reparse such a validated configuration. If the parse succeeds, the resulting parse tree represents a package selection that implements the required configuration. Fig. 6 shows an example configuration, parsed by both grammars. The leafs of the solution parse tree are the atomic features specified by the configuration. The internal nodes are the packages required to realize those features. If no solutions exist, the parse will fail. If several solutions exist, the parse will be ambiguous.

5 From solution to software bundle

Once the list of required packages has been determined by the reparsing method above, the `autobundle` [3] tool can be used to retrieve the packages from the OPB. This tool performs version resolution, merges the source trees of the required packages, and integrates their configuration and build processes.

The resulting source bundle, with pre-configured packages, forms the implementation of a configuration (product instance) in the problem space.

References

[1] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, 2000.

[2] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 2002.

[3] Merijn de Jonge. Source tree composition. In *Proceedings: Seventh International Conference on Software Reuse*, LNCS. Springer-Verlag, 2002.

[4] Merijn de Jonge, Eelco Visser, and Joost Visser. Collaborative software development. Technical Report SEN-R0113, CWI, 2001.