A Design of a Configurable Feature Model Configurator*

Goetz Botterweck
Lero
University of Limerick
goetz.botterweck@lero.ie

Mikoláš Janota
Lero
University College Dublin
mikolas.janota@ucd.ie

Denny Schneeweiss
BTU Cottbus
Cottbus, Germany
denny.schneeweiss@tu-cottbus.de

Abstract

Our feature configuration tool S^2T^2 Configurator integrates (1) a visual interactive representation of the feature model and (2) a formal reasoning engine that calculates consequences of the user's actions and provides formal explanations. The tool's software architecture is designed as a chain of components, which provide mappings between visual elements and their corresponding formal representations. Using these mappings, consequences and explanations calculated by the reasoning engine are communicated in the interactive representation.

1. Introduction

In the research on feature models different aspects have been addressed. First, there is work on *formal semantics* of feature models [6], which enables us to precisely express the available configurations of a product line in the form of a feature model. Second, there is the *interactive configuration* of feature models, as addressed by visualization of feature models [2] or feature modeling tools [1]. In this paper we strive to link these two worlds. So how can we provide a usable feature model representation, which can be configured interactively and precisely implements the underlying formal semantics?

We address this problem with S^2T^2 Configurator, a research prototype which integrates an interactive visual representation of feature models and a formal reasoning engine.[1] The architecture of the Configurator is designed as a chain of components, which provide mappings between visual elements and their corresponding representations in the formal reasoning engine, see Figure 1. The *Software Engineer* interacts with multiple *Views* of the *Model*. The *Consequencer* infers consequences and provides explanations. A *Translator* serves as a mediator between the representations used in these components.

*This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre.

[1] S^2T^2 stands for " SPL of SPL Techniques and Tools ".

2. Requirements

Before we present the Configurator tool, we have to briefly discuss the required functionality. First, the application has to *load the model* and *translate it into a formal representation*. Subsequently, the user configures the model by *making and retracting decisions*. For Boolean feature models, a user decision is either a *selection* or an *elimination* of a certain feature. Hence, we have four potential configuration states (the power set of $\{true, false\}$): *Undecided*, *Selected*, *Eliminated*, and *Unsatisfiable*.

After any change (loading, user interaction) the tool has to *infer consequences*, taking into account constraints imposed by the model and user decisions. These consequences have to be communicated in the visual representation. We distinguish four sources of configuration: *M = Model* (given in the model), *MC = ModelConsequence* (consequence of M), *U = User* (given by interaction), and *UC = UserConsequence* (consequences of *U*, might rely on *M*).

The tool must *enforce constraints* and disable decisions that lead to configuration states where no valid configuration is possible without retracting decisions ("*backtrack freeness*"). The tool shall *explain* why certain configuration decisions were made automatically. The explanation shall be given within the model by highlighting elements that led to the explained decision.

3. Feature Model

The goal of interactive configuration of feature models led us to a particular design of our modeling language. To be able to map consequences and explanations generated by the Consequencer to visual representations we "chopped up" our feature models in smaller pieces which we call *feature model primitives*. For instance, to describe feature groups, we use primitives like `AlternativeGroup`,

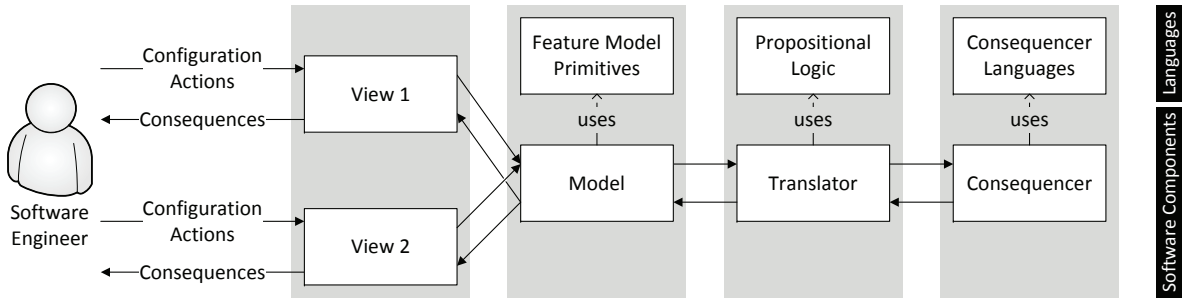


Figure 1. Overview of the components and languages used in S^2T^2 Configurator

`GroupHasParent`, or `GroupHasChild`. Similar primitives exists for other elements typically found in feature models (e.g., root, mandatory and optional subfeatures) and to capture user decisions (selection, elimination). Overall the `FeatureModel` consists of a set of `Features` and a set of `FeaturePrimitives`.

Features can be interpreted as variables and primitives as constraints over these variables. A legal configuration has to fulfill all of these constraints. Hence, if we interpret each primitive by translating it into a formal representation and conjoin all of these translations, this gives us the formal semantics of the overall model. We use a similar structure (variables + constraints) for other more formal languages. Consequently, (1) we can use a *generic design* for all the configurators operating upon these languages and (2) reasoning and explanations are implemented by a *chain of mappings* between constraints in various languages.

When the user starts configuring a model, his decisions can be described by adding primitives, e.g., `SelectedFeature` or `EliminatedFeature`. Since the tool only offers configuration decisions that keep the model in a consistent state, making decisions and adding the corresponding primitives will create a more constrained feature model, which represents a subset of feature configurations of the original model. During this process, the backtrack-freeness of the configurator guarantees that at least one legal configuration remains.

4. User Interface

The meta-model gives us the means for describing a feature model as a set of primitives. Let us see how this is presented to the user. Figure 2(a) shows an example feature model (based on [3]) in S^2T^2 Configurator right after loading.

The features `Car`, `KeylessEntry`, `Body`, `Gear` are mandatory and selected. The `Engine` configuration source for all of these primitives is *M = Model*. Because `Engine Requires Injection`, the configurator infers that the latter has to be selected as well and creates the corresponding `SelectedFeature`-primitive with configuration source $ModelConsequence(MC)$. The configuration states of features are represented as icons (check mark = selected, X = eliminated, empty box = undecided). Icons for features with the source M or MC are shown in gray to indicate that the user cannot change the state.

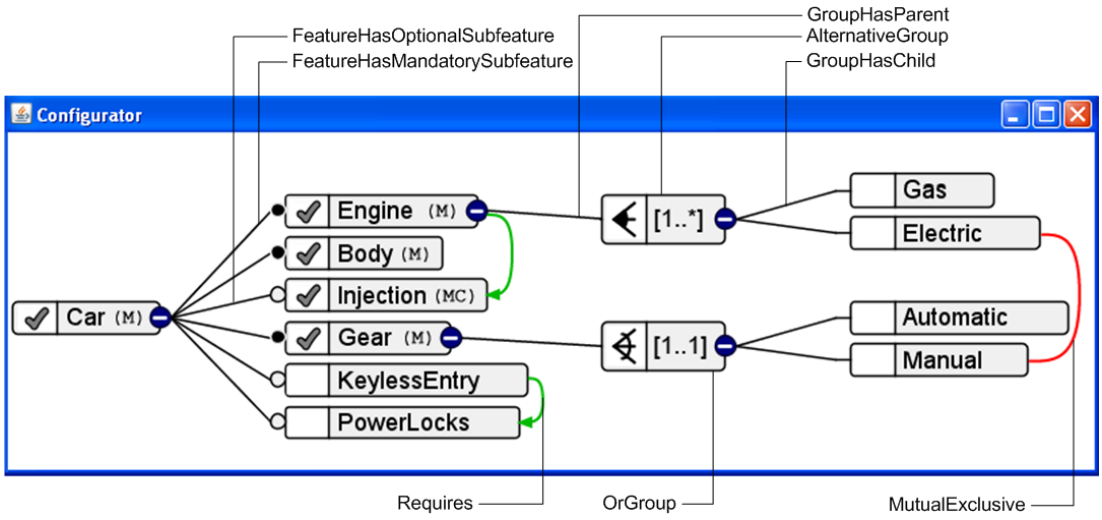
If the user now selects `KeylessEntry` the Consequencer deduces that `PowerLocks` has to be selected as well. Therefore, a `SelectedFeature(PowerLocks)` primitive is created and the view is updated accordingly (see Figure 2(b)).

The user might want to get an explanation why a certain feature was automatically selected or eliminated. This can be done via a pop up menu (see Figure 2(c)). When the user clicks `Explain`, the view queries the configurator for an explanation for the currently focussed feature. Thanks to our software design, the explanation can be mapped back to a list of primitives, which get highlighted in the view. For instance, when asking "Why is PowerLocks selected?" the tool will highlight `SelectedFeature(KeylessEntry)` and the corresponding `Requires({KeylessEntry}, {PowerLocks})`.

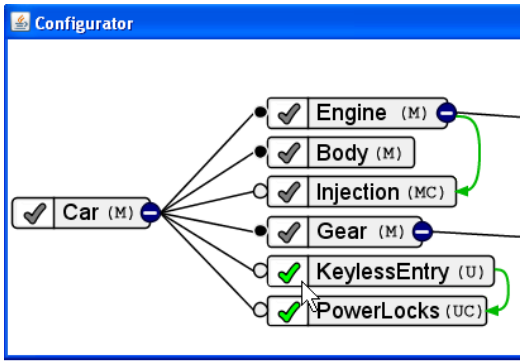
5. Integration between UI and Consequencer

One of our design goals was to allow *multiple* views, which can be used side-by-side, e.g., to focus on different aspects of the same model. Hence, when a configuration decision is made within one view, all resulting updates have to be propagated to the other views.

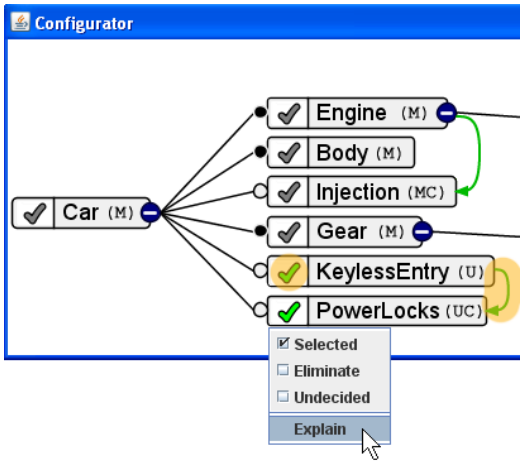
Hidden from the views the model communicates with the Consequencer. When a view commits a change to the model by adding or removing a primitive, this modification is first passed to the Consequencer, which produces consequences. These are then applied to the model. The modification (as triggered by the view) and the application of the consequences are performed atomically, in the sense that



(a) Visual representation of the feature model (call-outs indicate corresponding primitives from the meta-model).



(b) Configuration by interaction and consequences).



(c) Explanation of consequences.

Figure 2. Visual representation of the model in the view of S^2T^2 Configurator

no other operations are allowed before the consequences are applied. This is enforced by the interface, which all views must use to perform operations on the set of primitives. Thus the model is back in a valid state at the end of such an modification-consequence-combination. Subsequently all views are notified about the changes (including the inferred consequences).

6. Translator

The purpose of the translator is to get *from* feature model primitives *to* a format understood by the Consequencer, which is reasoning on some form of mathematical logic.

However, the Translator is not one-way. When providing consequences and explanations, it has to realize communication *from* the Consequencer *to* feature model primitives.

From a Software Engineering perspective, it is important that the tool can be easily used with *different reasoning engines* that realize the Consequencer component. Such engine typically has its own language as it can be used independently of the configurator.

To facilitate this, the Translator decomposes the translation process into several steps, each represented by a different component, a mini-configurator. Each of the mini-configurators communicates via certain language.

The following diagram depicts the mini-configurator chain as realized in the current implementation.

Feature Primitive Configurator (`FPC`) translates between feature primitives and propositional logic. Propositional

Logic Configurator (PLC) provides communication between propositional logic and Reasoning Engine Configurator (REC), which performs the actual reasoning.

The output of FPC is a machine-readable language of propositional logic with logic conjunctives and negation and thus amenable to further conversion to reasoning engines.

Each feature f corresponds to a Boolean variable V_f. And the translation of the primitives is done according to the traditional semantics of feature models [6]. The following table lists several examples.

primitive	**logic formula**
OptionalChild(c, p)	$V_c \rightarrow V_p$
MandatoryChild(c, p)	$V_c \leftrightarrow V_p$
SelectedFeature(f)	V_f
Excludes($\{a, b\}$)	$\neg(V_a \wedge V_b)$

Hence the input of the mini-configurator REC is propositional logic while its output is propositional logic in the Conjunctive Normal Form (CNF). This form is required by the reasoning engine used in the implementation (see Section 7). A different engine might require a different format and this mini-configurator would have to be replaced.

To obtain a uniform view on these languages, we assume that in each of them a sentence comprises a set of *constraints* and a set of *variables*. Depending on the particular language, we use different variables and constraints as shown in the following table.

language	**variables**	**constraints**
feature model	features	feature primitives
prop. logic	Boolean variables	prop. formulas

With respect using Boolean variables for formal representation, note that while in the current implementation the mappings between variables in the different languages are 1-to-1, in general, more complicated mappings may arise. For instance, if we model a variable with a larger domain by using multiple Boolean variables.

This uniform view on the used languages enables us to provide a generic interface which any of the mini-configurators (e.g., FPC) implements. This interface can be found in Figure 3. Constraints can be added and removed later using the methods addConstraint and removeConstraint. The method computeConstraints infers consequences that apply to the given variable while the method explain explains why a given consequence was inferred.

```
interface IConfigurator<Variable, Constraint> {
  void addConstraint(/*@non_null*/Constraint c);
  void removeConstraint(/*@non_null*/Constraint c);
  Set<Constraint> computeConstraints(Variable v);
  Set<Constraint> explain(/*@non_null*/Constraint c);
}
```

Figure 3. Configurator interface

This architecture is rather flexible as any of the components can be easily replaced by another one as long as it implements the same interface and relies only on the instance of the IConfigurator interface of the succeeding component.

7. Consequencer

The reasoning engine used in our implementation relies on a SAT solver [5], which is why it requires the Conjunctive Normal Form. The engine has been developed in our previous work and more details can be found elsewhere [4]. To connect this reasoning engine to the component chain of S^2T^2 Configurator, it was merely necessary to provide the IConfigurator interface for it (see Figure 3).

A different reasoning engine (e.g., [7]), would be added analogously.

References

[1] D. Beuche. Variants and variability management with pure::variants. In *3rd Software Product Line Conference (SPLC 2004), Workshop on Software Variability Management for Product Derivation*, Boston, MA, August 2004.

[2] G. Botterweck, S. Thiel, D. Nestor, S. bin Abid, and C. Cawley. Visual tool support for configuring and understanding software product lines. In *12th International Software Product Line Conference (SPLC 2008)*, Limerick, Ireland, September 2008. ISBN 978-7695-3303-2.

[3] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*, pages 23–34, Washington, DC, USA, 2007. IEEE Computer Society.

[4] M. Janota. Do SAT solvers make good configurators? In *First Workshop on Analyses of Software Product Lines (ASPL '08)*, 2008. Available at http://www.isa.us.es/aspl08.

[5] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. CHAFF: Engineering an efficient SAT solver. In *39th Design Automation Conference (DAC '01)*, 2001.

[6] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Requirements Engineering Conference, 2006. RE 2006. 14th IEEE International*, pages 136–145, 2006.

[7] S. Subbarayan. Integrating CSP decomposition techniques and BDDs for compiling configuration problems. In *Proceedings of the CP-AI-OR*. Springer-Verlag, 2005.