Using Answer Set Programming for Feature Model Representation and Configuration

Varvana Myllärniemi [1] and **Juha Tiihonen**[1] and **Mikko Raatikainen** [1] and **Alexander Felfernig**[2]

Abstract. Feature models are a wide-spread approach used for expressing variability in software product lines. Answer set programming (ASP) is nowadays an increasingly popular approach to configuration knowledge representation. In this paper, we study the similarities between feature modeling and configuration knowledge representation with ASP. We define the feature configuration problem utilizing ASP, and show two different ways using an example of translating the basic feature modeling concepts embodied in the graphical feature models into ASP programs. This way we want to emphasize the role of ASP as a means to tackle the feature configuration problem.

1 Introduction

Features and feature models [11, 17, 18] have been proposed as a means to represent the variability of a software system. Variability in software is defined as the ability of a system to be efficiently extended, changed, customized or configured for use in a particular context [27]. Correct and efficient management of variability is especially important for software product lines. A software product line is a set of products that share a common, managed set of features, a common architecture and a set of reusable assets, thus enabling the preplanned production of products with slightly varying capabilities [7, 10]. In fact, feature modeling has become the *de facto* means to represent and reason about variability in software product lines in academia [6]. Within software product lines, feature models can be used for two purposes: to manage and reason about commonality and variability at the domain engineering level, and to support the derivation of valid products at the application engineering level.

Software product line variability, and consequently, feature models, can grow large and complex. Due to the combinatorial explosion, analyzing feature models and finding a valid feature configuration is infeasible to do manually with large-scale feature models [3]. Thus, there is a need for automated analysis and reasoning of feature models [3]. However, it seems that current feature model analysis focuses on the analysis of the variability, that is, analysis at the domain engineering level, rather than on analysis of the derivation or configuration task. Out of the feature analysis operations listed in [3], only a few analyses are related to derivation: whether a given feature configuration is a valid product, and the operation to enumerate all possible valid configurations [3]. The problem of feature configuration has been studied to some extent, for example, for staged feature configuration [12] that elaborates several stages of making selections and pruning the variability space. Within this paper, we are interested in the simple configuration problem: given a set of requirements for a product, what are the valid feature configurations?

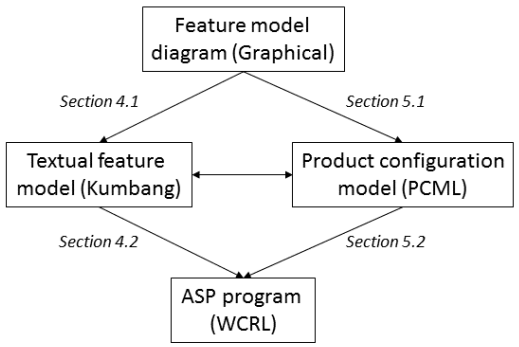

Figure 1. An illustration of how the research problem is addressed in this paper. The languages used to capture each model are marked in parenthesis.

In the field of mechanical and physical products, configuration has a long and successful history as a basis for mass-customization, see, e.g., [15]. The variability of the product is captured in a *configuration model* that represents the taxonomy and compositional structure of a product along with relevant constraints. The configuration task for a configuration model results in a *configuration*, a specification of a product individual [19, 30, 23] that meets the customer requirements.

As a supporting tooling, Answer Set Programming (ASP) is an increasingly important formalism for the representation of configuration models. Configuration is one of the first applications of ASP solving; the requirements of configuration problems were taken into account already in the development of the early ASP tool Smodels [25]. On the one hand, ASP programs have been applied directly to model configuration [24, 28] and reconfiguration [13, 24] problems in research systems. On the other hand, another approach is to model configuration models with a high-level language and to translate the resulting model into a corresponding ASP program [31, 29].

The two disciplines of software product lines and configurable products have similar goals and challenges in the variability management [16, 4]. A major goal of this paper is to show in an easily accessible manner and through concrete examples how ASP can be applied in the context of feature modeling. Previous work has described these aspects on a higher level of abstraction. Therefore, our research problem is to study the similarities between feature modeling and configuration knowledge representation with ASP. For this purpose, the following research questions are set:

- RQ1: How can the feature configuration problem be stated

[1] Aalto University, Finland, email: {firstname.lastname}@aalto.fi
[2] TU Graz, Austria, email: alexander.felfernig@ist.tugraz.at

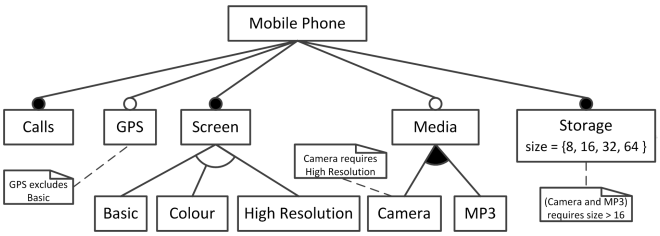

Figure 2. Example feature model slightly extended from [3].

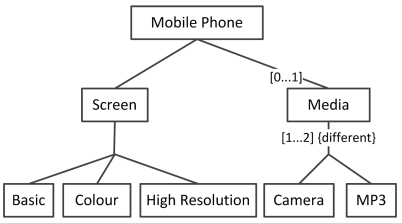

Figure 3. An excerpt from the feature model in Figure 1 modelled with cardinalities, following the notation used in [2].

through ASP?

- RQ2: What are the different ways to represent a feature model diagram as an ASP program?
- RQ3: What are the synergies in the variability management between feature modeling and product configuration?

Figure 1 illustrates the strategy that this paper utilizes to answer the research problem and questions. In particular, it shows how the graphical feature diagrams are represented with textual languages, and these textual languages are then automatically translated to ASP programs. Since the same graphical feature model can be represented both with the textual feature modeling language (Kumbang) as well as with the product configuration language (PCML), it is possible to compare and identify conceptual similarities and differences between software variability management and product configuration. Moreover, the figure illustrates the strategy of utilizing intermediate level languages: this omits the need to manually write ASP programs directly, and consequently, any inherent cognitive difficulties.

The contributions of this paper are the following. Firstly, we adapt the existing work [26] to define the feature configuration problem based on answer sets and stable model semantics. Secondly, we show how the basic concepts of feature models can be represented as ASP programs utilizing a concrete running example. This enables the use of existing ASP solvers to efficiently solve the feature configuration problem. Thirdly, for translating the feature models to ASP programs, we utilize two existing intermediate level languages; these languages enable the product line engineer to operate on domain-specific modeling constructs. Since these two languages originate from different paradigms, this highlights the conceptual similarities between software product line engineering and product configuration.

The remainder of this paper is organized as follows. Section 2 lays out the background as a previous work. Section 3 defines the feature configuration task and problem with ASP. Section 4 shows how graphical feature models can be represented as ASP programs by translating them through a textual feature modeling language called Kumbang (cf. Figure 1). Section 5 demonstrates that the same graphical feature model can be represented by Product Configuration Modeling Language (PCML) and its translation to ASP. Section 6 discusses the similarities of the software variability and traditional product configuration. Section 7 concludes.

2 Background

2.1 Feature modeling

A *feature* in a feature model can be seen as a characteristic of a system that is visible to the end-user [17]. For example, for a software product line for mobile phones, feature `MP3` might represent the capability to listen to and store audio files in MP3 format (see Figure 2). Since features can be used to capture also technological or implementation decisions [18], the definition of a feature has been extended to be a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among product variants [11].

Given a set of features, a *feature model* represents the variability and relations of those features. A feature model is represented as a hierarchically arranged set of features that consists of relations between a parent (or compound) feature and its child features (or subfeatures) and cross-hierarchy constraints [3]. Typically, feature models are presented as graphical diagrams. An example feature model for mobile phones is illustrated in Figure 2.

At least four basic relations between parent and child features can be identified [3]. Firstly, a child feature can be *mandatory* in relation to its parent feature: the child feature must be included in all products that include the parent feature. For example, feature `Calls` is mandatory in relation to feature `Mobile Phone` (see Figure 2). Secondly, a child feature can be *optional* in relation to its parent feature, for example, feature `GPS` can either be selected or left out for all mobile phones. Thirdly, a set of child features can be *alternative* in relation to their parent feature, which means that exactly one of the child features must be selected when the parent feature is in the product. As an example, exactly one of features `Basic`, `Colour`, and `High resolution` must be present in the product that has feature *Screen*. Fourthly, a set of child features can be in *or* relation to their parent feature, which means that one or more of them are present in the product that has the parent feature; this is exemplified by features `Camera` and `MP3` in Figure 2.

Additionally, there can be cross-hierarchy constraints. For example, features `GPS` and `Basic` are mutually exclusive, which means they cannot be in the same product, whereas feature `High resolution` must always be included in a product that contains feature `Camera`. These constraints are presented as annotations in Figure 2.

Various feature models and extensions to basic feature models have been proposed, as discussed in [3].

Firstly, there can be feature models with attributes [12, 5], as illustrated in Figure 2. Feature `Storage` has been characterized with attribute that describes the size in gigabytes, with an enumerated value range. Attributes are typically defined by stating a name and a specific range of values. Typically, a variation point that has a finite number of variants can be represented both as a set of features and as an attribute in a feature.

Secondly, there can be feature models with cardinality [11, 12]. It has been argued that cardinalities can be used to express similar relations as with basic feature relations. For example, Figure 3 illustrates

how a part of the model in Figure 2 is represented with cardinalities.

The usage of feature models varies from an informal documentation or visualization to more rigorous usages enabling even automated analysis. Respectively, the research has matured from the early notations [17] to various formalizations and analyses [3]. One possible usage of feature models is with configurable software product lines [8]: a product can be derived without further development [8] by configuring features, resulting in a model of a product individual.

2.2 Answer Set Programming

As summarized in [14], Answer Set Programming (ASP) has become a popular approach to declarative problem solving. The attractiveness of ASP stems from a combination of a rich and yet simple modeling language and the availability of high-performance solvers. The roots of ASP include knowledge representation, logic programming, (nonmonotonic) reasoning, databases, and Boolean constraint solving.

ASP makes it possible to express the problem as a theory consisting of logic program rules with clear declarative semantics, and the *stable models*, i.e., the *answer sets* of the theory correspond to the solutions to the problem [25].

Programs that follow the Answer Set Programming paradigm are a generalization of normal logic programs. A generalized and unified syntax of ASP programs called *ASP-Core-2* has been defined [9]. This input language has been adopted by many ASP solvers [1]. Optimality criteria, variables and built-in functions can be defined. The syntax of ASP programs is close to Prolog, but the computation method via model generation is different [14].

There are a number of ASP solvers available, see [33], that can tackle a number of complex problems. The best ASP solvers perform well for a range of hard problems; see, for example, problems and results of the Fourth Answer Set Programming Competition [1]. The competition tasks included 3 problems in complexity class P, 15 problems in NP, 3 problems Beyond-NP ($\sum_2^P$), and 5 optimization problems; the domains of the tasks include combinatorial, database, diagnosis, graph, planning and scheduling problems. An example of current, well performing set of tools is Potassco, the Potsdam Answer Set Solving Collection[14], available from [22].

The authors of this paper have applied *weight constraint rule language (WCRL)* that is almost a genuine subset of ASP-Core-2. The languages ASP-Core-2 and WCRL are compatible enough so that the concrete WCRL logic programs generated by our tools are valid input to systems based on ASP-Core-2. This was verified with Clingo version 4.3, available from [22]. Thus, when describing WCRL, we actually describe a part of ASP-Core-2 that is sufficient for this paper. We can do this in a slightly more intuitive yet compact way than we could describe the full ASP-Core-2.

In the following, we describe the basic concepts of weight constraint rules focusing on the concepts needed in the rest of the paper. Instead of explaining the concepts utilizing a running example, these concepts are exemplified for Kumbang in Section 4 and for PCML in Section 5. For further details and examples, please see [25, 9].

Cardinality constraints are used as the primary basic building blocks of the product configuration rules. Cardinality constraints are of the form

$$l\{a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m\}u$$

where l and u are the lower and upper bounds of the constraint. Basic *atoms* are the smallest lexical units, for example a, or b. A literal is an atom b or a not-atom $not\ b$. A cardinality constraint is satisfied by a set of atoms S if the number of those literals in $\{a_1, \ldots, a_n, not\ b_1, \ldots,\ b_m\}$ that are satisfied by S is between the bounds l and u.

A *constraint rule* is an expression of the form

$$C_0 \text{ :- } C_1, \ldots, C_n$$

where the body of the rule consists of a number of cardinality constraints C_i, and the head C_0 cannot contain negated atoms. A program P is then a set of constraint rules.

For product configuration, the following rules are often useful. Firstly, in *choice rules* the number of satisfied atoms in the head must be between l and u:

$$l\{a_1, \ldots, a_n\}u \text{ :- } C_1, \ldots, C_n$$

Secondly, a rule with an empty head yields an *integrity constraint* :- $C_1, \ldots, C_n$, that is, an unsatisfiable constraint that allows specifying inconsistent situations where finding the answer is not possible. Finally, a rule with an empty body is called a *fact*. For example, a fact C_0 states that C_0 is always true.

Given a set of atoms S, a rule C_0 :- $C_1, \ldots, C_n$ is *satisfied* iff S satisfies C_0 whenever S satisfies each of $C_1, \ldots, C_n$. A program P is satisfied by S if each rule in P is satisfied by S. A *stable model* or answer set of a weight constraint rule program is defined as a set of atoms that 1) satisfies the program (is a classical model of the program) and 2) every atom in a stable model is *justified* (*grounded*) by the rules in the program. For example, consider the logical formula $b \wedge (b \wedge \neg c \rightarrow a)$ that has three (classical) models $\{b, c\}$, $\{a, b\}$ and $\{a, b, c\}$. The answer set program

$$b.\ a \text{ :- } b, not\ c.$$

has one stable model $\{a, b\}$. For the formalization of this definition, refer to [25].

Variable-free *ground* weight constraint rules discussed up to now become more practical by allowing the use of variables, function symbols, and predicates. A rule with variables is treated as a short hand for all its ground instantiations with respect to the Herbrand universe of the program. Decidability is retained by allowing only *domain-restricted* rules. Ignoring the details, each variable in a rule must appear in a *domain predicate* which occurs positively in the body of the rule. For example, $p(X)$:- $q(X)$ over constants $\{a, b, c\}$ is an abbreviation of

$$p(a) \text{ :- } q(a), p(b) \text{ :- } q(b), p(c) \text{ :- } q(c)$$

Given predicates and domains, rules with the so called *conditional literals* are frequently applied in product configuration. For example, a fact with predicate $chair$ and domain predicate $member$ states that every board must have exactly one chair that must also be a member:

$$1\ \{chair(X) : member(X)\}\ 1.$$

3 Feature Configuration Problem Utilizing ASP

Research question **RQ1** identified the need to address the feature configuration problem with ASP. In order to utilize ASP and existing solvers (see Section 2.2), one needs to define the basic concepts of the feature configuration problem. Figure 4 defines the feature configuration problem. Here, we adapt the definition of [26] to the domain of feature models in a straightforward manner. We describe each key concept in the definition informally and through examples from the domain of feature models. For further information about the configuration problem in more general terms, see [26].

Definition of the feature configuration task. Given	
CM	a feature configuration model CM translated to a set of rules,
GF	a set of ground facts representing the types in CM and unique identifiers for the instances of types, and
R	a set of rules R representing requirements,
is there a feature configuration C, that is,	
a stable model of $CM \cup S$, such that C satisfies R?	

Figure 4. The definition of the feature configuration task adhering to [26].

Firstly, a feature configuration model CM in Figure 4 specifies the entities, such as features; their properties, such as feature attributes; and composition structure, i.e. the feature tree structure; and the rules how the entities and their properties can be combined in a proper manner for a valid product. More informally, a feature configuration model represents the variability in the product line. For example, the feature model in Figure 2 is represented as one configuration model CM.

Within the definition in Figure 4, a distinction is made between types in a configuration model and instances in a configuration. Types in a configuration model define the properties of their individuals that can appear in a configuration. For example, in Figure 2, feature type storage defines the different attributes and their values, whereas feature instance storage in the actual product has a specific value for the size, for example 16 GB.

Ground facts GF in Figure 4 describe the possible feature instances and the attribute values of instances that can exist in a feature configuration. For example, for the feature `Storage` in Figure 2, a ground fact `featStorage(i).` indicates that feature instance with a unique identifier `i` is of feature type `Storage`. Additionally, a ground fact `hasattr(i,attrsizeGB,16).` tells that this instance has a specific attribute value assignment to indicate 16GB storage.

The set of rules R define requirements thus having a different status from the rules in the configuration model: these requirements represent the requirements that a specific product instance must satisfy. In a valid product configuration, the requirements must be satisfied by a configuration but cannot justify any elements in it. For a feature configuration problem, the requirements are stated as features that must be present in the configuration, or as attribute values that these features have. For example, for Figure 2, one requirement could be stated as `hasattr(i,attrsizeGB,16).`, meaning that there must be 16 GB storage in the product.

A feature configuration C consists of a set of positive and negative atoms. Positive atoms represent the feature instances and attribute values that are in the configuration. Due to the characteristics of ASP and stable models discussed in Section 2.2, the feature instances and attribute values in the configuration C, that is, the positive atoms in C, both *satisfy* the configuration model and its requirements, and are *justified* by them. For example, among the atoms that would be in the feature configuration for Figure 2, an atom `in(i)` indicates the inclusion of feature `Storage`. Further, if the storage is set to 16GB, an atom `hasattr(i,attrsizeGB,16)` is true, while atoms representing other attribute values, such as `hasattr(i,attrsizeGB,32)`, are false.

Consequently, the feature configuration C in the definition above is both consistent and complete. Informally, a *consistent* feature configuration is such that no rules of the configuration model are violated. A *complete* feature configuration is such that all the necessary selections have been made.

An ASP solver can be used to find consistent and complete configurations that meet a set of given requirements, given that such configurations exist. Therefore, the configuration problem definition above and its ASP solution can be used to support both domain and application engineering activities. At the domain engineering level, it can be checked whether the given feature configuration model CM doesn't have any consistent and complete configurations, which implies a self-contradictory model. At the application engineering level, the configuration task can support the finding of consistent and complete configurations, potentially even specifying the requirements R in an iterative manner.

For supporting the user in the configuration task, deducing the consequences of requirements is based on computing an approximation of the set of configurations satisfying the requirements that are valid but not necessarily all consequences are found. Intuitively, the consequences contain a set of facts that must hold for the configurations satisfying the requirements, a set of facts that cannot be true for the given requirements, and a set of unknown facts.

From the practical point of view, a product line engineer needs to capture the product line features and their commonality and variability into a configuration model CM. There are two options for this representation. The first option is to represent the informal feature model, for example, the visual notation in Figure 2, directly as an ASP program. However, this kind of a modeling task requires skills in logic programming, which may not be the case with an average product line engineer. The second option is to capture the feature model with a machine-processable, but human-readable textual language that utilizes directly the concepts known to a product line engineer, and then automatically translate the resulting middle-level model to an ASP program. This translation to ASP also gives the semantics to the middle-level representation language, as well as enables the use of existing ASP solvers for the configuration task. As is illustrated in Figure 1, this paper takes the latter approach.

In the following, we discuss how feature models can be represented as ASP programs, and consequently, how to represent the configuration model CM.

4 Representing Feature Models as ASP Programs through Textual Feature modeling Language

Section 3 presented the feature configuration problem utilizing ASP programs and identified the need to represent a given feature model as an ASP program. In the following, we show how the graphical feature model in Figure 2 and the basic feature modeling concepts can be represented as ASP programs. This is done in two phases, as illustrated in Figure 1: firstly, Section 4.1 shows how the feature model is represented as a textual model in Kumbang, and thereafter Section 4.2 shows how the textual model in Kumbang is translated to WCRL automatically with the Kumbang tool set [20]. Thus, for the purpose of this paper, we utilize WCRL as an example language to construct ASP programs (see also Section 2.2).

4.1 Representing the Feature Model in Kumbang

In order to enable the feature configuration with ASP, the feature model in Figure 2 needs to be represented in a form that is both understandable to a product line engineer, and can be unambiguously translated to an ASP program. For this purpose, we utilize Kumbang language [2], which is a modeling language and an ontology for modeling variability in software product line architectures from the

```
Forfamel model mobilephone
  root feature MobilePhone
feature type MobilePhone {
  contains
    Calls calls;
    GPS gps[0-1];
    Screen screen;
    Media media[0-1];
    Storage storage;
}
feature type Calls {}
feature type GPS {
  constraints not has_instances(Basic);
}
feature type Screen {
  contains (Basic,Colour,HighResolution) type;
}
feature type Basic {}
feature type Colour {}
feature type HighResolution {}
feature type Media {
  contains (Camera,MP3) apps[1-2] {different};
}
feature type Camera {
  constraints has_instances(HighResolution);
}
feature type MP3 {}
feature type Storage {
  attributes Size sizeGB;
  constraints
    (has_instances(Camera) and has_instances(MP3))
    => value(sizeGB) > 16;
}
attribute type Size = { 8, 16, 32, 64 }
```

Figure 5. Feature model from Figure 2 represented with the Kumbang language.

feature and component points of view. Kumbang is built on the product configuration concepts [26], on feature modeling approaches, and on the Koala architecture modeling language [32]. Kumbang is also supported by a set of tools that enable modeling and configuration tasks [20].

Figure 5 illustrates how the feature model in Figure 2 is represented with Kumbang language. In the following, we discuss the main characteristics and differences to the notation used in Figure 2.

Firstly, to adhere to the definition of the feature configuration task in Figure 4, Kumbang differentiates between a configuration model and a configuration. Variability in features is modelled explicitly in a configuration model (illustrated in Figure 5), whereas in a configuration, all variability has been resolved. The elements in a configuration model are referred to as types (for example, feature type `Storage` in Figure 5), while the elements in a configuration are referred to as instances. In contrast, traditional feature modeling notations do not usually make the conceptual distinction between feature types and instances. However, this may cause some difficulties in situations in which the definition of the features needs to be distinct from the feature compositional hierarchy. For example, if features need to be referred to in several places in the hierarchy (c.f., [12]), additional constructs, such as feature cloning or references may be needed. Thus, it seems that the distinction between types and instances allows more expressiveness in the model as such.

Secondly, traditional feature modeling uses a number of compositional relations between features, such as mandatory, optional, and alternative. As illustrated in Figure 3, the multitude of these relations can be expressed with one relation: cardinality. In order to define such relations in the configuration model, the cardinality needs a placeholder in the textual notation: such a placeholder in Kumbang is called a part definition. For example, the part definition `Media media[0-1]` in feature type `MobilePhone` states that `Media` is an optional feature i.e. has a cardinality from zero to one. Part definitions can be more complex: For example, part definition `apps` in type feature `Media` has two possible types of which one or two need to be in a configuration, and if two are selected, they need to be different. The use of part definitions with cardinalities is also advocated in [26].

```
% Definitions of feature types
featureType(featMobilePhone). featureType(featCalls).
featureType(featGPS).         featureType(featScreen).
featureType(featColour).      featureType(featBasic).
featureType(featHighResolution).
featureType(featMedia).       featureType(featMP3).
featureType(featCamera).      featureType(featStorage).

% Root feature MobilePhone
froot(X) :- featMobilePhone(X).
% The feature root is always in the configuration
1 { in(F) : froot(F) } 1.

% Some example part definitions (not all shown)
1{haspart(X1,X2,partDeftype):ppart(X1,X2,partDeftype,I)}1
:- featScreen(X1), in(X1).
1{haspart(X1,X2,partDefapps):ppart(X1,X2,partDefapps,I)}2
:- featMedia(X1), in(X1).

% Attribute definition for feature Storage
1 {hasattr(X,attrDefsizeGB,V):attrSize(V)} 1
:- in(X), featStorage(X).

% Definition of attribute value type Size
attrSize(8). attrSize(16). attrSize(32). attrSize(64).

% Constraint "Camera requires HighResolution"
% Other constraints omitted
constr5(X) :- in(X0),featHighResolution(X0),featCamera(X).
cf(5,X) :- featCamera(X), in(X), not constr5(X).
cff :- cf(5,X), featCamera(X).


% Possible feature instances in the configuration
% are enumerated with unique identifiers and
% corresponding possible parts are defined.
featMobilePhone(i0).
featCalls(i1).          ppart(i0,i1,partDefcalls,1).
featGPS(i2).            ppart(i0,i2,partDefgps,1).
featScreen(i3).         ppart(i0,i3,partDefscreen,1).
featBasic(i4).          ppart(i3,i4,partDeftype,1).
featColour(i5).         ppart(i3,i5,partDeftype,1).
featHighResolution(i6). ppart(i3,i6,partDeftype,1).
featMedia(i7).          ppart(i0,i7,partDefmedia,1).
featCamera(i8).         ppart(i7,i8,partDefapps,1).
featMP3(i9).            ppart(i7,i9,partDefapps,1).
featStorage(i10).       ppart(i0,i10,partDefstorage,1).

% A feature instance is in the configuration
% if it is both actual and possible part of something
in(X2) :- haspart(X1, X2, N), ppart(X1, X2, N, I).
```

Figure 6. The Kumbang representation of Figure 2 (Figure 5) translated to WCRL (some parts omitted and revised for clarity).

Thirdly, the constraints in Figure 2 need to be captured in an unambiguously defined, textual representation. In Figure 5, each constraint is defined in exactly one feature type, utilizing the existing constraint language [2] that supports logical expressions through, e.g., equivalence, implication, universal quantifiers, and references to the compositional hierarchy.

4.2 Representing the Kumbang Model in WCRL

Figure 6 illustrates how the feature model in Figure 5 is translated to WCRL. The translation has been performed automatically with the Kumbang tool set [20] and revised and organized for clarity.

Firstly, each feature type in the configuration model must be defined: for example, fact `featureType(featMobilePhone).` states that object constant `featMobilePhone` represents a feature type. Similarly, the attribute value types are defined, for example, fact `attrSize(8).` states that attribute value type named `Size` has 8 as one possible value.

Secondly, the root of the model must be defined. Rule

$$1\{in(F) : froot(F)\}1.$$

states that if feature type `F` is the root, a valid configuration must have exactly one feature instance selected (`in(F)`) that is instantiated from the root type, defined using predicate `froot`.

Thirdly, the compositional structure of the features must be defined. For each part definition, a rule with the following format is added:

$$n\{haspart(X_1, X_2, P) : ppart(X_1, X_2, P, I)\}m \text{ :- } F(X_1), in(X_1).$$

where F and P are replaced with feature and part names, and n, m replaced with the lower and upper bounds of the cardinality. Predicate `haspart` is used to indicate that a feature instance is instantiated as a part in the configuration, whereas predicate `ppart` is merely stating the possible parts. Together, these predicates justify the inclusion of a feature instance through composition:

$$in(X_2) \text{ :- } haspart(X_1, X_2, N), ppart(X_1, X_2, N, I).$$

Fourthly, attribute definitions are captured with the following rule:

$$1\{hasattr(X, A_d, V) : A_v(V)\}1 \text{ :- } in(X), F(X).$$

where A_d is replaced with the name of the attribute definition, A_v with the name of the attribute value type, and F with the name of the defining feature type.

Finally, the configuration model must also define the identifiers for each feature instance. This enables, for example, to state requirements R about the features that must be present in the configuration (see Figure 4). In Figure 6, the feature instances are given identifiers by enumerating all possible instances in the configuration, for example, fact `featMobilePhone(i0).` gives identifier `i0` to feature `MobilePhone`. Additionally, the identifiers are used to state the possible compositional relations between the instances with the predicate `ppart`. Using these identifiers, it is possible to state the requirements about the feature instances that must be in the configuration, for example, `in(i8).` requires that feature `Camera` must be present in the configuration.

5 Representing Feature Models as ASP Programs through Product Configuration modeling Language

In this Section, the example feature model of Figure 2 is represented with a configuration modeling language designed to model the variability of physical products. We also exemplify the corresponding ASP presentation.

5.1 Representing the Feature Model in PCML

For illustrating the application of a configuration modeling language, we apply *PCML*, Product Configuration Modeling Language [21]. PCML is used by the WeCoTin configurator [29] as the language for representing configuration models. PCML is object-oriented, declarative and has formal implementation-independent semantics.

The main concepts of PCML are *feature types*, their *compositional structure*, *attributes*, and *constraints*. Feature types define the subfeatures (parts) and attributes of their *individuals* that can appear in a configuration. In a configuration, subfeatures (parts) of a feature individual are *realized* with feature individuals. The realizing feature individual(s) "fill the role" created by the subfeature definition. If the cardinality includes 0, an empty realization is possible. A configuration is a non-empty tree of feature individuals and individuals representing attribute values. In addition, the compositional structure is explicitly presented.

The main modeling mechanism of this example is the compositional structure. Feature type `Mobile_Phone_t` in 7 serves as the root

```
configuration model MyProduct
 feature Mobile_Phone_t
   subfeature Screen_p allowed features
     Basic_t, Colour_t, High_resolution_t
     cardinality 1
   subfeature Calls_p
     allowed features Calls_t cardinality 1
   subfeature GPS_p
     allowed features GPS_t cardinality 0 to 1
   subfeature Media_p
     allowed features Media_t cardinality 0 to 1
   subfeature Storage_p
     allowed features Storage_t cardinality 1
   constraint GPS_excludes_Basic not ((present(
     GPS_p)) and (Screen_p individual of Basic_t))
 feature Basic_t
 feature Colour_t
 feature High_resolution_t
 feature Media_t
   subfeature Camera
     allowed features Camera_t cardinality 0 to 1
   subfeature MP3
     allowed features MP3_t cardinality 0 to 1
   constraint Camera_requires_High_resolution
     (present(Camera)) implies
    ($config.Screen_p individual of High_resolution_t)
   constraint Media_requires_Camera_or_MP3
     (present(Camera)) or ( present ( MP3 )  )
   constraint Camera_and_Mp3_require_min_32GB
     ((present(Camera)) and (present(MP3))) implies
     ($config.Storage_p.Size_GB >= 32)
 feature Camera_t
 feature MP3_t
 feature GPS_t
 feature Calls_t
 feature Storage_t
   attribute Size_GB value type integer
     constrained by $ in list(8,16,32,64)
 configuration feature Mobile_Phone_t
```

Figure 7. The feature model of Figure 2 represented with PCML.

of the compositional structure '*configuration type*', see Figure 7. An individual of the type serves as the root of the configuration.

Feature type `Mobile_Phone_t` defines it's compositional structure through a set of *subfeature definitions*. A subfeature definition specifies a *subfeature name*, a non-empty set of *possible subfeature types* (*allowed types* for brevity) and a *cardinality* indicating the valid number of subfeatures. Note that the example of Figure 7 applies a naming convention where the names of feature types end with `_t` and names of subfeatures (parts) with `_p`.

A mandatory subfeature is represented by specifying cardinality `1` and by specifying exactly one allowed type. An example is the mandatory feature `Calls_p`. An optional subfeature is modeled with a subfeature definition whose cardinality is `0 to 1`, e.g. the feature `GPS_p`. Alternative features are modeled with cardinality `1` and more than one allowed type. E.g., feature `Screen_p`. Or-subfeatures are not directly supported by PCML, because with large cardinalities individuals of the same type would be allowed. Therefore for modeling `Media_t`, further subfeatures were defined and a constraint added that enforces the presence of at least one subfeature.

The only attribute of the example is `Storage_t` defining an enumerated integer attribute `Size_GB`.

5.2 Representing the PCML Model in WCRL

Figure 8 shows a partial WCRL/ASP representation of the example feature model. When studying the WCRL/ASP presentation of Figure 8, it is visible that early versions of PCML and WeCoTin applied terminology where feature types were called *component types* and *subfeatures* were called *parts*.

Figure 8 shows the corresponding WCRL presentation (partial). The comments explain the predicates. For a more complete explanation, see [29].

Figure 9 shows one of the 52 answer sets. It represents a feature configuration with `Colour`, `Calls`, `Storage`, Storage `size=16` GB.

```
% if an individual C2 is as part of C1 -> in(C2)
in(C2) :- pa(C1,T,C2,Pn), ppa(T,C1,C2,Pn).
% exclusive parthood: same individual cannot
% be a part of several whole individuals
:- 2{pa(C1,T,C2,Pn):ppa(T,C1,C2,Pn)}, compT_Feature(C2).
%transitivity of is-a hierachy
isa(X,Z):- isa(X,Y), isa (Y,Z),
  compTDom(X), compTDom(Y), compTDom(Z).
% reflexivity of is-a
isa(X,X):- compTDom(X).

%Example types
% Screen_t is a component type and a subtype of 'Feature'
compTDom(compT_Feature).
%Screen types are direct subtypes of 'Feature'
compTDom(compT_Basic_t).
compT_Feature(C) :- compT_Basic_t(C).
isa(compT_Basic_t,compT_Feature).
compTDom(compT_Colour_t).
compT_Feature(C) :- compT_Colour_t(C).
isa(compT_Colour_t,compT_Feature).
compTDom(compT_High_resolution_t).
compT_Feature(C) :- compT_High_resolution_t(C).
isa(compT_High_resolution_t,compT_Feature).
% Storage_t
compTDom(compT_StoraStorage_t).
compT_Feature(C) :- compT_Storage_t(C).
isa(compT_Storage_t,compT_Feature).
% attribute Size_GB of Storage_t
1{prop_Storage_t_Size_GB(X,compT_Storage_t,Y):prSpec(Y)}1
  :- in(X),compT_Storage_t(X).
prSpec(8).
prSpec(16).
prSpec(32).
prSpec(64).

%part name Screen_p
pan(part_Screen_p).
%cardinality 1
1{pa(C1,compT_Mobile_Phone_t,C2,part_Screen_p):
  ppa(compT_Mobile_Phone_t,C1,C2,part_Screen_p)}1 :-
    in(C1),compT_Mobile_Phone_t(C1).
% assignment of possible part individuals of allowed
% types for part screen_p with helper predicate for.
% The automated translation makes such an allocation
% for symmetry breaking, which this example
% does not need
ppa(compT_Mobile_Phone_t,C1,C2,part_Screen_p) :-
  compT_Mobile_Phone_t(C1),compT_Basic_t(C2),
  for(compT_Mobile_Phone_t,C1,C2,part_Screen_p).
ppa(compT_Mobile_Phone_t,C1,C2,part_Screen_p) :-
  compT_Mobile_Phone_t(C1),compT_Colour_t(C2),
  for(compT_Mobile_Phone_t,C1,C2,part_Screen_p).
ppa(compT_Mobile_Phone_t,C1,C2,part_Screen_p) :-
  compT_Mobile_Phone_t(C1),compT_High_resolution_t(C2),
  for(compT_Mobile_Phone_t,C1,C2,part_Screen_p).

% Constraint compilation omitted for brevity.
% it is performed by subexpression.
```

Figure 8. PCML representation of Figure 2 (Figure 7) translated to WCRL (some parts omitted for brevity).

6 Discussion

In this paper, we showed two ways to represent feature models as ASP programs by utilizing existing textual modeling languages designed for feature modeling and product configuration modeling. The use of an intermediate, textual language between the graphical feature models and logic programs is not that common: it seems typical that graphical feature diagrams are directly translated, e.g., to propositional logic [3], rather than utilizing an intermediate textual language.

```
in(ind_compT_Colour_t_1)
pa(ind_compT_Mobile_Phone_t_1,compT_Mobile_Phone_t,
  ind_compT_Colour_t_1,part_Screen_p)
in(ind_compT_Calls_t_1)
pa(ind_compT_Mobile_Phone_t_1,compT_Mobile_Phone_t,
  ind_compT_Calls_t_1,part_Calls_p)
in(ind_compT_Storage_t_1)
pa(ind_compT_Mobile_Phone_t_1,compT_Mobile_Phone_t,
  ind_compT_Storage_t_1,part_Storage_p)
in(ind_compT_Mobile_Phone_t_1)
prop_Storage_t_Size_GB(ind_compT_Storage_t_1,
  compT_Storage_t,16)
```

Figure 9. An answer set representing a feature configuration with `Colour`, `Calls`, `Storage`, Storage `size_GB=16`. Ground atoms were derived from the WCRL of Figure 8. Long atoms are split into two lines.

The benefit of using such intermediate languages and models is that they may be more approachable to product line engineers: they utilize modeling concepts that more or less directly correspond to the concepts used to represent software variability. Such intermediate languages can serve a multitude of purposes: they can be represented graphically and modelled with the aid of graphical tools; they can be created or edited directly if need arises; and they can be automatically translated to ASP programs.

Another option would have been to directly represent or encode the entities and relations in feature models as ASP programs. The benefit of writing directly ASP programs is that the resulting ASP programs most probably are more compact and directly human-readable. The drawback is that logic programming even in the form of ASP programs might be challenging for a product line engineer not trained in computational logic programming.

For simplicity, our representation in this paper covered some basic concepts of feature models. Nevertheless, the languages discussed in Sections 4 and 5 cover much richer sets of modeling constructs. For example, the capability to represent feature inheritance was not utilized in the examples. Similarly, the literature contains numerous proposed extensions of feature models. Some of them are included in our conceptualizations and corresponding tools (e.g. attributes, cardinalities) while some are not. In any case, a detailed discussion about the needed modeling concepts is a future work item.

By mapping the feature modeling notation to both Kumbang and PCML, we demonstrated that both approaches, one tailored for feature modeling and one for product configuration, can be utilized for modeling software variability. A specific addition to the traditional feature modeling concepts done in this paper is to differentiate between feature instances and feature types. This dichotomy, however, parallels with domain and application engineering in software product families and is, therefore, quite natural for software variability although it has not been applied explicitly in feature modeling.

The product configuration community has applied configuration modeling and configuration techniques in full scale production use for decades. It may be that some modeling constructs and approaches related to managing variability could be carried over to describe and analyze feature models. In such a case, existing analyses and respective tools could be readily utilized.

However, the derivation of product lines is not just about configuration: feature models are applicable to a wide range of settings, not just to configurable software product lines. Because of this, the tools intended for product configuration do not necessarily support all the relevant activities in the application engineering phase of software product lines.

In general, due to the availability of a variety of different efficient ASP solvers, it seems beneficial to represent feature models as ASP programs. Despite the fact that the theoretical computational complexity inherent in the feature configuration problem is NP-hard, the current ASP solvers are efficient in calculating the stable models even for programs that represent real-life feature models. We believe that it is more important to find and utilize real problems in testing scalability instead of generated random problems. Consequently, we have configured real problems interactively, without no noticeable delay: see the configuration model with slightly less than 500 variation points [29] and the configuration model with dozens of different types [2] as examples.

7 Conclusions

This study shows how feature models can be represented as ASP programs by means of two different mappings of a graphical feature diagram through intermediate languages. The representation of feature models as ASP programs enables utilizing existing inference engines that are efficient for practical problems. Moreover, the mapping shows significant similarities between feature modeling and product configuration, in particular demonstrating how a feature model diagram can be presented using a product configuration language. This is one concrete step towards better unification between these two similar disciplines of research.

Acknowledgment

We acknowledge the financial support of TEKES as part of the Need 4 Speed (N4S) program of DIGILE and the Austrian Research Promotion Agency (Casa Vecchia, 825889).

REFERENCES

[1] Fourth (open) answer set programming competition - 2013. https://www.mat.unical.it/aspcomp2013, 2013. retrieved 2014-05-05.

[2] Timo Asikainen, Tomi Männistö, and Timo Soininen, 'Kumbang: A domain ontology for modelling variability in software product families', *Advanced engineering informatics journal*, **21**(1), (2007).

[3] D. Benavides, S. Segura, and A. Ruiz-Cortes, 'Automated analysis of feature models 20 years later: A literature review', *Information Systems*, **35**, 615–636, (2010).

[4] David Benavides, Alexander Felfernig, JosA. Galindo, and Florian Reinfrank, 'Automated analysis in feature modelling and product configuration', in *Safe and Secure Software Reuse*, eds., John Favaro and Maurizio Morisio, volume 7925 of *Lecture Notes in Computer Science*, 160–175, Springer Berlin Heidelberg, (2013).

[5] David Benavides, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés, 'Automated reasoning on feature models', in *International Conference on Advanced Information Systems Engineering*, (2005).

[6] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, 'A study of variability models and languages in the systems software domain', *Software Engineering, IEEE Transactions on*, **39**(12), 1611–1640, (Dec 2013).

[7] Jan Bosch, *Design and Use of Software Architectures: Adapting and Evolving a Product-Line Approach*, Addison-Wesley, 2000.

[8] Jan Bosch, 'Maturity and evolution in software product lines: Approaches, artefacts and organization', in *Proc. of Software Product Line Conference*, (2002).

[9] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. ASP-Core-2: Input language format (v.2.03b). ASP Standardization Working Group, https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf, 2012. retrieved 2014-05-05.

[10] Paul Clements and Linda Northrop, *Software Product Lines—Practices and Patterns*, Addison-Wesley, 2001.

[11] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker, 'Formalizing cardinality-based feature models and their specialization', *Softw. Proc. Improv. Pract.*, **10**(1), 7–29, (2005).

[12] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker, 'Staged configuration through specialization and multilevel configuration of feature models', *Software Process: Improvement and Practice*, **10**(2), 143–169, (2005).

[13] Gerhard Friedrich, Anna Ryabokon, Andreas A. Falkner, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner, '(Re)configuration using Answer Set Programming', in *22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), Workshop on Configuration*, eds., Kostyantyn Shchekotykhin, Markus Zanker, and Dietmar Jannach, pp. 17–25, (2011).

[14] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider, 'Potassco: The potsdam answer set solving collection', *AI Communications*, **24**(2), 107–124, (2011).

[15] L. Hotz, A. Felfernig, A. Günter, and J. Tiihonen, 'A Short History of Configuration Technologies', in *Knowledge-based Configuration – From Research to Business Cases*, eds., A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen, chapter 2, 9–19, Morgan Kaufmann Publishers, (2013).

[16] Arnaud Hubaux, Dietmar Jannach, Conrad Drescher, Leonardo Murta, Tomi Mnnist, Krzysztof Czarnecki, Patrick Heymans, Tien Nguyen, and Markus Zanker, 'Unifying software and product configuration: A research roadmap', in *Proceedings of the workshop on configuration (confws12)*, (2012).

[17] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson, 'Feature-oriented domain analysis (foda) feasibility study', Technical Report CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute, (1990).

[18] K.C. Kang, Jaejoon Lee, and P. Donohoe, 'Feature-oriented product line engineering', *IEEE Software*, **19**(4), 58–65, (2002).

[19] S. Mittal and F. Frayman, 'Towards a Generic Model of Configuration Tasks', in *11th International Joint Conference on Artificial Intelligence (IJCAI-89)*, volume 2, pp. 1395–1401, Detroit, Michigan, USA, (1989).

[20] Varvana Myllärniemi, Mikko Raatikainen, and Tomi Männistö, 'Kumbang tools', in *Software Product Line Conference*, volume 2, pp. 135–136, (2007).

[21] Hannu Peltonen, Juha Tiihonen, and Andreas Anderson. Configurator tool concepts and model definition language. Unpublished working document of Helsinki University of Technology, Software Business and Engineering Institute, Product Data Management Group, Espoo, Finland, 2001.

[22] Potassco. Potassco, the Potsdam Answer Set Solving Collection, bundles tools for answer set programming developed at the university of potsdamanswer set programming. SourceForge project `http://potassco.sourceforge.net/`. Accessed 2014-05-06.

[23] Daniel Sabin and Reiner Weigel, 'Product Configuration Frameworks - A Survey', *IEEE Intelligent Systems*, **13**(4), 42–49, (1998).

[24] Gottfried Schenner, Andreas Falkner, Anna Ryabokon, and Gerhard Friedrich, 'Solving object-oriented configuration scenarios with asp', in *15 th International Configuration Workshop*, eds., Michel Aldanondo and Andreas Falkner, pp. 55–62, (2013).

[25] Patrik Simons, Ilkka Niemelä, and Timo Soininen, 'Extending and implementing the stable model semantics', *Artificial Intelligence*, **138**, 181–234, (2002).

[26] Timo Soininen, Ilkka Niemelä, Juha Tiihonen, and Reijo Sulonen, 'Representing Configuration Knowledge with Weight Constraint Rules', in *1st International Workshop on Answer Set Programming: Towards Efficient and Scalable Knowledge (AAAI Technical Report SS-01-01)*, eds., Alessandro Provetti and Tran Cao Son, pp. 195–201, (2001).

[27] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch, 'A taxononomy of variability realization techniques', *Software—Practice and Experience*, **35**(8), 705–754, (2005).

[28] Tommi Syrjänen, 'Including diagnostic information in configuration models', in *First International Conference on Computational Logic (CL 2000)*, eds., John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, LuísMoniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, volume LNCS 1861, pp. 837–851. Springer, (2000).

[29] Juha Tiihonen, Mikko Heiskala, Andreas Anderson, and Timo Soininen, 'Wecotin–a practical logic-based sales configurator', *AI Communications*, **26**(1), 99–131, (2013).

[30] Juha Tiihonen and Timo Soininen, 'Product Configurators - Information System Support for Configurable Products', Technical Report TKO-B137, Helsinki University of Technology, Laboratory of Information Processing Science, (1997). also publsihed in: Increasing Sales Productivity through the Use of Information Technology during the Sales Visit, Hewson Consulting Group.

[31] Juha Tiihonen, Timo Soininen, Ilkka Niemelä, and Reijo Sulonen, 'A practical tool for mass-customising configurable products', in *Proceedings of the 14th International Conference on Engineering Design*, eds., A.Folkeson, K. Gralén, M. Norell, and U. Sellgren, pp. CDROM, paper number 1290, 10 pp., (August 19-21, 2003 2003).

[32] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee, 'The Koala component model for consumer electronics software', *Computer*, **33**(3), 78–85, (March 2000).

[33] Wikipedia. Answer set programming. `http://en.wikipedia.org/wiki/Answer_set_programming`. Accessed 2014-05-06.