Software Product Line Conference 2005
(See update on page 4)

Feature Models, Grammars, and Propositional Formulas

Don Batory

Department of Computer Sciences,
University of Texas at Austin,
Austin, Texas 78712
batory@cs.utexas.edu

Abstract. Feature models are used to specify members of a product-line. Despite years of progress, contemporary tools often provide limited support for feature constraints and offer little or no support for debugging feature models. We integrate prior results to connect feature models, grammars, and propositional formulas. This connection allows arbitrary propositional constraints to be defined among features and enables off-the-shelf satisfiability solvers to debug feature models. We also show how our ideas can generalize recent results on the staged configuration of feature models.

1 Introduction

A key technical innovation of software product-lines is the use of features to distinguish product-line members. A *feature* is an increment in program functionality [29]. A particular product-line member is defined by a unique combination of features. The set of all legal feature combinations defines the set of product-line members [23].

Feature models define features and their usage constraints in product-lines [12][20]. Current methodologies organize features into a tree, called a *feature diagram (FD)*, which is used to declaratively specify product-line members [2]. Relationships among FDs and grammars [21][13], and FDs and formal models/logic programming [7][24][26][27] have been noted in the past, but the potential of their integration is not yet fully realized.

Despite progress, tools for feature models often seem ad hoc; they exhibit odd limitations and provide little or no support for debugging feature models. This is to be expected when a fundamental underpinning of feature models is lacking. In this paper, we integrate prior results to connect FDs, grammars, and propositional formulas. This connection enables general-purpose, light-weight, and efficient *logic truth maintenance systems (LTMSs)*[17] to propagate constraints as users select features so that inconsistent product specifications are avoided — much like syntax-directed editors guarantee compilable programs [28]. This connection also allows us to use off-the-shelf tools, called *satisfiability solvers* or *SAT solvers* [16], to help debug feature models by confirming compatible and incomplete feature sets. To our knowledge, the use of LTMSs and SAT solvers in feature modeling tools is novel.

Our approach is tutorial. We believe it is important that researchers and practitioners clearly see the fundamental underpinnings of feature models, and that

© Springer-Verlag Berlin Heidelberg 2005

light-weight and easy-to-build LTMS algorithms and easy-to-use SAT solvers can help address key weaknesses in existing feature model tools and theories

2 Feature Models

A *feature model* is a hierarchically arranged set of features. Relationships between a *parent* (or *compound*) feature and its *child* features (or *subfeatures*) are categorized as:

- *And* — all subfeatures must be selected,
- *Alternative* — only one subfeature can be selected,
- *Or* — one or more can be selected,
- *Mandatory* — features that required, and
- *Optional* — features that are optional.

Or relationships can have **n:m** cardinalities: a minimum of **n** features and at most **m** features can be selected [12]. More elaborate cardinalities are possible [13].

A *feature diagram* is a graphical representation of a feature model [23]. It is a tree where primitive features are leaves and compound features are interior nodes. Common graphical notations are depicted in Figure 1.

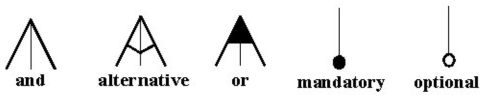
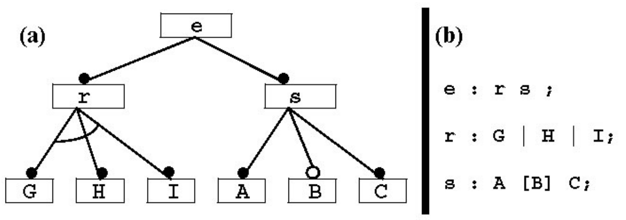


Fig. 1. Feature Diagram Notations

Figure 2a is a feature diagram. It defines a product-line where each application contains two features **r** and **s**, where **r** is an alternative feature: only one of **G, H,** and **I** can be present in an application. **s** is a compound feature that consists of mandatory features **A** and **C**, and optional feature **B**.



```
e : r s ;
r : G | H | I;
s : A [B] C;
```

Fig. 2. A Feature Diagram and its Grammar

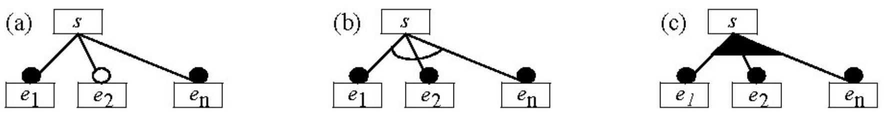


Fig. 3. Parent-Child Relationships in FDs

The connection between FDs and grammars is due to de Jong and Visser [21]. We will use iterative tree grammars. An *iterative grammar* uses iteration (e.g., one-or-more **t+** and zero-or-more **t*** constructs) rather than recursion, to express repetition.

A *tree grammar* requires every token to appear in exactly one pattern, and the name of every production to appear in exactly one pattern. The root production is an exception; it is not referenced in any pattern. More general grammars can be used, but iterative trees capture the minimum properties needed for our discussions.

Figure 3 enumerates the basic hierarchical relationships that can be expressed in a feature diagram. Each has a straightforward iterative tree grammar representation:

- Figure 3a is the production **`s:e`**$_1$ **`e`**$_2$**`...e`**$_n$ assuming all subfeatures are mandatory. If a subfeature is optional (as is **`e`**$_2$), it is surrounded by **`[`**brackets**`]`**. Thus, the production for Figure 3a is **`s:e`**$_1$ **`[e`**$_2$**`]...e`**$_n$.
- Figure 3b is the production: **`s:e`**$_1$**`|`** **`e`**$_2$**`|...|e`**$_n$.
- Figure 3c corresponds to a pair of rules: **`s:t+;`** and **`t:e`**$_1$ **`|e`**$_2$ **`|...|e`**$_n$**`;`** meaning one or more of the **`e`**$_i$ are to be selected. In general, each non-terminal node of a feature diagram is a production. The root is the start production; leaves are tokens. Figure 2b is the grammar of Figure 2a. An application defined by the feature diagram of Figure 2a is a sentence of this grammar.

Henceforth, we use the following notation for grammars. Tokens are **`UPPERCASE`** and non-terminals are **`lowercase`**. **`r+`** denotes one or more instances of non-terminal **`r`**; **`r*`** denotes zero or more. **`[r]`** and **`[R]`** denote optional non-terminal **`r`** and optional token **`R`**. A pattern is a named sequence of (possibly optional or repeating) non-terminals and (possibly optional) terminals. Consider the production:

```
r : b+ A C        :: First
  | [D] E F       :: Second ;
```

The name of this production is **`r`**; it has two patterns **`First`** and **`Second`**. The **`First`** pattern has one or more instances of **`b`** followed by terminals **`A`** and **`C`**. The **`Second`** pattern has optional token **`D`** followed by terminals **`E`** and **`F`**.

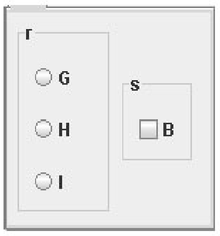


Fig. 4. GUI Specification

Grammars provide a graphics-neutral representation of feature models. For example, the grammar of Figure 2b could be displayed by the FD of Figure 2a or the GUI of Figure 4. (The GUI doesn't display features **`E`** and **`F`**, as they are mandatory — nothing needs to be selected). A popular Eclipse plug-in provides other possible graphical representations of FDs (tree and wizard-based), all of which are derived from a grammar-like specification [2].

In the next section, we show how iterative tree grammars (or equivalently feature diagrams) are mapped to propositional formulas.

3 Propositional Formulas

Mannion was the first to connect propositional formulas to product-lines [26]; we show how his results integrate with those of Section 2. A *propositional formula* is a set of boolean variables and a propositional logic predicate that constrains the values of these variables. Besides the standard $\wedge$, $\vee$, $\neg$, $\Rightarrow$, and $\Leftrightarrow$ operations of propositional logic, we also use **`choose`**$_1$**`(e`**$_1$...**`e`**$_k$**`)`** to mean at most one of the expressions $\mathbf{e}_1 \ldots \mathbf{e}_k$ is true. More generally, $\mathbf{choose}_{n,m}(\mathbf{e}_1 \ldots \mathbf{e}_k)$ means at least n and at most m of the expressions $\mathbf{e}_1 \ldots \mathbf{e}_k$ are true, where $0 \le n \le m \le k$.

A grammar is a compact representation of a propositional formula. A variable of the formula is either: a token, the name of a non-terminal, or the name of a pattern. For example, the production:

```
r : A B          :: P1
  | C [r1]       :: P2 ;
```
(1)

has seven variables: three **{A, B, C}** are tokens, two are non-terminals **{r, r1}**, and two are names of patterns **{P1, P2}**. Given these variables, the rules for mapping a grammar to a propositional formula are straightforward.

Mapping Productions. Consider production $\mathbf{r{:}P}_1|\ldots|\mathbf{P}_n$, which has n patterns $\mathbf{P}_1 \ldots \mathbf{P}_n$. Production **r** can be referenced in one of three ways: **r** (choose one), **r+** (choose one or more), and **r*** (choose zero or more). As **r*** can be encoded as **[r+]** (optionally choose one or more), there are only two basic references: **r** and **r+**. The propositional formulas for both are listed below.

Pattern	Formula
r	$\mathbf{r} \Leftrightarrow \mathbf{choose}_1(\mathbf{P}_1,\ldots,\mathbf{P}_n)$
r+	$\mathbf{r} \Leftrightarrow (\mathbf{P}_1 \vee \ldots \vee \mathbf{P}_n)$

see note at bottom of this page

Mapping Patterns. A *basic term* is either a token or a production reference. A *pattern* is a sequence of one or more basic terms or optional basic terms. Let **P** be the name of a pattern and let $\mathbf{t}_1 \ldots \mathbf{t}_n$ be a sequence of basic terms. The formula for **P** is:

$$\mathbf{P} \Leftrightarrow \mathbf{t}_1 \wedge \mathbf{P} \Leftrightarrow \mathbf{t}_2 \wedge \ldots \wedge \mathbf{P} \Leftrightarrow \mathbf{t}_n \qquad (2)$$

That is, if **P** is included in a design then terms $\mathbf{t}_1 \ldots \mathbf{t}_n$ are also included, and vice versa. Consider pattern **Q** whose second term is optional: $\mathbf{t}_1\ [\mathbf{t}_2] \ldots \mathbf{t}_n$. The formula for **Q** is:

$$\mathbf{Q} \Leftrightarrow \mathbf{t}_1 \wedge \mathbf{t}_2 \Rightarrow \mathbf{Q} \wedge \ldots \wedge \mathbf{Q} \Leftrightarrow \mathbf{t}_n \qquad (3)$$

That is, if **Q** is included in a design then terms $\mathbf{t}_1$ and $\mathbf{t}_n$ are also included, and vice versa. In the case of optional term $\mathbf{t}_2$, if $\mathbf{t}_2$ is selected, **Q** is also selected; however, the converse is not true.

Using these rules, production **(1)** would be translated to the following formula:

$$\mathbf{r} \Leftrightarrow \mathbf{choose}_1(\mathbf{P1},\mathbf{P2}) \wedge \mathbf{P1} \Leftrightarrow \mathbf{A} \wedge \mathbf{P1} \Leftrightarrow \mathbf{B} \wedge \mathbf{P2} \Leftrightarrow \mathbf{C} \wedge \mathbf{r1} \Rightarrow \mathbf{P2}$$

Mapping Grammars. The propositional formula of a grammar is the conjunction of: (i) the formula for each production, (ii) the formula for each pattern, and (iii) the predicate **root=true**, where **root** is the grammar's start production. The propositional formula for the grammar of Figure 2b is:

$$\mathbf{e{=}true} \wedge \mathbf{e} \Leftrightarrow \mathbf{r} \wedge \mathbf{e} \Leftrightarrow \mathbf{s} \wedge \mathbf{r} \Leftrightarrow \mathbf{choose}_1(\mathbf{G},\mathbf{H},\mathbf{I}) \wedge \mathbf{s} \Leftrightarrow \mathbf{A} \wedge \mathbf{B} \Rightarrow \mathbf{s} \wedge \mathbf{s} \Leftrightarrow \mathbf{C} \qquad (4)$$

note that the formula for pattern r is incorrect above. It should be:

(r <=> (P1v...vPn)) and atmost1(P1,...Pn) (**)

The formula (choose1()<=>r) is true when multiple Pi are true and r is false. The corrected formula is never true when multiple Pi are true. Please consult reference [17], p278 for a definition of atmost1() (see TAXONOMY). The GUIDSL tool described in this paper has always implemented (**).

Contrary to current literature, feature models are generally *not* context free grammars. There are often additional constraints, here called *non-grammar constraints*, that govern the compatibility of features. Current tools often limit non-grammar constraints to simple exclusion (choosing feature **`I`** automatically *excludes* a given feature list) and inclusion (choosing feature **`I`** *includes* or *requires* a given feature list). We argue exclusion and inclusion constraints are too simplistic. In earlier work [4], we implemented feature models as attribute grammars enabling us to write constraints of the form:

```
F implies A or B or C
```

This means **`F`** *needs* features **`A`**, **`B`**, or **`C`** or any combination thereof. More often, we found that preconditions for feature usage were based not on a single property but on *sets* of properties that could be satisfied by *combinations* of features, leading to predicates of the form:

```
F implies (A and X) or (B and (Y or Z)) or C
```

meaning **`F`** *needs* the feature pairs (**`A`**,**`X`**), (**`B`**,**`Y`**), (**`B`**,**`Z`**), or **`C`**, or any combination thereof. Exclusion constraints had a similar generality. For this reason, we concluded that non-grammar constraints should be *arbitrary* propositional formulas. By mapping a grammar to a propositional formula, we now can admit arbitrary propositional constraints by conjoining them onto the grammar's formula. In this way, a feature model (grammar + constraints) *is* a propositional formula.

An immediate application of these ideas may help resolve a pesky problem in that feature models do not have unique representations as feature diagrams. (That is, there are multiple ways of expressing the same constraints [12]). It is a daunting task to know if two FDs are equivalent; how tools handle redundant representations is left to tool implementors [14]. It is possible to show that two FDs are equivalent if their propositional formulas are equivalent. See [19] for details.

4 Logic Truth Maintenance Systems

Feature models are the basis for declarative domain-specific languages for product specifications. As users select features for a desired application, we want the implications of these selections to be propagated, so users cannot write incorrect specifications. A *Logic-Truth Maintenance Systems (LTMS)* can used for this purpose.

A LTMS is a classic AI program that maintains the consequences of a propositional formula. An LTMS application is defined by:

- a set of boolean *variables*,
- a set of *propositional logic predicates* to constrain the values of these variables,[1]
- *premises* (assignments to variables that hold universally),
- *assumptions* (assignments to variables that hold for the moment, but may be later retracted), and
- *inferences* (assignments to variables that follow from premises and assumptions).

[1] Equivalently, a single predicate can be used which is the conjunction of the input predicates.

The activities of an LTMS are to:

- compute inferences,
- provide a rationale for variable assignments,
- detect and report contradictions,
- retract and/or make new assumptions, and
- maintain a database of inferences for efficient backtracking.

A *SAT (propositional satisfiability)* solver relies on an LTMS to help it search the combinatorial space for a set of variable assignments that satisfy all predicates. The efficiency of SAT solvers relies on a database of knowledge of previously computed inferences to avoid redundant or unnecessary searches [17].

What makes an LTMS complicated is (a) how it is to be used (e.g., a SAT solver requires considerable support) and (b) the number of rules and variables. If the number is large, then it is computationally infeasible to recompute inferences from scratch; retractions and new assumptions require incremental updates to existing assignments. This requires a non-trivial amount of bookkeeping by an LTMS.

Fortunately, a particularly simple LTMS suffices for our needs. First, the number of rules and variables that arise in feature models isn't large enough (e.g, in the hundreds) for performance to be an issue. (Inferences can be recomputed from scratch in a fraction of a second). Second, searching the space of possible variable assignments is performed *manually* by feature model users as they select and deselect features. Thus, an LTMS that supports only the first three activities previously listed is needed. Better still, standard algorithms for implementing LTMSs are well-documented in AI texts [17]. The challenge is to adapt these algorithms to our needs.

The mapping of LTMS inputs to feature models is straightforward. The variables are the tokens, production names, and pattern names of a grammar. The propositional formula is derived from the feature model (grammar + constraints). There is a single premise: **`root=true`**. Assumptions are features that are manually selected by users. Inferences are variable assignments that follow from the premise and assumptions.

In the next sections, we outline the LTMS algorithms that we have used in building our feature modeling tool **`guidsl`**, whose use we illustrate in Section 5.

4.1 LTMS Algorithms

The *Boolean Constraint Propagation (BCP)* algorithm is the inference engine of an LTMS. Inputs to a BCP are a set of variables $\{v_1 \ldots v_m\}$ and a set of arbitrary propositional predicates $\{p_1 \ldots p_n\}$ whose conjunction $p_1 \wedge \ldots \wedge p_n$ defines the *global constraint (GC)* on variable assignments (i.e., the formula of a feature model). BCP algorithms require the GC to be in *conjunctive normal form (CNF)* [17]. Simple and efficient algorithms convert arbitrary p_j to a conjunction of clauses, where a *clause* is a disjunction of one or more *terms*, a term being a variable or its negation [17].

BCP uses three-value logic (**`true`**, **`false`**, **`unknown`**) for variable assignments. Initially, BCP assigns **`unknown`** to all variables, except for premises which it assigns **`true`**. Given a set of variable assignments, each clause **`C`** of GC is either:

- *satisfied*: some term is **`true`**.
- *violated*: all terms are **`false`**.

- *unit-open*: one term is **unknown**, the rest are **false**.
- *non-unit open*: more than one term is **unknown** and the rest are **false**.

A unit-open term enables the BCP to change the **unknown** assignment to **true**. Thus, if clause **C** is **x**∨¬**y** and **x** is **false** and **y** is **unknown**, BCP concludes **y** is **false**.

The BCP algorithm maintains a stack **S** of clauses to examine. Whenever it encounters a violated clause, it signals a contradiction (more on this later). Assume for now there are no contradictions. The BCP algorithm is simple: it marches through **S** finding unit-open clauses and setting their terms.

```
while (S is not empty) {
   c = S.pop();
   if (c.is_unit_open) {
       let t be term of c whose value is unknown;
       set(t);
    }
 }
```

set(t) — setting a term — involves updating the term's variable's assignment (e.g., if **t** is ¬**y** then **y** is assigned **false**), pushing unit-open terms onto **S**, and signalling contradictions:

```
set variable of t so that t is true;
for each clause C of GC containing ¬t {
   if (C.is_unit_open) S.push(C);
   else
   if (C.is_violated) signal_contradiction();
}
```

Invoking BCP on its initial assignment to variables propagates the consequences of the premises. For each subsequent assumption, the variable assignment is made and BCP is invoked. Let **L** be the sequence of assumptions (i.e., user-made variable assignments). The consequences that follow from **L** are computed by:

```
for each variable l in L {
   set(l);
   BCP();
}
```

If an assumption is retracted, it is simply removed from **L**.

A contradiction reveals an inconsistency in the feature model. When contradictions are encountered, they (and their details) must be reported to the feature model designers for model repairs.

Example. Suppose a feature model has the contradictory predicates **x**⇒**y** and **y**⇒¬**x**. If **x=true** is a premise, BCP infers **y=true** (from clause **x**⇒**y**), and discovers clause (**y**⇒¬**x**) to be violated, thus signalling a contradiction.

Explanations for why a variable has a value (or why a contradiction occurs) requires extra bookkeeping. Each time BCP encounters a unit-open clause, it keeps a record of its conclusions by maintaining a 3-tuple of its actions **<conclusion**, **reason**, **{antecedents}>** where *conclusion* is a variable assignment, *reason* is the predicate (or clause) that lead to this inference, and *antecedents* are the 3-tuples of variables whose values were referenced. By traversing antecedents backwards, a justification for a conclusion can be presented in a human-understandable form.

Example. The prior example generates a pair of tuples: `#1:<x=true, premise, {}}` and `#2:<y=true`, `x⇒y`, `{#1}>`. The explanation for `y=true` is: `x=true` is a premise and `y=true` follows from `x⇒y`.

4.2 A Complete Specification

A product specification (or equivalently, a variable assignment) is *complete* if the GC predicate is satisfied. What makes this problem interesting is how the GC predicate is checked. Assume that a user specifies a product by selecting features from a GUI or FD. When a feature is selected, the variable for that feature is set to `true`; a deselection sets it to `unknown`. (Inferencing can set a variable to `true` or `false`). Under normal use, users can only declare the features that they want, *not what they don't want*.

At the time that a specification is to be output, all variables whose values are `unknown` are assumed `false` (i.e., these features are not to be in the target product). The GC is then evaluated with this variable assignment in mind. If the GC predicate is satisfied, a valid configuration of the feature model has been specified. However, if a clause of GC fails, then either a complete sentence has not yet been specified or certain non-grammar constraints are unsatisfied. In either case, the predicate that triggered the failure is reported thus providing guidance to the user on how to complete the specification. This guidance is usually helpful.

5 An Example

We have built a tool, called `guidsl`, that implements the ideas in the previous sections. `guidsl` is part of the AHEAD Tool Suite [5] a set of tools for product-line development that support feature modularizations and their compositions. In the following section, we describe a classical product line and the `guidsl` implementation of its feature model.

5.1 The Graph Product Line (GPL)

The *Graph Product-Line (GPL)* is a family of graph applications that was inspired by early work on modular software extensibility [29]. Each GPL application implements one or more graph algorithms. A `guidsl` feature model for GPL (i.e., its grammar + constraints) is listed in Figure 5, where token names are not capitalized.

The semantics of the GPL domain are straightforward. A graph is either `Directed` or `Undirected`. Edges can be `Weighted` with non-negative numbers or `Unweighted`. A graph application requires at most one search algorithm: depth-first search (`DFS`) or breadth-first search (`BFS`), and one or more of the following algorithms:

- **Vertex Numbering** (`Number`): A unique number is assigned to each vertex.
- **Connected Components (`Connected`):** Computes the *connected components* of an undirected graph, which are equivalence classes under the reachable-from relation. For every pair of vertices `x` and `y` in a component, there is a path from `x` to `y`.
- **Strongly Connected Components (`StrongC`):** Computes the *strongly connected components* of a directed graph, which are equivalence classes under the reachable relation. Vertex `y` is reachable from vertex `x` if there is a path from `x` to `y`.
- **Cycle Checking** (`Cycle`): Determines if there are cycles in a graph. A cycle in directed graphs must have at least 2 edges, while in undirected graphs it must have at least 3 edges.
- **Minimum Spanning Tree (`MSTPrim`, `MSTKruskal`):** Computes a *Minimum Spanning Tree (MST)*, which contains all the vertices in the graph such that the sum of the weights of the edges in the tree is minimal.
- **Single-Source Shortest Path (`Shortest`):** Computes the shortest path from a source vertex to all other vertices.

```
 // grammar

 GPL : Driver Alg+ [Src] [Wgt] Gtp :: MainGpl ;
 Gtp : Directed | Undirected ;
 Wgt : Weighted | Unweighted ;
 Src : BFS | DFS ;
 Alg : Number | Connected | Transpose StronglyConnected :: StrongC
        | Cycle | MSTPrim | MSTKruskal | Shortest ;
Driver : Prog Benchmark :: DriverProg ;

 %% // constraints

 Number implies Src ;
 Connected implies Undirected and Src ; StrongC implies Directed
 and DFS ;
 Cycle implies DFS ;
 MSTKruskal or MSTPrim implies Undirected and Weighted ;
 MSTKruskal or MSTPrim implies not (MSTKruskal and MSTPrim) ; //#
 Shortest implies Directed and Weighted ;
```

Fig. 5. GPL Model

The grammar that defines the order in which GPL features are composed is shown in Figure 5. Not all combinations of features are possible. The rules that govern compatibilities are taken directly from algorithm texts [11] and are listed in Figure 6. These constraints are listed as additional propositional formulas (below the **%%** in Figure 5). When combined with the GPL grammar, a feature model for GPL is defined. Note: **`MSTKruskal`** and **`MSTPrim`** are mutually exclusive (constraint **#** in Figure 5); at most one can be selected in a GPL product.

Algorithm	**Required Graph Type**	**Required Weight**	**Required Search**
Vertex Numbering	Any	Any	BFS, DFS
Connected Components	Undirected	Any	BFS, DFS
Strongly Connected Components	Directed	Any	DFS
Cycle Checking	Any	Any	DFS
Minimum Spanning Tree	Undirected	Weighted	None
Shortest Path	Directed	Weighted	None

Fig. 6. Feature Constraints in GPL

The GUI that is generated from Figure 5 is shown in Figure 7. The state that is shown results from the selection of **`MSTKruskal`** — the **`Weighted`** and **`Undirected`** features are automatically selected as a consequence of constraint propagation. Further, **`Shortest`**, **`MSTPrim`**, **`StrongC`**, **`Unweighted`**, and **`Directed`** are greyed out, meaning that they are no longer selectable as doing so would create an inconsistent specification. Using an LTMS to propagate constraints, users can only create correct specifications. In effect, the generated GUI is a declarative domain-specific language that acts as a syntax-directed editor which prevents users from making certain errors.

Although not illustrated, **`guidsl`** allows additional variables to be declared in the constraint section to define properties. Feature constraints can then be expressed in terms of properties, like that in [4], to support our observations in Section 3.

Another useful capability of LTMSs is to provide a justification for automatically selected/deselected features. We have incorporated this into **`guidsl`**: placing the mouse over a selected feature, a justification (in the form of a proof) is displayed. In the example of Figure 7, the justification for **`Undirected`** being selected is:

```
MSTKruskal because set by user
Undirected because ((MSTKruskal or MSTPrim)) implies
((Undirected and Weighted))
```

Meaning that **`MSTKruskal`** was set by the user, and **`Undirected`** is set because the selection of **`MSTKruskal`** implies **`Undirected`** and **`Weighted`**. More complex explanations are generated as additional selections are made.

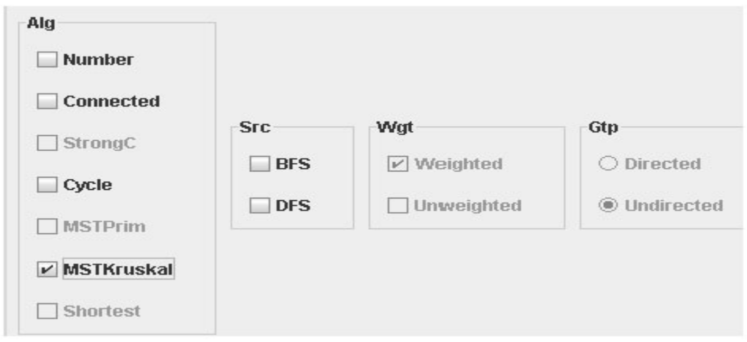


Fig. 7. Generated GUI for the GPL Model

5.2 Debugging Feature Models

Debugging a feature model without tool support is notoriously difficult. When we debugged feature models prior to this work, it was a laborious, painstaking, and error-prone effort to enumerate feature combinations. By equating feature models with propositional formulas, the task of debugging is substantially simplified.

An LTMS is helpful in debugging feature models, but only to a limited extent. Only if users select the right combination of features will a contradiction be exposed. But models need not have contradictions to be wrong (e.g., **`Number implies`**

`Weight`). More help is needed. Given a propositional formula and a set of variable assignments, a SAT solver can determine whether there is a value assignment to the remaining variables that will satisfy the predicate. Thus, debugging scripts in **`guidsl`** are simply statements of the form **`<S,L>`** where **`L`** is a list of variable assignments and **`S`** is **`true`** or **`false`**. If **`S`** is **`true`**, then the SAT solver is expected to confirm that **`L`** is a compatible set of variable assignments; if **`S`** is **`false`**, the solver is expected to confirm that **`L`** is an incompatible set of assignments. Additional simple automatic tests, not requiring a SAT solver, is to verify that a given combination of features defines a product (i.e., a legal and complete program specification). Both the SAT solver and complete-specification-tests were instrumental in helping us debug the GPL feature model.

It is straightforward to list a large number of tests to validate a model; test suites can be run quickly. (SAT solvers have become very efficient, finding variable assignments for thousands of variables in minutes). Although we cannot prove a model is correct, we are comforted by the fact that we can now run a *much* more thorough set of tests on our models automatically than we could have performed previously.

6 Staged Configuration Models

Staged configuration has recently been proposed as an incremental way to progressively specialize feature models [13][14]. At each stage, different groups or developers make product configuration choices, rather than a configuration being specified by one person at one time. Specializations involve the selection or deselection of features and adding more constraints (e.g., converting a one-or-more selection to single selection).

Staged configuration is accomplished by (1) simplifying the grammar by eliminating choices or making optional choices mandatory, and (2) simplifying the non-grammar constraints. Both are required ([14] addresses grammar simplification). By limiting changes *only* to grammars, it is possible to preselect **`MSTKruskal`** and deselect **`Unweighted`** in a staged configuration and adjust the GPL grammar (making **`MSTKruskal`** mandatory and removing **`Unweighted`**). But the resulting model is unsatisfiable, as **`MSTKruskal`** requires **`Unweighted`**.

A generalization of the GUI presented earlier could be used to accomplish staged specifications. Each selectable feature will require a toggle that allows a feature to be selected (**`true`**), deselected (**`false`**), or to postpone its choice to a later stage (**`unknown`**). In this way, designers can distinguish features that are preselected from those that are permanently removed. The LTMS algorithm remains unchanged; constraints are propagated as before guaranteeing that the resulting model is consistent. Inferred feature selections and deselections can be used to further simplify the grammar and its non-grammar constraints.

More generally, where constraints on non-boolean variables (e.g. performance constraints) are part of a feature model, a more general logic, constraint propagation algorithms, and predicate simplification algorithms will be needed [15]. However, our work applies to many existing feature models, and we believe that current results on staged configuration can be improved for these cases with our suggestions.

7 Related Work

There is a great deal of prior work on feature modeling. For brevity, we focus on the key papers that are relevant. Some feature modeling tools support arbitrary propositional formulas [8][10], but these formulas are validated at product-build time, not incrementally as features are selected. We are aware that technologies that dynamically prune the design space — similar to that presented in this paper — may be known to pockets of researchers in industry (e.g., [1][7][18]), but the basic relationship of feature models, attribute grammars, and propositional formulas does not seem to be widely appreciated or understood.

The connection of feature models to grammars is not new. In 1992, Batory and O'Malley used grammars to specify feature models [3], and in 1997 showed how attribute grammars expressed non-grammar constraints [4]. In 2002, de Jonge and Visser recognized that feature diagrams were context free grammars. Czarnecki, Eisenecker, et al. have since used grammars to simplify feature models during staged configuration [12].

The connection of product-line configurations with propositional formulas is due to Mannion [26]. Beuche [7] and Pure::Variants [27] translate feature models into Prolog. Prolog is used as a constraint inference engine to accomplish the role of an LTMS. Non-grammar constraints are expressed by inclusion and exclusion predicates; while user-defined constraints (i.e., Prolog programs) could be arbitrary. We are unaware of tools that follow from [7] to debug feature models.

Neema, Sztipanovits, and Karsai represent design spaces as trees, where leaves are primitive components and interior nodes are design templates [24]. Constraints among nodes are expressed as OCL predicates, and so too are resource and performance constraints. *Ordered binary decision diagrams (OBDDs)* are used to encode this design space, and operations on OBDDs are used to find solutions (i.e., designs that satisfy constraints), possibly through user-interactions.

Concurrently and independently of our work, Benavides, Trinidad, and Ruiz-Cortes [6] also noted the connection between feature models and propositional formulas, and recognized that handling additional performance, resource, and other constraints is a general *constraint satisfaction problem (CSP)*, which is not limited to the boolean CSP techniques discussed in this paper. We believe their work is a valuable complement to our paper; read together, it is easy to imagine a new and powerful generation of feature modeling tools that leverage automated analyses.

8 Conclusions

In this paper, we integrated existing results to expose a fundamental connection between FDs, grammars, and propositional formulas. This connection has enabled us to leverage light-weight, efficient, and easy-to-build LTMSs and off-the-shelf SAT solvers to bring useful new capabilities to feature modeling tools. LTMSs provide a simple way to propagate constraints as users select features in product specifications. SAT solvers provide automated support to help debug feature models. We believe that the use of LTMSs and SAT solvers in feature model tools is novel. Further, we explained how work on staged configuration models could be improved by integrating non-grammar constraints into a staging process.

We believe that the foundations presented in this paper will be useful in future tools for product-line development.

Acknowledgements. I thank Ben Kuipers and Ray Mooney for directing me to the literature on LTMSs. I gratefully acknowledge the helpful comments from the referees, Mark Grechanik, Jack Sarvela, and Jack Greenfield. I thank Daniel Le Berre for his help with the SAT4J solver. I also thank Jun Yuan, Karim Jamal, and Maria Zolotova for their help in building `guidsl`.

References

[1] American Standard, http://www.americanstandard-us.com/planDesign/
[2] M. Antkiewicz and K. Czarnecki, "FeaturePlugIn: Feature Modeling Plug-In for Eclipse", OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, 2004.
[3] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", ACM TOSEM, October 1992.
[4] D. Batory and B.J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators", IEEE TSE, February 1997, 67-82.
[5] D. Batory, AHEAD Tool Suite, www.cs.utexas.edu/users/schwartz/ATS.html
[6] D. Benavides, P. Trinidad, and A. Ruiz-Cortes, "Automated Reasoning on Feature Models", Conference on Advanced Information Systems Engineering (CAISE), July 2005.
[7] D. Beuche, "Composition and Construction of Embedded Software Families", Ph.D. thesis, Otto-von-Guericke-Universitaet, Magdeburg, Germany, 2003.
[8] Big Lever, GEARS tool, http://www.biglever.com/
[9] BMW, http://www.bmwusa.com/
[10] Captain Feature, https://sourceforge.net/projects/captainfeature/
[11] T.H. Cormen, C.E. Leiserson, and R.L.Rivest. Introduction to Algorithms, MIT Press,1990.
[12] K. Czarnecki and U. Eisenecker. Generative Programming Methods, Tools, and Applications. Addison-Wesley, Boston, MA, 2000.
[13] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing Cardinality-based Feature Models and their Specialization", Software Process Improvement and Practice, 2005 10(1).
[14] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models", Software Process Improvement and Practice, 10(2), 2005.
[15] K. Czarnecki, private correspondence, 2005.
[16] N. Eén and N. Sörensson, "An extensible SAT solver". 6th International Conference on Theory and Applications of Satisfiability Testing, LNCS 2919, p 502-518, 2003.
[17] K.D. Forbus and J. de Kleer, Building Problem Solvers, MIT Press 1993.
[18] Gateway Computers. http://www.gateway.com/index.shtml

[19] M. Grechanik and D. Batory, “Verification of Dynamically Reconfigurable Applications”, in preparation 2005.

[20] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi, Software Factories: Assembling Applications with Patterns, models, Frameworks and Tools, Wiley, 2004.

[21] M. de Jong and J. Visser, “Grammars as Feature Diagrams”.

[22] D. Streitferdt, M. Riebisch, I. Philippow, “Details of Formalized Relations in Feature Models Using OCL”. ECBS 2003, IEEE Computer Society, 2003, p. 297-304.

[23] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. “Feature-Oriented Domain Analysis (FODA) Feasibility Study”. Technical Report, CMU/SEI-90TR-21, November 1990.

[24] S. Neema, J. Sztipanovits, and G. Karsai, “Constraint-Based Design Space Exploration and Model Synthesis”, EMSOFT 2003, LNCS 2855, p. 290-305.

[25] R.E. Lopez-Herrejon and D. Batory, “A Standard Problem for Evaluating Product-Line Methodologies”, GCSE 2001, September 9-13, 2001 Messe Erfurt, Erfurt, Germany.

[26] M. Mannion, “Using first-order logic for product line model validation”. 2nd Software Product Line Conf. (SPLC2), #2379 in LNCS, 176–187, 2002.

[27] Pure-Systems, “Technical White Paper: Variant Management with pure::variants”, www.pure-systems.com, 2003.

[28] T. Teitelbaum and T. Reps, “The Cornell Program Synthesizer: a Syntax-Directed Programming Environment”, CACM, v.24 n.9, p.563-573, Sept. 1981.

[29] P. Zave, “FAQ Sheet on Feature Interactions”, www.research.att.com/~pamela/faq.html