How to Complete an Interactive Configuration Process?

Configuring as Shopping

Mikoláš Janota[1], Goetz Botterweck[2], Radu Grigore[3], and Joao Marques-Silva[3]

[1] Lero, University College Dublin, Ireland
[2] Lero, University of Limerick, Ireland
[3] University College Dublin, Ireland

Abstract. When configuring customizable software, it is useful to provide interactive tool-support that ensures that the configuration does not breach given constraints. But, when is a configuration complete and how can the tool help the user to complete it? We formalize this problem and relate it to concepts from non-monotonic reasoning well researched in Artificial Intelligence. The results are interesting for both practitioners and theoreticians. Practitioners will find a technique facilitating an interactive configuration process and experiments supporting feasibility of the approach. Theoreticians will find links between well-known formal concepts and a concrete practical application.

1 Introduction

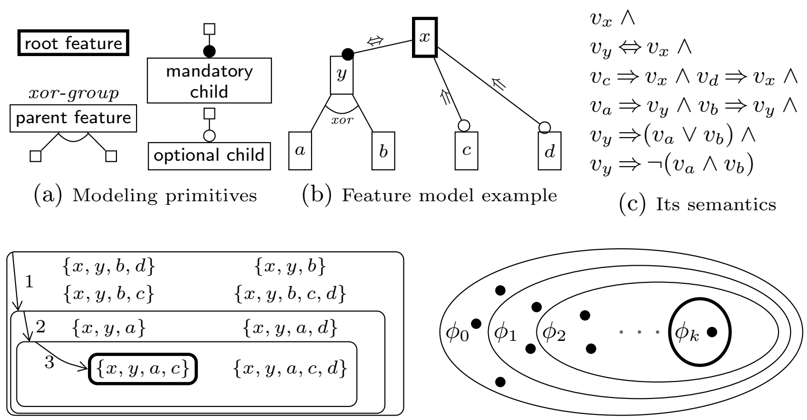
Software Product Lines (SPLs) build on the assumption that when developing software-intensive systems it is advantageous to decide upfront which products to include in scope and then manage construction and reuse systematically [4].

This approach is suitable for families of products that share a significant amount of their user-visible or internal functionality. Parnas identified such *program families* as: ... *sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members.* [20]

A key aspect of SPLs is that the scope of products is defined and described *explicitly* using models of various expressivity [14,1,23,10]. Conceptually, we can consider an SPL as a mapping from a *problem space* to a *solution space*. The problem space comprises requirements that members of the product line satisfy, and, the solution space comprises possible realizations, e.g., programs in C^{++}. These spaces are defined by some constraints, i.e., requirements or solutions violating the constraints are not within the respective space.

A specification for a new product is constructed from the requirements which must be matched to the constraints that define the scope of the product line. In effect, the purchaser picks a particular member of the problem space. Subsequently, software engineers are responsible for delivering a product, a member of the solution space, corresponding to the given specification.

Fig. 1: FODA notation and Configuration Processes



(a) Modeling primitives (b) Feature model example

$$
\begin{aligned}
& v_x \wedge \\
& v_y \Leftrightarrow v_x \wedge \\
& v_c \Rightarrow v_x \wedge v_d \Rightarrow v_x \wedge \\
& v_a \Rightarrow v_y \wedge v_b \Rightarrow v_y \wedge \\
& v_y \Rightarrow (v_a \vee v_b) \wedge \\
& v_y \Rightarrow \neg(v_a \wedge v_b)
\end{aligned}
$$

(c) Its semantics

(d) Example configuration of Fig. 2b (e) Configuration schematically

If the problem space is complex, picking one of its members is not trivial. Hence, we strive to support interactive configuration with *configurator* tools. Despite the fact that this has been researched extensively (see [17,8,1,24,11,9]), little attention has been paid to the completion of a configuration process. Namely, how shall we treat variables of the configuration model that have not been bound by the user at all? (This issue has been noted by Batory in [1, Section 4.2]).

This article studies this problem (Sect. 2.2) and designs an enhancement of a configurator that helps the user to get closer to finishing the configuration process by binding variability without making decisions for the user, i.e., it is aiming at not being overly smart. The article focuses mainly on this functionality for propositional configuration (Sect. 3) and it relates to research on Closed World Assumption (Sect. 3.4). The general, non-propositional, case is conceptualized relying on the notion of preference (Sect. 4).

2 Background and Motivation

Kang et al. developed a methodology *Feature Oriented Domain Analysis* (*FODA*), where *feature models* are used to carry out *domain analysis*—a systematic identification of variable and common parts of a family of systems [14]. For the purpose of this article, it is sufficient to consider a feature as "*a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system*" [14], and a product as a combination of its features. A *Software Product Line* is a system for developing products from a certain family captured as a set of feature combinations defined by a *feature model*. The semantics of a feature model is typically defined with propositional logic. Each feature f is represented by a propositional variable v_f and a Boolean formula is constructed as a conjunction of formulæ representing the different modeling primitives. The satisfying assignments of the resulting formula define the set of possible feature combinations [23]. A popular notation is the *FODA notation* [14] with the primitives in Fig. 2a exemplified by Fig. 2b whose semantics is in Fig. 2c. The corresponding feature combinations, defining the problem space, are listed in Fig. 2d. The FODA notation has several extensions, e.g., *feature attributes* represent values such as price. For general attributes, Boolean logic is insufficient and the semantics is expressed as a Constraint Satisfaction Problem [2].

2.1 Configuration Process

In the interactive configuration process, the user specifies his requirements step-by-step, gradually shrinking the problem space (see Fig. 2e). Hence, this process can be seen as a step-wise refinement of the constraint defining the space [5,11]. How the problem space is refined in each step is called a *decision*.

A *configurator* is a tool that displays the model capturing the constraints, presents the user with possible (manual) decisions, and infers necessary (automatic) decisions. The tool discourages the user from making decisions inconsistent with the constraints, and, it suggests decisions that are necessary to satisfy the constraints. For convenience, configurators enable the user to *retract* previously made decisions; some even enable to temporarily violate the constraints but this is out of the scope of this article and gradual refinement will be assumed.

For illustration consider the feature model in Fig. 2b and the following steps (see Fig. 2d). (1) The user selects the feature a. This implicitly deselects b as a and b are in an xor-group and there are no feature configurations with both a and b. (2) The user selects the feature c, which has no further effects. (3) Finally, he deselects the feature d and the process is completed since exactly one feature combination is left, i.e., each feature is either in the product or not.

In summary, the input to the configurator is a constraint defining a large set of possibilities—the outermost ellipse in Fig. 2e. Gradually, this set is shrunk until exactly one possibility is left (assuming the user succeeded).

2.2 Completing a Configuration Process and the Shopping Principle

The concepts introduced in the previous sections are well known [8]. However, the configuration literature does not study the completion of a configuration process. As stated above, at the end of the process the decisions must determine exactly one feature combination (the innermost ellipse in Fig. 2e). So, how does the user achieve this? This article introduces the following classification.

M *(Manual).* The user makes decisions up to the point when all considered variables have been bound, i.e., each variable has been assigned a value by the user or by a decision inferred by the configurator. The disadvantage of this approach is that the user needs to fill in every single detail, which is cumbersome especially if there are some parts of the problem that are not of a high relevance to the user. The only assistance the tool provides is the mechanism that infers new decisions or disables some decision. We will not discuss this case further.

A *(Full blind automation).* A function automatically computes *some* values for all variables that do not have a value yet. The disadvantage of this option is that it takes all the control from the user as it is essentially making decisions for him.

A$^+$ *(Smart automation).* If we assume some form of an a priori, common sense assumptions, there is an approach somewhere between the options **M** and **A**. In the example above, the user had to *explicitly* deselect the optional feature d but would it be possible to instead say that all features not selected should be deselected? The motivation for this approach can be explained by an analogy with shopping for groceries (thus the *shopping principle*). The customer asks only for those items that he wants rather than saying for each item in the store whether he wants it or not. If some variables cannot be bound according to this principle, due to some dependencies, the tool will highlight them since it is possible that the user forgot to make a certain decision, e.g., we wouldn't want the tool to decide between features in an xor-group (a and b).

In some sense, the scenario $\mathbf{A}^+$ is a more careful version of scenario **A**. Both support the user to complete the configuration process. Scenario **A** binds **all** variables, whereas $\mathbf{A}^+$ only those for which this doesn't mean deciding something for the user. The following section investigates these scenarios in constraints defined as Boolean formulæ (recall that FODA produces Boolean formulæ).

3 Propositional Configuration

First, let us recall some basic terms from propositional logic. Let $\mathcal{V}$ be some finite set of variables. The propositional formulæ discussed from now on will be only on these variables. A *variable assignment* assigns either *true* or *false* to each considered variable. Often it is useful to think of a variable assignment as the set of variables that are assigned the value *true*, e.g., the variable assignment $x \mapsto \mathit{true}, y \mapsto \mathit{false}$ corresponds to the set $\{x\}$ in the set-based notation.

A *model* of a formula ϕ is such a variable assignment under which ϕ evaluates to *true*. We say that the formula ϕ is *satisfiable*, denoted as $\textsc{Sat}(\phi)$, if and only if ϕ has at least one model, e.g., the formula $x \vee y$ is satisfiable whereas the formula $x \wedge \neg x$ is not. We write $\phi \models \psi$ to denote that the formula ψ evaluates to *true* under all models of ϕ, e.g., it holds that $x \wedge y \models x$.

3.1 Propositional Configuration Process

In order to reason about the feature model and the user's requirements, the configurator translates them in some form of mathematical representation. In this section we assume that the model has already been translated into propositional logic (see Fig. 2c for illustration).

Definition 1. *A* propositional configuration process *for some finite set of variables $\mathcal{V}$ and a satisfiable propositional formula ϕ only on the variables from $\mathcal{V}$ is a sequence of propositional formulæ $\phi_0, \ldots, \phi_k$ such that $\phi_0 \stackrel{def}{=} \phi$, $\phi_{i+1} \stackrel{def}{=} \phi_i \wedge \xi_i$*

for all $i \in 0 \ldots k-1$*, and* ϕ_k *is satisfied by one variable assignment. The formulæ* ξ_i *are decisions made by the user or decisions inferred by the configurator. If* ξ_i *is of the form* v *for some variable* $v \in \mathcal{V}$*, then we say that the variable* v *has been* assigned the value *true in step* i*; if* ξ_i *is of the form* $\neg v$*, we say that it has been* assigned the value *false in step* i*. Observe that* $\phi_{i+1} \Rightarrow \phi_i$ *for all* $i \in 0 \ldots k-1$*, i.e., the set of models is shrunk along the process.*

Example 1. Let $\phi_0 \stackrel{\text{def}}{=} (\neg u \vee \neg v) \wedge (x \Rightarrow y)$. The user sets u to *true* ($\phi_1 \stackrel{\text{def}}{=} \phi_0 \wedge u$); the configurator sets v to *false* as u and v are mutually exclusive ($\phi_2 \stackrel{\text{def}}{=} \phi_1 \wedge \neg v$). The user sets y to *false* ($\phi_3 \stackrel{\text{def}}{=} \phi_2 \wedge \neg y$); the configurator sets x to *false* ($\phi_4 \stackrel{\text{def}}{=} \phi_3 \wedge \neg x$). The process is finished as all variables were assigned a value.

The inference mechanism of the configurator typically inspects for all variables $v \in \mathcal{V}$ whether $\phi_l \models v$, in which case it sets v to *true*, and whether $\phi_l \models \neg v$, in which case it sets v to *false*. If a value has been inferred, the user is discouraged by the user interface to change it ("graying out"). This can be computed with the help of a SAT solver [9] or Binary Decision Diagrams (BDDs) [8].

3.2 Completing a Propositional Configuration Process

Let us look at the scenarios for completing a propositional configuration process. Earlier, we have identified two types of functions that the user may invoke at any step of the process: (**A**) a function that binds all the remaining variables; ($\mathbf{A}^+$) a function that finds values for only some variables according to an a priori knowledge; we call this function a *shopping principle function*.

The case **A** is straightforward, finding a solution to the formula ϕ_i in step i is a satisfiability problem which can be solved by a call to a SAT solver or by a traversal of a BDD corresponding to ϕ_i (see [9,8] for further references).

The scenario $\mathbf{A}^+$, however, is more intriguing. The a priori knowledge that we apply is the shopping principle (see Sect. 2.2), i.e., what has not been selected should be *false*. According to our experience and intuition (as well as other researchers [1, Section 4.2]), this is well in accord with human reasoning: the user has the impression that if a variable (a feature in a feature model) has not been selected, then it should be deselected once the process is over.

However, it is not possible to set all unassigned variables to *false* in all cases. For instance, in $u \vee v$ we cannot set both u and v to *false*—the user must choose which one should be *true*, and we do not want to make such decision for him (otherwise we would be in scenario **A**). Another way to see the problem is that setting u to *false* will *force* the variable v to be *true*, and vice-versa. If we consider the formula $x \Rightarrow (y \vee z)$, however, all the variables can be set to *false* at once and no further input from the user is necessary.

In summary, the objective is to maximize the set of variables that can be set to *false* without making any decisions for the user, i.e., variables that can be deselected safely. Upon a request, the configurator will set the safely-deselectable variables to *false* and highlight the rest as they need attention from the user.

3.3 Deselecting Safely

We start with an auxiliary definition that identifies sets of variables that can be deselected at once. This definition enables us to specify those variables that can be deselected safely; we call such variables dispensable variables (we kindly ask the reader to distinguish the terms deselectable and dispensable).

Definition 2 (Deselectable). *A set of variables $X \subseteq \mathcal{V}$ is* deselectable *w.r.t. the formula ψ, denoted as $\mathcal{D}(\psi, X)$, iff all variables in X can be set to false at once. Formally defined as $\mathcal{D}(\psi, X) \stackrel{def}{=} \textsc{Sat}\left(\psi \wedge \bigwedge_{v \in X} \neg v\right)$. Analogously, a single variable $v \in \mathcal{V}$ is* deselectable *w.r.t. the formula ψ iff it is a member of some deselectable set of variables, i.e., $\mathcal{D}(\psi, v) \stackrel{def}{=} \textsc{Sat}(\psi \wedge \neg v)$.*

Definition 3 (Dispensable variables). *A variable $v \in \mathcal{V}$ is* dispensable *w.r.t. a formula ψ iff the following holds: $(\forall X \subseteq \mathcal{V})\,(\mathcal{D}(\psi, X) \Rightarrow \mathcal{D}(\psi \wedge \neg v, X))$.*

In plain English, a variable v is dispensable iff any deselectable set of variables X remains deselectable after v has been deselected (set to *false*). Intuitively, the deselection of v does not force selection of anything else, which follows the motivation that we will not be making decisions for the user. In light of the shopping principle (see Sect. 2.2), a customer can skip a grocery item only if skipping it does not require him to obtain some other items. The following examples illustrate the two definitions above.

Example 2. Let $\phi \stackrel{\text{def}}{=} (u \vee v) \wedge (x \Rightarrow y)$. Each of the variables is deselectable but only x and y are dispensable. The set $\{x, y\}$ is deselectable, while the set $\{u, v\}$ is not. The variable u is not dispensable as $\{v\}$ ceases to be deselectable when u is set to *false*; analogously for v. The variable x is dispensable since after x has been deselected, y can still be deselected and the variables u, v are independent of x's value. Analogously, if y is deselected, x can be deselected.

Observe that we treat *true* and *false* asymmetrically, deselecting y forces x to *false*, which doesn't collide with dispensability; deselecting u forces v to *true* and therefore u is *not* dispensable.

Example 3. Let ϕ be defined as in the previous example and the user is performing configuration on it. The user invokes the shopping principle function. As x and y are dispensable, both are deselected (set to *false*). The variables u and v are highlighted as they need attention. The user selects u, which results in the formula $\phi_1 \stackrel{\text{def}}{=} \phi \wedge u$. The variable v becomes dispensable and can be deselected automatically. The configuration process is finished as all variables have a value.

As we have established the term dispensable variable, we continue by studying its properties in order to be able to compute the set of dispensable variables and to gain more intuition about them.

Lemma 1. *Let Υ_ϕ denote the set of all dispensable variables of ϕ. For a satisfiable ψ, the dispensable variables can be deselected all at once, i.e., $\mathcal{D}(\psi, \Upsilon_\phi)$.*

Proof. By induction on the cardinality of subsets of Υ_ψ. Let $\Upsilon_0 \stackrel{\text{def}}{=} \emptyset$, then $\mathcal{D}(\psi, \Upsilon_0)$ as ψ is satisfiable. Let $\Upsilon_i, \Upsilon_{i+1} \subseteq \Upsilon_\psi$ s.t. $\Upsilon_{i+1} = \Upsilon_i \cup \{x\}$, $|\Upsilon_i| = i$, and $|\Upsilon_{i+1}| = i+1$. Since x is dispensable and $\mathcal{D}(\phi, \Upsilon_i)$, then $\mathcal{D}(\phi \wedge \neg x, \Upsilon_i)$, which is equivalent to $\mathcal{D}(\phi, \Upsilon_i \cup \{x\})$.

The following lemma reinforces that the definition of dispensable variables adheres to the principles we set out for it, i.e., it maximizes the number of deselected variables while not arbitrarily deciding between variables.

Lemma 2. *The set Υ_ϕ—the set of all dispensable variables of ϕ—is the intersection of all maximal sets of deselectable variables of ϕ.*

Proof (sketch). From definition of dispensability, any deselectable set remains deselectable after any dispensable variable is added to it, hence Υ_ϕ is a subset of any maximal deselectable set. Υ_ϕ is a maximal set with this property because for each deselectable set that contains at least one non-dispensable variable there is another deselectable set that does not contain this variable.

3.4 Dispensable Variables and Non-monotonic Reasoning

After defining the shopping principle in mathematical terms, the authors of this article realized that dispensable variables correspond to certain concepts from Artificial Intelligence as shown in this subsection.

The *Closed World Assumption (CWA)* is a term from logic programming and knowledge representation. Any inference that takes place builds on the assumption that if something has not been said to be *true* in a knowledge base, then it should be assumed *false*. Such reasoning is called *non-monotonic* as an increase in knowledge does not necessarily mean an increase in inferred facts. In terms of mathematical logic, CWA means adding negations of variables that should be assumed *false* in the reasoning process. Note that not all negateable variables ($\phi \nvDash v$) can be negated, e.g., for the formula $x \vee y$ both x and y are negateable but negating both of them would be inconsistent with the formula.

The literature offers several definitions of reasoning under Closed World Assumption [3,7]. A definition relevant to this article is the one of the *Generalized Closed World Assumption (GCWA)* introduced by Minker [19] (see [3, Def. 1]).

Definition 4. *The variable v is* free of negation *in the formula ϕ iff for any positive clause B for which $\phi \nvDash B$, it holds that $\phi \nvDash v \vee B$. The* closure $C(\phi)$ *of a formula ϕ is defined as $C(\phi) \stackrel{\text{def}}{=} \phi \cup \{\neg K \mid K$ is free for negation in $\phi\}$.*

It is not difficult to see that dispensable variables are those that are free of negation as shown by the following lemma.

Lemma 3. *Dispensable variables coincide with those that are free of negation.*

Proof. Observe that $\phi \nvDash \psi$ iff $\textsc{Sat}(\phi \wedge \neg\psi)$, then the definition above can be rewritten as: For $B' \stackrel{\text{def}}{=} \bigwedge_{v \in V_B} \neg v$ for some set of variables V_B for which $\textsc{Sat}(\phi \wedge B')$, it holds that $\textsc{Sat}(\phi \wedge \neg v \wedge B')$. According to the definition of $\mathcal{D}$, this is equivalent to $\mathcal{D}(\phi, V_B) \Rightarrow \mathcal{D}(\phi \wedge \neg v, V_B)$ (compare to Definition 3).

Table 1: Experimental Results

Name	Features	Clauses	Length	Done	Minimal models
tightvnc	21	22	5.5	5.5	1.0 ± 0.0
apl	27	41	12.2	11.9	1.0 ± 0.0
gg4	58	139	10.0	3.8	15.3 ± 22.6
berkeley	94	183	26.6	17.9	1.7 ± 1.1
violet	170	341	56.1	47.1	1.6 ± 0.9

Circumscription, in our case the *propositional circumscription*, is another important form of reasoning [18]. A circumscription of a propositional formula ϕ is a set of minimal models of ϕ. Where a model α of a formula ϕ is *minimal* iff ϕ has no model α' which would be a strict subset of α, e.g., the formula $x \vee y$ has the models $\{x\}, \{y\}, \{x, y\}$ where only $\{x\}$ and $\{y\}$ are minimal. We write $\phi \models_{\min} \psi$ to denote that ψ holds in all minimal models of ϕ, e.g., $x \vee y \models_{\min} \neg(x \wedge y)$.

The following lemma relates minimal models to dispensable variables (The proof of equivalence between minimal models and GCWA is found in [19]).

Lemma 4. *A variable v is dispensable iff it is false in all minimal models.*

Example 4. Let $\phi_0 \stackrel{\text{def}}{=} (u \vee v) \wedge (x \Rightarrow y)$. The minimal models of the formula ϕ_0 are $\{u\}$, $\{v\}$, hence $\phi_0 \models_{\min} \neg x$ and $\phi_0 \models_{\min} \neg y$. Then, if the user invokes the shopping principle function, x and y are deselected, i.e., $\phi_1 \stackrel{\text{def}}{=} \phi_0 \wedge \neg x \wedge \neg y$. And, the user is asked to resolve the competition between $u \vee v$, he selects u, resulting in the formula $\phi_2 \stackrel{\text{def}}{=} \phi_1 \wedge u$ with the models $\{u\}$ and $\{u, v\}$ where only the model $\{u\}$ is minimal hence v is set to *false* as dispensable. The configuration process is complete because u has the value *true* and the rest are dispensable.

3.5 Experimental Results

The previous section shows that dispensable variables can be found by enumerating minimal models. Since the circumscription problem is Π_2^P-complete [7] it is important to check if the computation is feasible in practice. We applied a simple evaluation procedure to five feature models[4]: For each feature model we simulated 1000 random manual configuration processes (scenario **M**). At each step we enumerated minimal models. (Algorithmic details can be found online [12].) We also counted how many times there was exactly one minimal model: At those steps the configuration process would have been completed if the user invoked the shopping principle function.

The results appear in Table 1. The column **Length** represents the number of user decisions required if the shopping principle function is not invoked; the column **Done** represents in how many steps an invocation of the shopping principle function completes the configuration; the column **Minimal models** shows that the exponential worst case tends not to occur in practice and therefore enumeration of all minimal models is feasible.

[4] from http://fm.gsdlab.org/index.php?title=Model:SampleFeatureModels

4 Beyond Boolean Constraints

The previous section investigated how to help a user with configuring propositional constraints. Motivated by the shopping principle, we were trying to set as many variables to *false* as possible. This can be alternatively seen as that the user **prefers** the undecided variables to be *false*.

This perspective helps us to generalize our approach to the case of non-propositional constraints under the assumption that there is some notion of preference between the solutions. First, let us establish the principles for preference that are assumed for this section. (1) It is a partial order on the set in question. (2) It is static in the sense that all users of the system agree on it, e.g., it is better to be healthy and rich than sick and poor. (3) If two elements are incomparable according to the ordering, the automated support shall not decide between them, instead the user shall be prompted to resolve it.

To be able to discuss these concepts precisely, we define them in mathematical terms. We start by a general definition of the problem to be configured, i.e., the initial input to the configurator, corresponding to the set of possibilities that the user can potentially reach—the outermost ellipse in Fig. 2e.

Definition 5 (Solution Domain). *A* Solution Domain (SD) *is a triple* $\langle \mathcal{V}, \mathcal{D}, \phi \rangle$ *where* $\mathcal{V}$ *is a set of variables* $\mathcal{V} = \{v_1, \ldots, v_n\}$*,* $\mathcal{D}$ *is a set of respective domains* $\mathcal{D} = \{D_1, \ldots, D_n\}$*, and the constraint* $\phi \subseteq D_1 \times \cdots \times D_n$ *is an n-ary relation on the domains (typically defined in terms of variables from* $\mathcal{V}$*).*

A variable assignment *is an n-tuple* $\langle c_1, \ldots, c_n \rangle$ *from the Cartesian product* $D_1 \times \cdots \times D_n$*, where the constant* c_i *determines the value of the variable* v_i *for* $i \in 1 \ldots n$*. For a constraint* ψ*, a variable assignment* α *is a* solution *iff it satisfies the constraint, i.e.,* $\alpha \in \psi$*.*

An Ordered Solution Domain (OSD) *is a quadruple* $\langle \mathcal{V}, \mathcal{D}, \phi, \prec \rangle$ *where* $\langle \mathcal{V}, \mathcal{D}, \phi \rangle$ *is an SD and* $\prec$ *is a partial order on* $D_1 \times \cdots \times D_n$*. For a constraint* ψ*, a solution* α *is* optimal *iff there is no solution* α' *of* ψ *s.t.* $\alpha' \neq \alpha$ *and* $\alpha' \prec \alpha$*.*

Recall that the user starts with a large set of potential solutions, gradually discards the undesired ones until only one solution is left. From a formal perspective, solution-discarding is carried out by strengthening the considered constraint, most typically by assigning a fixed value to some variable.

Definition 6 (Configuration Process). *Given a Solution Domain* $\langle \mathcal{V}, \mathcal{D}, \phi \rangle$*, an* interactive configuration process *is a sequence of constraints* $\phi_0, \ldots, \phi_k$ *such that* $\phi_0 \stackrel{def}{=} \phi$ *and* $|\phi_k| = 1$*. The constraint* ϕ_{j+1} *is defined as* $\phi_{j+1} \stackrel{def}{=} \phi_j \cap \xi_j$ *where the constraint* ξ_j *represents the decision in step* j *for* $j \in 0 \ldots k-1$*. If* ξ_j *is of the form* $v_i = c$ *for a variable* v_i *and a constant* $c \in D_i$*, we say that the variable* v_i *has been assigned the value* c *in step* j*. Observe that* $\phi_{j+1} \subseteq \phi_j$ *for* $j \in 0 \ldots k-1$ *and* $\phi_j \subseteq \phi$ *for* $j \in 0 \ldots k$*.*

A configurator in this process disables certain values or assigns them automatically. In particular, the configurator disallows selecting those values that are not part of any solution of the current constraint, i.e., in step l it disables all values $c \in D_i$ of the variable v_i for which there is no solution of the constraint ϕ_l of the form $\langle c_1, \ldots, c, \ldots c_n \rangle$. If all values but one are disabled for the domain D_i, then the configurator automatically assigns this value to the variable v_i.

Now as we have established the concept for general configuration, let us assume that a user is configuring an Ordered Solution Domain (Definition 5) and we wish to help him with configuring variables that have lesser importance for him, similarly as we did with the shopping principle. The configuration proceeds as normal except that after the user configured those values he wanted, he invokes a function that tries to automatically configure the unbound variables using the given preference.

The assumption we make here is that the variables that were not given a value yet should be configured such that the result is optimal while preserving the constraints given by the user so far. Since the preference relation is a partial order, there may be multiple optimal solutions. As we do not want to make a choice for the user, we let him focus only on optimal solutions.

If non-optimal solutions shall be ruled out, the configurator identifies such values that never appear in any optimal solution to reduce the number of decisions that the user must focus on. Dually, the configurator identifies values that appear in all optimal solutions, the following definitions establish these concepts.

Definition 7 (Settled variables.). *For a constraint ψ and a variable v_i, the value $c \in D_i$ is* non-optimal *iff the variable v_i has the value c only in* non-optimal *solutions of ψ (or, v_i has a different value from c in all optimal solutions of ψ). A value c is* settled *iff v_i has the value c in all optimal solutions of ψ. A variable v_i is settled if there is a settled value of v_i.*

Observation 1 *For some constraint and the variable v_i, a value $c \in D_i$ is settled iff all values $c' \in D_i$ different from c are non-optimal.*

Example 5. Let $x, y, z \in \{0, 1\}$. Consider a constraint requiring that at least one of x,y,z is set to 1 (is selected). The preference relation expresses that we prefer lighter and cheaper solutions where x, y, and z contribute to the total weight by 1, 2, 3 and to the total price by 10, 5, and 20, respectively. Hence, the solutions satisfy $(x + y + z > 0)$, and $\langle x_1, y_1, z_1 \rangle \prec \langle x_2, y_2, z_2 \rangle$ iff $(10x_1 + 5y_1 + 20z_1 \leq 10x_2 + 5y_2 + 20z_2) \wedge (1x_1 + 2y_1 + 3z_1 \leq 1x_2 + 2y_2 + 3z_2)$. Any solution setting z to 1 is non-optimal as z is more expensive and heavier than both x and y, and hence the configurator sets z to 0 (it is settled). Choosing between x and y, however, needs to be left up to the user because x is lighter than y but more expensive than y.

Propositional configuration, studied in the previous section, is a special case of a Solution Domain configuration with the variable domains $\{true, false\}$. The following observation relates settled and dispensable variables (definitions 7, 3).

Observation 2 *For a Boolean formula understood as an OSD with the preference relation as the subset relation, a variable is settled iff it is dispensable or it is true in all models (solutions). Additionally, if each variable is settled, the invocation of the shopping principle function completes the configuration process.*

This final observation is an answer to the question in the title, i.e., configuration may be completed when all variables are settled. And, according to our experiments this happens frequently in practice (column **Done** in Table 1).

5 Related Work

Interactive configuration as understood in this article has been studied e.g., by Hadžić et al. [8], Batory [1], and Janota [9]. In an analogous approach Janota et al. [11] discuss the use of interactive configuration for feature model construction. The work of van der Meer et al. [24] is along the same lines but for unbounded spaces. Lottaz et al. [17] focus on configuration of non-discrete domains in civil engineering.

There is a large body of research on *product configuration* (see [22] for an overview), which typically is conceptualized rather as a programming paradigm than a human-interaction problem. Moreover, the notion rules are used instead of formulæ. Similarly as do we, Junker [13] applies preference in this context. We should note that preference in logic has been studied extensively, see [6].

The problem how to help the user to finish the configuration process was studied by Krebs et al. [16] who applied machine learning to identify a certain plan in the decisions of the user.

Circumscription has been studied extensively since the 80's [18,19,7]. Calculation of propositional circumscription was studied by Reiter and Kleer [21]; calculation of all minimal models by Kavvadias et al. and work referenced therein [15].

6 Summary

This article proposes a novel extension for configurators—the shopping principle function (Sect. 3.2). This function automates part of the decision-making but is not trying to be *too* smart: it does not make decisions between equally plausible options. The article mainly focuses on the propositional case, as software engineering models' semantics are typically propositional. The relation with GCWA, known from Artificial Intelligence, offers ways how to compute the shopping principle function (Sect. 3.4). Several experiments were carried out suggesting that the use of the shopping principle function is feasible and useful (Sect. 3.5). The general, non-propositional, case is studied at a conceptual level opening doors to further research (Sect. 4). The authors are planning to integrate this function into a configurator and carry out further experiments as future work.

Acknowledgment This work is partially supported by Science Foundation Ireland under grant no. 03/CE2/I303_1 and the IST-2005-015905 MOBIUS project. The authors thank Don Batory and Fintan Farmichael for valuable feedback.

References

1. D. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, 2005.

2. D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *Advanced Information Systems Engineering (CAiSE)*, 2005.
3. M. Cadoli and M. Lenzerini. The complexity of closed world reasoning and circumscription. In *The Eighth National Conference on Artificial Intelligence*, 1990.
4. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
5. K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *SPLC*, 2004.
6. J. Delgrande, T. Schaub, H. Tompits, and K. Wang. A classification and survey of preference handling approaches in nonmonotonic reasoning. *Computational Intelligence*, 20(2):308–334, 2004.
7. T. Eiter and G. Gottlob. Propositional circumscription and extended closed world reasoning are Π_2^P-complete. *Theoretical Computer Science*, 1993.
8. T. Hadzic, S. Subbarayan, R. Jensen, H. Andersen, J. Møller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *The International Conference on Economic, Technical and Organizational aspects of Product Configuration Systems, DTU*, 2004.
9. M. Janota. Do SAT solvers make good configurators? In *First Workshop on Analyses of Software Product Lines (ASPL)*, 2008.
10. M. Janota and J. Kiniry. Reasoning about feature models in high-order logic. In *SPLC*, 2007.
11. M. Janota, V. Kuzina, and A. Wasowski. Model construction with external constraints: An interactive journey from semantics to syntax. In *MODELS*, 2008.
12. M. Janota, J. Marques-Silva, and R. Grigore. Algorithms for finding dispensable variables. http://arXiv.org/abs/0910.0013.
13. U. Junker. Preference programming for configuration. In *Workshop on Configuration*, 2001.
14. 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, SEI, Carnegie Mellon University, 1990.
15. D. J. Kavvadias, M. Sideri, and E. C. Stavropoulos. Generating all maximal models of a Boolean expression. *Information Processing Letters*, 2000.
16. T. Krebs, T. Wagner, and W. Runte. Recognizing user intentions in incremental configuration processes. In *Workshop on Configuration*, 2003.
17. C. Lottaz, R. Stalker, and I. Smith. Constraint solving and preference activation for interactive design. *AI EDAM*, 12(01):13–27, 1998.
18. J. McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.
19. J. Minker. On indefinite databases and the Closed World Assumption. In *Proceedings of the 6th Conference on Automated Deduction*. Springer-Verlag, 1982.
20. D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 1976.
21. R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems: Preliminary report. In *Proceedings of AAAI*, 1987.
22. D. Sabin and R. Weigel. Product configuration frameworks-a survey. *IEEE Intelligent Systems*, 13(4):42–49, 1998.
23. P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Requirements Engineering Conference (RE)*, 2006.
24. E. R. van der Meer, A. Wasowski, and H. R. Andersen. Efficient interactive configuration of unbounded modular systems. In *Symp. Applied Comput. (SAC)*, 2006.