# Evolution of Configuration Models – a Focus on Correctness

**Thorsten Krebs**[1]

**Abstract.** Structure-based configuration models describe commonality and variability as well as restrictions within and between components of a product domain. Innovation in the development of products drives the development of new components and adaptation of existing components. As a consequence of evolution, the configuration model has to evolve in parallel with its referants in the world. A side effect of using configuration models for describing different and evolving products in one model is an increasing possibility for errors. Keeping an overview of the hundreds or thousands of configurable components and the increasing number of interrelations and restrictions for combinations of those is hard, if not impossible, without tool support. This paper focuses on how to keep a configuration model correct despite its changes. Change operations implement changes to the configuration model. After applying changes, syntactical correctness of the configuration model is checked with pre-defined invariants. A three-step process is introduced that consists of compiling change operations, propagating changes and validating the model after change propagation.

## 1 Introduction

Configuration tools are widely used to assemble new products out of a predefined set of existing components. The *product line* approach helps to configure similar but different products that together form a *product family*. These product family members are achieved by different compositions of the product components, with respect to technical possibilities and given customer requirements. In structure-based configuration, *configuration models* are used, which contain a textual description of the components, their capabilities, properties and relations to other components. Thus, configuration models describe commonality and variability as well as restrictions within and between components of a product domain. With this, all potentially configurable product family members are implicitly represented.

Reusability is widely identified as a key to improving (software) development productivity and quality. The reuse of (software) components results in fewer developments for a new system and in less time spent on development [2]. With reuse alone, however, innovation is not considered. New components are developed and existing components are evolved in order to derive new products. A consequence of evolution is that the configuration model has to be evolved in parallel with its referants in the world. Only then configuration tools can use the new knowledge. In such a dynamic environment the domain knowledge evolves continually [6].

Configuration models are input for configuration tools that offer automated support for correct and consistent products. In order to achieve this, configuration models have to be correct and consistent representations of the product domain. A side effect of continual evolution is an increasing possibility for errors because of the hundreds or thousands of configurable components in that model and the increasing number of interrelations and restrictions for combinations of those [19]. It is apparent that tool support can improve the evolution of configuration models. Modelling environments for creating, evolving and diagnosing configuration models have been addressed in recent years, but no one-fits-all solution is available.

This paper sketches the core ideas of a PhD thesis, which focuses on ensuring correctness of configuration models while applying changes. This paper describes work in progress and discusses goals reached so far as well as also future goals.

The remainder of this paper is organized as follows. Section 2 compares similar topics to see if existing ideas for modeling and reasoning about evolution of configuration models can be valuable. Section 3 introduces basic notions of evolution such as correctness of configuration models and change operations that implement the applicable changes. Section 4 details the application of change operations and validation of correctness. Section 5 discusses related work and Section 6 presents relevant topics for future work. Finally, Section 7 concludes this paper.

## 2 Lessons Learned

Looking beyond one's own nose, there are a number of related topics. This section takes a look at them to see what can be learned and used for evolution of structure-based configuration models.

### 2.1 Ontologies

*Ontologies* have gained popularity within the knowledge engineering community. Research in the area of ontology evolution quickens the pace as the semantic web gains interest. Generally, ontologies provide a "shared and common" understanding of a domain and facilitate "knowledge sharing and reuse" [5]. An ontology is an explicit specification of a *conceptualization* of the objects and other entities that are assumed to exist in some area of interest and the relationships that hold among them [7].

The shared and common understanding of an ontology is represented in a domain-independent vocabulary. Typically, frame-based languages are used to model ontologies. The central modeling primitives are concepts (known as frames) with properties (known as slots). Frames provide a context for modeling concepts in a taxonomy with slot-value pairs used to specify attributes of the concepts. Slots are often treated as objects that can be arranged in a hierarchy themselves.

[1] LKI, University of Hamburg, Germany, email: krebs@informatik.uni-hamburg.de

Configuration models use considerably more formalisms to represent product structures:

- compositional relations build a partonomy in which parts are related to aggregates,
- a cardinality specifies the amount of potential instances for compositional relations,
- there are optional and alternative concept definitions, and
- constraints represent restrictions between concepts and concepts properties.

Nonetheless, similar representation formalisms are used to represent objects in a taxonomic hierarchy. Some basic changes, like adding and deleting concepts or adding, deleting and modifying properties of concepts exist in both approaches. Hence, research on ontology evolution is considered to be of interest also for evolution of configuration models.

## 2.2 Knowledge Representation

A lot of effort has been put into formalizing configuration models over the last decades. Alternative formalisms for modeling product structures exist. This is apparent since a model is a *representation*: it imitates, resembles or stands for things that exist in the world [4, 15]. Most representation formalisms have a common basis. A configuration model uniquely identifies the components of a system, their properties, structure and possible variability. Modeling facilities of structure-based configuration models are:

**Concepts** represent products, components and other entities of the domain. Each concept $c \in \mathcal{C}$ carries a name which makes it uniquely identifyable within a domain. A concept may specify an arbitrary number of attributes. An attribute is a binary tuple that consists of a uniquely identifyable name within the specialization relation, and a value. The value of an attribute is restricted to a set of predefined value domains like integers, floats, strings, intervals of integers or floats and sets of all three. The set of attributes of a concept $c$ is denoted with $\mathcal{A}_c$.

**Specialization relations** relate a concept $c_1$ to its subconcepts $c_2$ and with this form a taxonomic hierarchy $(c_1, c_2) \in \mathcal{H}$. Every concept has exactly one ancestor and can have an arbitrary number of descendants. Multiple inheritance is explicitly ruled out with the definition of a tree structure. The specialization relation is transitive. Therefore, the superconcept of a superconcept of a concept $c$ is an *indirect* superconcept of $c$. The set of relations in which concept $c$ is the aggregate is denoted with $\mathcal{R}_c$.

**Compositional relations** relate a part $p$ to its aggregate $a$ and with this form a partonomy $(a, p) \in \mathcal{P}$. Objects are either *primitive* or *composite*, i.e. they reside at the leaves of a component hierarchy or are the root of a subgraph, respectively. One aggregate can have multiple parts in compositional relations.

**Constraints** express interdependencies and restrictions between concept definitions and their properties such as incompatibilities, attribute values that depend on other attributes (of other concepts), and so on. There are local and global constraints $L \cup G = \Gamma$ . *Local constraints* are applied to single attributes, a single concept or to a relation between two concepts. *Global constraints* involve a larger number of concepts and attributes or relations.

Usually, this is a static structure that does not take versioning into account. It can be represented by a graph whose nodes and edges correspond to concepts and relationships, respectively [3]. For modeling product families (i.e. modeling a set of products within one model), this static, non-variable representation is enriched with notions to describe variability; like optional and alternative definitions.

Evolution of knowledge strutcures that are represented by concepts and relations is a well-understood domain. Usually, evolution is divided into historical and logical versioning – i.e. a two-dimensional representation in time and space, respectively. Typically, *versions* are characterized as "descendants of some existing version, if not the first one, and can serve as an ancestor for multiple versions" [8].

There are numerous problems within the domain of evolution and versioning. However, this paper focuses on correctness of configuration models and considers versions of models more appropriate. A change transforms a configuration model $\mathcal{M}$ into a new configuration model $\mathcal{M}'$. A *valid* change $c$, in addition, is a function that transforms a correct configuration model into another correct configuration model $c : \mathcal{M} \mapsto \mathcal{M}'$. Changed configuration models are different from each other but still represent the same underlying product domain. They are different versions of the same model. A version captures a specific point in time. This means that different versions describe different sets of configurable products – according to the existing configurable components at that time.

## 2.3 Configuration Management

*Configuration management* (CM) systems are concerned with managing the evolution of large and complex systems represented in an explicit, unambiguous configuration model [23, 24]. CM serves management support (i.e. controlling changes to products) as well as development support (i.e. providing functions which assist developers in performing coordinated changes to products) [3].

Basic requirements for configuration management are version control (keeping track of changes to components, supporting parallel development and enabling branching and merging), build management (the process of building components and producing a "bill-of-materials"), and process control (a set of policies, including monitoring changes, notification on changes, access control and reporting) [14].

Typically, CM systems distinguish between the *product space* and the *version space*. The product space consists of the configurable objects and their relationships, while the version space organizes the versions of these objects. Versions are organized into a derivation history [8]. Key features of versioning are the organization of the version space, the interrelations between product space and version space, and operations for retrieving existing versions and constructing new ones.

Traditional configuration models capture the versioning in space. This means they represent all admissible configurations. Evolution, i.e. versioning in space and time, is typically not supported [11]. Methods from CM can help at this point.

# 3 Evolution

One of the key benefits of configuration mechanisms is to guarantee consistency and correctness of configured products. To reach this goal, the configuration model has to be consistent and correct. In a dynamic environment with continual changes to the configuration model it is apparent that a major goal of the evolution process is keeping the model consistent and correct despite its changes.

*Consistency* means that none of the facts deducible from a configuration model contradict one another. Thus, consistency can be considered as an agreement among the knowledge entities with respect to

the semantic of the underlying modeling language [20]. *Correctness* of a configuration model is asserted when it is correct with respect to the underlying modeling language.

Therefore, two levels of analyzing configuration models can be distinguished: a *syntactic* and a *semantic* level. This section deals with the syntactic analysis of configuration models. As a first step towards an evolution process that keeps configuration models consistent and correct, the following subsections discuss *well-formed* (i.e. syntactically correct) configuration models and introduce invariants to check the model against.

## 3.1 Correctness

A configuration model $\mathcal{M}$ is *well-formed* with respect to a set of syntax invariants $\mathcal{I}$ if for all $i \in \mathcal{I}$, $\mathcal{M}$ satisfies the invariant $i(\mathcal{M})$.

*Invariants* are conditions that must hold for every configuration model [1]. Every change applied to a configuration model must maintain the correctness defined by the invariants. Three syntax invariants that are needed to understand the upcoming example are introduced in the following.

**I1 - Specialization Relation Invariant** A concept has exactly one superconcept; if not the root concept.

$$\forall c_1 \in \mathcal{C} \setminus \{\text{root}\}\ \exists c_2 \in \mathcal{C} \text{ such that } (c_2, c_1) \in \mathcal{H}$$
$$\text{and } \forall c_3 \in \mathcal{C} \text{ with } (c_3, c_1) \in \mathcal{H} \Rightarrow c_2 = c_3$$

**I2 - Inheritance Invariant** The value of every attribute $a$ of a concept $c_1$ that is inherited from $c_2$ has to be a subset of the corresponding value of $a'$ of $c_2$.

$$\forall c_1, c_2 \in \mathcal{C} \text{ with } (c_2, c_1) \in \mathcal{H}, \forall a' \in \mathcal{A}_{c_2}\ \exists a \in \mathcal{A}_{c_1}$$
$$\text{such that name}(a) = \text{name}(a') \wedge \text{value}(a) \subseteq \text{value}(a')$$

**I3 - Composition Reference Invariant** All parts that are referenced in a compositional relation have to be defined as concepts.

$$\forall (a, p) \in \mathcal{P}\ \exists c \in \mathcal{C} \text{ such that name}(p) = \text{name}(c)$$

Of course, these are just simple examples of invariants. A lot more invariants need to be defined for guaranteeing well-formed configuration models.

## 3.2 Example

A PC consists, among others, of a VGA card. Two types of VGA cards are shown in Figure 1, AGP cards and the new type, PCIe. Because PCIe is the new type, AGP cards are no longer produced and should be removed from the configuration model. This is indicated by the large "X" in place of where the AGP card is in the hierarchy.

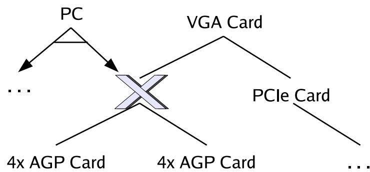


**Figure 1.** Extract of the PC domain. The taxonomic hierarchy of VGA cards is shown for two types: AGP and PCIe.

The following discussions focus on two issues of this removal. The first issue is concerned with the 4x AGP and the 8x AGP cards that are specializations of the AGP card. The second issue is concerned with the AGP card being a part of a PC.

## 3.3 Change Operations

Change operations have to be clearly defined. They must compare and present *structural* and *semantic* changes rather than *syntactical* changes in text representation. The latter is e.g. common when comparing versions of software code. But the same content can be modeled with different syntactical means or simply in different order. Two configuration models can be the same conceptually, but have very different text representations [18].

*Base operations* represent elementary changes that cannot be decomposed into simpler changes. Base operations are for example adding or deleting a concept definition, adding and deleting a concept attribute or a compositional relation, etc. This fine granularity of separated changes is not always appropriate. Changes should also be defined on a higher level that allows semantic interpretation. *Compound operations* group base operations into a meaningful unity [10]. Compound operations are for example modifications to compositional relations or constraints, that affect multiple concepts, or *tree-level* changes such moving a subtree of concepts or modifying an attribute value, which affects all descendants as well, due to inheritance within the taxonomy.

Grouping base operations to more meaningful compound operations is reasonable due to the fact that one (compound) operation is more concise than multiple (base) operations that might be needed because a change implies action in different places of the configuration model. This makes the use of compound operations more suitable for a user interface. And last but not least, they are needed because elementary changes may lead to incorrect configuration models [22]. In this case, additional change operations should re-transform the configuration model into a correct state [13].

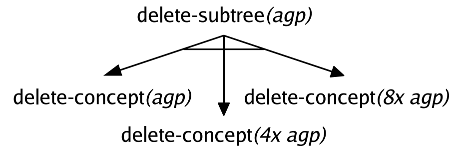


**Figure 2.** Definition of the compound operation delete-subtree$(apg)$. The arrows indicate its composition.

An example for a compound change operation is deleting a subtree of concept definitions. Figure 2 depicts how this operation delete-subtree$(apg)$ is composed of deleting concept $agp$ (delete-concept$(agp)$) and recursively deleting its descendants $4\text{x}agp, 8\text{x}agp \in \mathcal{C}$ with $(agp, 4\text{x}agp), (agp, 8\text{x}agp) \in \mathcal{H}$ (delete-concept$(4\text{x}agp)$ and delete-concept$(8\text{x}agp)$).

Compound operations can be composed of base operations and possibly other compound operations. The compositions forms a tree structure in which compound operations have further descendants and base operations are the leaf nodes, respectively.

Change operations have *preconditions*. This means that they can only be applied if certain assertions are satisfied by the configuration model. Typical preconditions are that some knowledge entity exists – i.e. it is defined in the configuration model. Deleting a concept $c$ for example can only be applied if $c$ is modeled: $c \in \mathcal{C}$.

The application of a change operation also has *postconditions* on the model: it changes assertions, introduces or deletes assertions. The set of assertions that is satisfied by the configuration model can increase or decrease, but definitely changes when an operation is applied. Adding elements to the configuration model usually increases the set of assertions while removing elements usually decreases it. Deleting concept $c$ for example removes the assertion that $c$ is modeled: $c \notin \mathcal{C}$.

Preconditions and postconditions both are represented by propositions that describe the content of assertions in a way that they are either true or false. A proposition $p \in P$ can define the existence (e.g. $c \in \mathcal{C}$) or absence (e.g. $c \notin \mathcal{C}$) of any knowledge entity as well as specific values for attributes, relations and constraints (e.g. $\text{value}(a) = 1$).

By organizing base operations in a taxonomy, the inheritance mechanism can be exploited to specify common properties of the operations in an efficient way. Common and varying properties of base operations are the preconditions defining their applicability and the postconditions they have on the configuration model. A taxonomy of change operations for ontology evolution is for example given in [9].

A change operation is *sound* if the operation itself is applicable and all operations it contains are applicable in some order. All sound changes to manipulate a configuration model can be specified by base or compound operations. Changes can be applied to a correct version of the model $\mathcal{M}$, and after all operations are performed, the model must transform into another correct version $\mathcal{M}'$ [22].

However, simply concatenating base operations to a compound operation in a pre-defined manner has some drawbacks:

- There may be a mismatch between the intent of the change and the way the pre-defined operation is composed.
- Unnecessary changes may be performed if they are applied independent from each other.
- The applicability of change operations depends on what is currently modeled. Therefore, the order in which base operations inside a compound operation are applicable can vary.

The first two items have also been identified by [21]. The third item introduces the fact that the knowledge inside the configuration model dictates when certain operations are applicable and when they can not. The following section details the dynamic composition of base operations into a compound operation.

## 4 Evolution Process

In the ideal case, a change operation can simply be applied to a configuration model. But two issues make this process a bit more complex. These issues are

1. that change operations are not always applicable – depending on the model, and
2. the interrelations of concept definitions according to taxonomic and compositional relations and constraints.

The latter means that a change operation can have unforeseen consequences leading to an incorrect configuration model, which has to be resolved by additional changes.

One change to the configuration model is implemented with one change operation. The issues mentioned above show the necessity of arranging the evolution process in three steps:

**Compilation of Complex Change Operations:** Complex change operations are compiled, based on the preconditions and postconditions of the corresponding base operations, before this operation is applied. A particular change may be implemented with different operations, depending on the configuration model.

**Change Propagation:** The identified change operations are applied to the configuration model with respect to their applicability. Specific combinations of changes and the affected knowledge entities require an analysis of the model for additional change operations.

**Change Validation:** After a change to the configuration model has been propagated, all defined syntax invariants are checked against the model. When an incorrectness is detected, additional change operations have to be identified to implement changes resolving this incorrectness. This means that the previous steps are performed again – leading to an iterative process.

The following three sections detail the compilation of complex change operations, the change propagation and the change validation, respectively.

### 4.1 Compilation of Complex Change Operations

Applying changes to a configuration model modifies its contents and / or structure. This is wanted since evolution intends to change the model. However, some changes can have unforeseen consequences. These can arise because of constraints and other, change-dependent, interrelations. Changing concept attributes for example also affects all descendants – see the inheritance invariant (**I2**).

Consequences of change operations can be evaluated by analyzing the configuration model. Additional changes that become necessary for repairing an incorrect model can be identified based on information about the nature of the change and the affected knowledge entities.

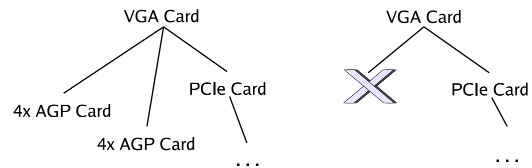


**Figure 3.** The concept VGA card is deleted. In the left its subconcepts are kept while in the right they are also deleted.

Deleting the AGP card, for example, is a simple operation when this concept is a leaf node.[2] Because there are descendants $4\text{x}agp, 8\text{x}agp \in \mathcal{C}$ with $(agp, 4\text{x}agp), (agp, 8\text{x}agp) \in \mathcal{H}$, however, $agp$ cannot be simply deleted. This would violate the specialization relation invariant (**I1**). There are two possible resolutions for this incorrectness: the 4x agp and 8x agp are also deleted (see Figure 3 on the right), or they are moved to some new parent. Considering the semantics of inheritance, $vga$, the parent of $agp$, is the next indirect superconcept and should be used as a new superconcept. This is shown in Figure 3 on the left. Another violation is that of the composition reference invariant (**I3**) because $agp$ is referenced in a compositional relation $(pc, agp) \in \mathcal{P}$. In this case the compositional relation should also be deleted.

Both violations of invariants and the additional change operations identified to repair them are given in Table 1.

Note that in general there are two possible changes to repair the violated composition reference invariant (see Table 1). These are

[2] For reasons of simplicity, at this point only dependencies concerning specialization and compositional relations are discussed, not considering constraints.

| Change | Incorrectness | Additional Changes |
|---|---|---|
| Remove Concept | $\exists c_2 \in \mathcal{C}$ such that $\forall c_1 \in \mathcal{C},\ (c_1, c_2) \notin \mathcal{H}$ | |
| | | add-specialization$(c_1, c_2)$ |
| | | delete-concept$(c_2)$ |
| | $\exists (a, p) \in \mathcal{P} \wedge \neg \exists c \in \mathcal{C}$ with name$(p)$ = name$(c)$ | |
| | | add-concept$(c)$ |
| | | delete-composition$(a, p)$ |

**Table 1.** Identifying additional change operations.

adding the corresponding concept and deleting the relation. Both are meaningful to repair the incorrectness – depending on the nature of change that led to this incorrectness. Adding the concept that is missing for the compositional relation, however, is not appropriate in the case that it has been deleted before due to the fact that this inverts the intended deletion.

Change operations that are inappropriate for repair can generally be identified based in their preconditions and postconditions. Deleting a concept $c$ has the precondition $c \in \mathcal{C}$ and postcondition $c \notin \mathcal{C}$, while deleting concept $c$ has the exactly inverse precondition $c \notin \mathcal{C}$ and postcondition $c \in \mathcal{C}$. It is apparent that either of these changes is not appropriate to repair an incorrectness that occurred because the other change.

Not every incorrectness can be resolved automatically. Some alternative resolutions can seem equally well suited. Figure 3 for example shows two conceivable scenarios. The descendants of the AGP card do not have to be deleted; they may be kept for legacy support.

## 4.2 Change Propagation

The preconditions and postconditions of change operations can be used to compute a temporal order. For some base operations there might not be any limitations while further operations can only be applied after postconditions of others satisfy preconditions of these.

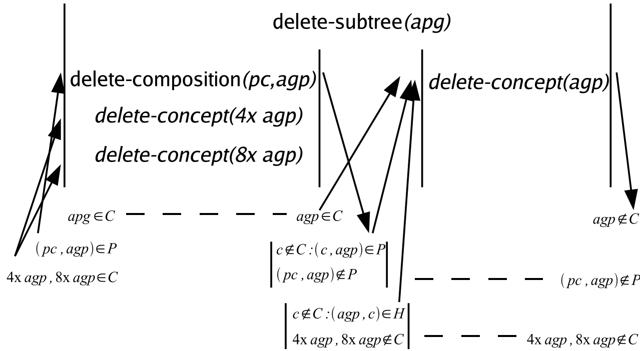


**Figure 4.** Temporal order of the compound operation delete-subtree$(agp)$. Time proceeds from left to right. Arrows represent pre- and postconditions. Dashed lines indicate persistence of propositions, if not negated.

Figure 4 depicts the temporal order of operations inside delete-subtree$(agp)$. The descendants of $agp$ and the compositional relation between $pc$ and $agp$ are deleted first (delete-subtree$(4\text{x}agp)$, delete-subtree$(8\text{x}agp)$ and delete-composition$(pc, agp)$). Preconditions for these are their existence – that is $4\text{x}agp, 8\text{x}agp \in \mathcal{C}$ such that $(agp, 4\text{x}agp), (agp, 8\text{x}agp) \in \mathcal{H}$ and $(pc, agp) \in \mathcal{P}$. After that, $agp$ itself can be deleted (delete-concept$(agp)$). Preconditions for this are the existence $agp \in \mathcal{C}$ and that there are no descendants $d \notin \mathcal{C}$ such that $(agp, d) \in \mathcal{H}$ and no compositions $c \notin \mathcal{C}$ such that $(c, agp) \in \mathcal{P}$.

The existence preconditions ($agp \in \mathcal{C}$ and $4\text{x}agp, 8\text{x}agp \in \mathcal{C}$ such that $(agp, 4\text{x}agp), (agp, 8\text{x}agp) \in \mathcal{H}$) have to be satisfied before the compound operation can be applied. Therefore, they are also preconditions of this compound operation. This is different for the absence preconditions ($c \notin \mathcal{C}$ such that $(agp, c) \in \mathcal{H}$ and $c \notin \mathcal{C}$ such that $(c, agp) \in \mathcal{P}$): they are postconditions of other operations inside this compound operation and therefore do not have to be satisfied beforehand.

Note that in Figure 4 the propositions $(pc, agp) \notin \mathcal{P}$ and $c \notin \mathcal{C}$ such that $(a, agp) \in \mathcal{P}$ are treated equally because the PC is the only aggregate that the AGP card is part of. Analogously, the propositions $4\text{x}agp, 8\text{x}agp \notin \mathcal{C}$ and $c \notin \mathcal{C}$ such that $(agp, c) \in \mathcal{H}$ are also treated equally.

At least one operation within a sound compound operation must be applicable! If there are base operations that do not have preconditions or that only have preconditions satisfied by the configuration model, these can be applied first; in an arbitrary order. After a change operation has been applied, the assertions represented by the configuration model have changed. This means that preconditions for other base operations may have become satisfied. Therefore, the applicability of all operations has to be verified after each application of an operation.

The algorithm to compute a temporal order between the base operations $B$ within a sound compound operation $o$ is given in the following. The configuration model is denoted with $\mathcal{M}$.

---

ALGORITHM: computeTemporalOrder$(o)$

---

1. Initialize the set of applicable operations $A = \emptyset$.
2. If $B = \emptyset$, then return $A$.
3. For all $b \in B$ do:
   (a) If the set of preconditions of $b$ is empty ($P_b = \emptyset$), then
      i. add $b$ to the set of applicable operations ($A = A \cup \{b\}$)
      ii. remove $b$ ($B = B \setminus \{b\}$).
   (b) Else, for all $p \in P_b$, do:
      i. If $p$ is not satisfied ($\mathcal{M} \cap p = \emptyset$), then
         - move $b$ to the end of $B$ ($\{b, b_1, \ldots, b_n\} \rightarrow \{b_1, \ldots, b_n, b\}$)
         - continue with next $b$.
      ii. Else if $p$ is the last element of $P_b$, then
         - add $b$ to the set of applicable operations ($A = A \cup \{b\}$)
         - remove $b$ ($B = B \setminus \{b\}$).
         - continue with next $b$.
      iii. Else continue with next $p$.
4. Continue with step (2).

---

In principle, the set of operations that belong to a compound operation can be split into two sets, according to whether they are applicable or not. An operation is applicable if all its preconditions are satisfied, if any, and cannot be applied elsewise. The order in which applicable operations are applied is not of importance.

## 4.3 Change Validation

After change propagation has taken place, all invariants defined for the configuration have to be checked against the model. If no violation is detected, the configuration model is transformed into a new,

correct, state – i.e. into a new version of that model. If a violation of an invariant is detected, this means that the change has introduced an incorrectness.

Incorrect configuration models are not viable. There are two possibilities to cope with this:

1. the configuration model has to be repaired by applying additional changes, or
2. the intended change has to be undone.

The latter indicates the need to define transaction sets for the the changes to a configuration model. All changes are accepted in one go if the new version of the configuration model is correct. If the new version is not correct and no additional changes are intended, it is always possible to reclaim the version from before the change.

## 5 Related Work

There are a few research groups also dedicated to the evolution of structure-based configuration models and models for product families. For example, in [17, 16] the notion of generic objects and the division of versions into variants and revisions [8] is used for evolution of configuration knowledge. A lot of ideas also come from previous work, for example [12, 13].

Ontology evolution has gained interest in recent years. This may have a lot to do with the Semantic Web. In [9] for example there is a taxonomy of change operations defined for ontology evolution. [20] focuses on consistency of evolving ontologies and defines invariants to check consistency of an ontology.

## 6 Outlook

This paper describes work in progress. This means that some work still has to be done. Future work includes the following issues:

- Invariants define well-formed configuration models. A list of invariants that completely cover all facilities of the modeling language is needed in order to guarantee correctness of a model despite its changes.
- Change operations are to be defined in a taxonomy. The characteristics according to which they may be aligned have to be identified. Possible choices are the types of changes (add, delete, modify), the knowledge entities they operate on (concept, specialization relation, compositional relation, constraint) or the preconditions defining their applicability and the postconditions the operations have on the configuration model.
- Changes applied to a configuration model may potentially violate invariants. Different types of changes applied to different knowledge entities are conceivable (see Section 4.3). The identification of appropriate change operations for all cases is still an open issue. Possible choices of characteristics for this identification are the arguments of an operation, its preconditions and postconditions, or a semantic interpretation of the nature of the change.

## 7 Conclusion

This paper addresses potential problems that may arise within continual evolution of configuration models. It presents an approach to prevent incorrect configuration models. This approach consists of a set of invariants to check the syntactical correctness of model, clear semantics of changes to the configuration model and their implementation in change operations. A three-step evolution process defines how to compile compound change operations, propagate changes and validate the configuration model after change propagation.

## REFERENCES


[1] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth, ‘Semantics and implementation of schema evolution in object-oriented databases’, in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pp. 311–322. ACM Press, (1987).

[2] Ted J. Biggerstaff and Charles Richter, ‘Reusability framework, assessment, and diretions.’, *IEEE Software*, **4**(2), 41–49, (1987).

[3] Reidar Conradi and Bernhard Westfechtel, ‘Version models for software configuration management’, *ACM Computing Surveys (CSUR) archive*, **30**(2), 232–282, (1998).

[4] Randall Davis, Howard E. Shrobe, and Peter Szolovits, ‘What is a knowledge representation?’, *AI Magazine*, **14**(1), 17–33, (1993).

[5] Dieter Fensel, *Ontologies – A Silver Bullet for Knowledge Management and Electronic Commerce*, Springer Verlag, 2001.

[6] Dieter Fensel, ‘Ontologies: Dynamics networks of meaning’, in *Proceedings of the 1st Semantic web working symposium*, (2001).

[7] Thomas R. Gruber, ‘Ontolingua: A mechanism to support portable ontologies’, Technical Report KSL 91-66, Version 3.0, Stanford University, Knowledge Systems Laboratory, (1992).

[8] Randy H. Katz, Ellis E. Chang, and Rajiv Bhateja, ‘Version modeling concepts for computer-aided design databases’, in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pp. 379–386. ACM Press, (1986).

[9] Michel Klein, *Change Management for Distributed Ontologies*, Ph.D. dissertation, Vrije Universiteit Amsterdam, 2004.

[10] Michel Klein and Natalya Noy, ‘A component-based framework for ontology evolution’, Technical Report IR-504, Department of Computer Science, Vrije Universiteit Amsterdam, (2003).

[11] Tero Kojo, Tomi Männistö, and Timo Soininen, ‘Towards intelligent support for managing evolution of configurable software product families’, in *Proceedings of 11th International Workshop on Software Configuration Management (SCM-11)*, pp. 86–101. Springer Verlag, (2003).

[12] Thorsten Krebs, Lothar Hotz, Christoph Ranze, and Guido Vehring, ‘Towards evolving configuration models’, in *PuK2003 – Papers from the KI Workshop*, pp. 123–134, (2003).

[13] Thorsten Krebs, Katharina Wolter, and Lothar Hotz, ‘Mass customization for evolving product families’, in *Proceedings of International Conference on Economic, Technical and Organizational Apects of Product Configuration Systems*, pp. 79–86, (2004).

[14] David B. Leblang and Paul H. Levine, ‘Software configuration management: Why is it needed and what should it do?’, in *Selected papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*, pp. 53–60. Springer-Verlag, (1995).

[15] Mark H. Lee, ‘On models, modelling and the distinctive nature of model-based reasoning’, *AI Communications*, **12**(3), 127–137, (1999).

[16] Tomi Männistö, *A Conceptual Modelling Approach to Product Families and Their Evolution*, Ph.D. dissertation, 2000.

[17] Tomi Männistö, Hannu Peltonen, and Reijo Sulonen, ‘View to product configuration knowledge modelling and evolution’, in *Configuration – Papers from the 1996 Fall Symposium*, pp. 111–118. AAAI Press, (1996).

[18] Natalya F. Noy, Sandhya Kunnatur, Michel Klein, and Mark A. Musen, ‘Tracking changes during ontology evolution’, 259–273, (2004).

[19] Daniel Sabin and Rainer Weigel, ‘Product configuration frameworks – a survey’, *IEEE Intelligent Systems*, **13**(4), 42–49, (1998).

[20] Ljiljana Stojanovic, *Methods and Tools for Ontology Evolution*, Ph.D. dissertation, Universität Karlsruhe, 2004.

[21] Ljiljana Stojanovic, Alexander Maedche, Boris Motik, and Nenad Stojanovic, ‘User-driven ontology evolution management’, in *Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management*, (2002.).

[22] Ljiljana Stojanovic and Boris Motik. Ontology evolution within ontology editors, 2002.

[23] W. F. Tichy, ‘Tools for software configuration management’, in *Proceedings of the International Workshop on Software Version and Configuration Control*, pp. 1–20. Teubner Verlag, (1988).

[24] David Whitgift, *Methods and Tools for Software Configuration Management*, Wiley & Sons Ltd., 1991.