Dependency Analysis and its Use for Evolution Tasks

Lothar Hotz[1], Thorsten Krebs[2], and Katharina Wolter[3]

[1] HITeC c/o Fachbereich Informatik, Universität Hamburg
Hamburg, Germany, 22527
hotz@informatik.uni-hamburg.de
[2] LKI, Fachbereich Informatik, Universität Hamburg
Hamburg, Germany, 22527
krebs@informatik.uni-hamburg.de
[3] LKI, Fachbereich Informatik, Universität Hamburg
Hamburg, Germany, 22527
kwolter@informatik.uni-hamburg.de

Abstract. During the life-cycle of products, evolution of the knowledge used for configuring these products is demanded. Change operations are introduced to capture modifications to the configuration model. Preconditions and impacts of these operations are explained and motivate a dependency analysis for supporting change operations on the configuration models. We give a detailed discussion of the dependency analysis for structure-based configuration models.

1 Introduction

Structure-based configuration is used to configure software-intensive systems in the European project *ConIPF (Configuration of Industrial Product Families)*. Since configuration models are used to configure families of products over time, it is apparent that new functionality is introduced for being able to create new products. This has an impact on the configuration model: the knowledge has to evolve in parallel with the components used for product derivation.

Change operations are introduced to capture modifications to the configuration model. But since modifications may lead to inconsistent states of the model, the need for complex change operations (i.e. a concatenation of basic change operations that are dependent on another) arises. Such complex change operations are built to avoid inconsistent states within the configuration model. The impacts of certain modifications have to be known for constructing a sequence of basic operations that fulfills the task of the complex operation and therefore avoids inconsistencies.

Impacts of modifications can be anticipated by knowing dependencies between the diverse knowledge entities of the modeling facilities of structure-based configuration. These dependencies can be computed by analyzing the model. Therefore, a dependency graph can be generated that helps in predicting impacts of modifications. Evolution tasks can be supported with such dependency graphs.

In this paper, we focus on the dependency analysis for the structure-based configuration tool KONWERK [3]. The modeling facilities of this configuration approach and their dependencies are introduced, analyzed and discussed in detail. As an example for applying the dependency analysis we show how evolution of the configuration model can be enhanced by using dependency graphs. Further potential application areas are mentioned in the summary.

The remainder of this paper is organized as follows: in Section 2 we describe the basic modeling facilities and the configuration procedure of structure-based configuration. In Section 3 evolution of the configured product family and the configuration model is introduced. Change operations are identified as a mechanism to capture modifications to the model. Section 4 presents the general idea behind dependency analysis, lists and discusses concrete dependencies we identified for structure-based configuration and describes how these dependencies can be computed automatically. In Section 5 we give the evolution of configuration models as an example for applying dependency analysis. Finally, we conclude with the summary in Section 6.

2 Structure-based Configuration

2.1 Modeling Facilities

In structure-based configuration, product components and the mapping between these are formalized in a configuration model. This model is represented with the *Configuration Knowledge Modeling Language (CKML)*[4] and is used for automated product configuration. Components of diverse types can be represented with concepts provided by the modeling facilities of structure-based configuration. A brief definition is given in the following (see [2] for a more detailed view):

- Each concept has a name which represents a uniquely identifiable character string.
- Attributes of concepts can be represented through parameters. A parameter is a tuple consisting of a name and a value descriptor. Diverse types of domains are predefined for value descriptors - e.g. integers, floats, sets and ranges.
- Concepts are related to exactly one other concept in the *taxonomic relation*. Thus, the latter concept is the superconcept of the former one – the subconcept. Properties (i.e. parameters and relations) of the superconcept are inherited by the subconcept. The superconcept subsumes the subconcept, i.e. all properties of the subconcept are subsets of those of the superconcept.
- *Partonomies* are generated by modeling compositional (has-parts) relations. Such a relation definition contains a list of parts (i.e. other concept definitions) that are identified by their names. Each of these parts is assigned with a minimum and a maximum cardinality that together specify how many instances of these concepts can be instantiated as parts of the aggregate.
  Has-parts relations are a class of compositional relations. Thus, own relations with new names can be modeled – e.g. a concept can contain a has-hardware

[4] CKML is an extension of the language BHIPS used for KONWERK

definition next to a has-software definition to emphasize on the difference between the parts in both relations.

Concepts describe classes of objects from which multiple concept instances can be generated during the configuration process. The concept is a static description that is not altered at any time while a concept instance of such a concept definition is dynamically created during the configuration process and configured until its properties are fully specified in terms of the task specification. A concept instance always is instance of exactly one concept definition.

Constraints are used to describe restrictions between properties of distinct concepts. There are three layers for describing constraints: the *conceptual constraint* consists of a precondition that has to hold in order to be executed. *constraint relations* define the restriction itself. A constraint relation is reusable for an arbitrary number of conceptual constraints. A conceptual constraint can call an arbitrary number of constraint relations. Constraint relations can be of different types (e.g. functions, extensional, etc.). Instances of constraint relations *(constraints)* are automatically generated during configuration and represent restrictions between concept instances. See Figure 3 for an example.

By analyzing the configuration model necessary decisions that have to be made are inferred during the configuration process. Each decision is represented by a configuration step. There are 4 types of configuration steps:

1. *parameterization* represents a decision of setting a parameter.
2. *specialization* represents a decision of specializing a concept instance to a more specific concept according to the taxonomy.
3. *decomposition* represents a decision of decomposing a concept into its parts (i.e. top-down reasoning). During decomposition appropriate instances of the parts are generated (instantiation is seen as a sub step of decomposition).
4. *integration* represents a decision of integrating a part into an aggregate (i.e. bottom up reasoning). During integration appropriate instances of the aggregates are generated (instantiation is also seen as a sub step of integration.)

Consistency of the model is defined as follows (compare [7]):

Specialization-related: given a super- and a subconcept, all values of the subconcept's properties have to be subsets of the corresponding property (identified by name) of the superconcept.

Composition-related: given an aggregate and its parts, each part has to be defined as a concept.

Constraint-related: given a constraint and the participating concept properties, the constraint may only use value ranges defined in the model. Furthermore, only subsets of values of the concept properties are allowed as propagation results.

2.2 Example Domain

As an example we give the application domain of Car Periphery Supervision (CPS) systems introduced by [8]. A CPS system consists of automotive systems that are based on sensors installed around the car to monitor its local

environment. Sensor measurement methods and evaluation mechanisms provide information for various kinds of high-level comfort and safety related applications like Parking Assistance and Pre-Crash Detection. In CPS systems sensors are mounted on the vehicle (e.g. ultrasonic sensors hidden in the bumper).

In Figure 1, a small configuration model taken from the CPS domain is presented. Besides the specialization and decomposition one constraint is given (see Figure 3), which ensure that the parameters `Type` of `Main` and `Sensor-Configuration` are equal. This (probably simplifying) example is used in the following sections for illustrating the notions behind dependency computation.

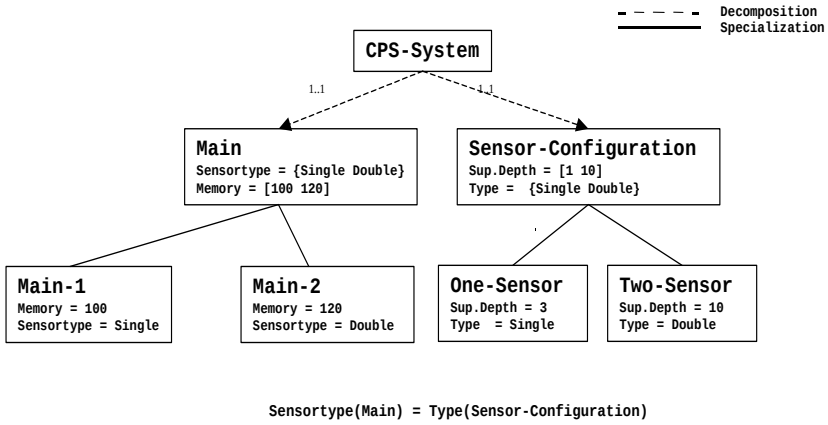

Fig. 1. Example of a configuration model

Given a configuration model the configuration steps to be resolved during the configuration process are automatically computed from it. In Figure 2, all configuration steps are listed (incl. definition of later used short-cuts) which are identified when the configuration model shown in Figure 1 is given. All configuration steps have to be determined (i.e. configured) in one configuration process. A terminal value (i.e. a value not further specifiable) for each configuration step has to be computed by a value determination method (like automatic inferencing, asking the user, invoking a function etc.). Computing of the dependencies of configuration steps is presented in the following.

3 Operations for Evolving Configuration Models

Evolution of products and product components is inevitable throughout their life cycle – driven by advancing technology, increasing customer requirements or bug fixes. New versions and variants are built and therefore also modeled. It is common that evolution is anticipated to a certain extent, e.g. by modeling planned features [4]. But it is impossible to predict all future developments (like bug fixes) and therefore the model has to be repaired at some time.

```
Instantiate CPS-System (Instant(CPS-System))
Decompose CPS-System (Decompose(CPS-System))
Instantiate Main (Instant(M))
Instantiate Sensor Configuration (Instant(SC))
Specialize Main (Specialize(M))
Specialize Sensor Configuration (Specialize(SC))
Determine the parameter Sensortype of the Main (Param(Type,M))
Determine the parameter Memory of the Main (Param(Memory,M))
Determine the type of the Sensor Configuration (Param(Type,SC))
Determine the parameter Supervision Depth of the Sensor Configuration
(Param(Sup.Depth,SC))
```

Fig. 2. Configuration Steps of the Example with Short-cuts

Concept
name: M-SC-Equal
precondition:
?c **name:** CPS-System
?s **name:** Sensor-Configuration
relation: part-of ?c
?m **name:** Main
relation: part-of ?c
constraint-call:
equal (?m sensortype) (?s type)

Fig. 3. A Conceptual Constraint

Evolving the configuration model can have diverse effects on its consistency and on existing or potentially derivable products. Different scenarios like richer or smaller variety of products (leading to different configuration solutions) are conceivable. In the following, we confine ourselves to impacts that modifications to the configuration model can have and how consistency of the model can be guaranteed (see Section 2).

Modifications to the configuration model are captured through *change operations*. *Basic operations* are those that cannot be further partitioned – i.e. simple modifications like adding a parameter value. *Complex operations* are composed of multiple basic operations or include some additional knowledge about the modification. They provide a mechanism for grouping basic operations into a logical unity (compare [6]). This is needed because an operation may lead to an inconsistent state of the configuration model and thus should be followed by further operations that correct this situation. Hierarchies of change operations

can be formulated to exploit the inheritance mechanism for specifying common properties.

Standard operations e.g. are adding, deleting and modifying the diverse knowledge entities. Adding a parameter value and adding a compositional relation have similar characteristics and impacts while they are fundamentally different from deleting these knowledge entities. Thus, it is obvious that a hierarchy of change operations is reasonable. As an example, in the following we show our hierarchy of operations for the complex change operation *modifying concept parameters* (operations are bold, impacts are in italics):

- **Renaming a parameter**
  - *in case of existing subconcepts: also change the name in those*
  - *in case of constraints binding this parameter: change the name in the precondition of the conceptual constraint*
- **Changing a parameter value**
  - *in case of existing subconcepts: check that the corresponding parameter values of subconcepts are subsets of the new value*
  - *in case of constraints binding this parameter: check that the constraint relation still computes reasonable values*
  - **Enlarging value range**
    - *no impacts on inheritance have to be addressed – subconcepts will keep subsets of values after this operation*
  - **Scaling down value range**
    - *impacts on inheritance have to be addressed – the new value might be a subset of the former subconcept's values*
  - **Changing value type**
    - *check that values are still reasonable*

To find out which combination of basic change operations is needed to perform a more complex operation, a dependency analysis is needed. Impacts of modifications are examined and indicate the need for further modifications to reach a consistent state of the knowledge base.

4 Dependency Analysis

Dependency detection is performed on the configuration model as a whole. It can be used for computing the impacts a change of the configuration model would have.

When the configuration model is specified, all possible dependencies are already present, i.e. it is not necessary to *model* dependencies, but only to compute them from the configuration model. Besides the configuration model the dependencies are of course affected by the semantics of CKML, which is implemented in the inference machine of KONWERK. We identified rules that are based on the configuration model on the one hand, and on the inference machine that interprets the configuration model on the other hand. An example rules is:

A specialization determines parameterization of all parameters which are new or changed in c in respect to its superconcept in the taxonomical hierarchy.

There are rules between configuration steps, conceptual constraints and constraint relations, and those which are established in the constraint net by instances of constraint relations. However, the rules are not related to a specific domain (like CPS). Thus, when an arbitrary configuration model expressed in CKML is given with these rules the dependencies can be computed automatically. All identified rules are given in Section 4.2. But first we present the general concept of dependency identification.

4.1 General Idea

Two general types of dependencies can be identified: those things that are *prerequisites* for other things, and those things that *determine* other things.

Looking at the example in Figure 1 the concept Sensor-Configuration can only be instantiated after the aggregate named CPS-System is decomposed. Thus, the decomposition of the aggregate is a *prerequisite* of the instantiation. However, the value of the instantiation is not determined by such a dependency. On the other side the parameterization of the parameter Type of an instance of concept Main determines the specialization of this instance, e.g. when Type = Double is given, a Main instance would be specialized to Main-2. Thus, when deciding a parameter value of an instance the specialization of that instance may be determined. This type of dependency is called *determination*. Hereby a value in the partial configuration is changed.

Also for constraints, prerequisites and determination dependencies can be identified. For conceptual constraints, only if the patterns listed in the condition part of a conceptual constraint (see Figure 3) are fulfilled, the constraint calls specified in the constraint-calls part of the conceptual constraint are instantiated. Thus, the patterns are prerequisites for the constraint-relation instance. The constraint itself (in our example the constraint equal) *determines* the values of the involved parameters.

In Figure 4 the dependencies of the example are presented in a *dependency graph*. An arrow means the node at the arrowhead *depends on* the node at the end of this arrow. Dependencies of type prerequisite are depicted as *v-edges*. Dependencies of type determination are depicted as *d-edges*.

Given such a dependency graph, when a product is configured, not all dependencies are used for each task. Thus, the dependencies for one configuration depend on the given task specification. However, one configuration process is a walk through the general (i.e. not task-specific) dependency graph.

In Figure 5 the decomposition of the CPS-System concept and the specialization of the Sensor-Configuration is given by the user. The inference machine computes the other configuration steps. This is done by the indicated walk through the dependency graph. When the values for memory of Main and for Type of Sensor-Configuration are given by the task specification, the equal constraint concerning the parameters Type of Main and Sensor-Configuration might indicate a

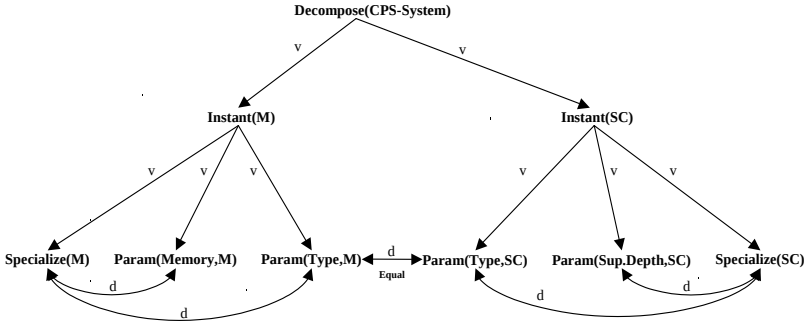

Fig. 4. Dependencies of the Example with Prerequisites and Determinations

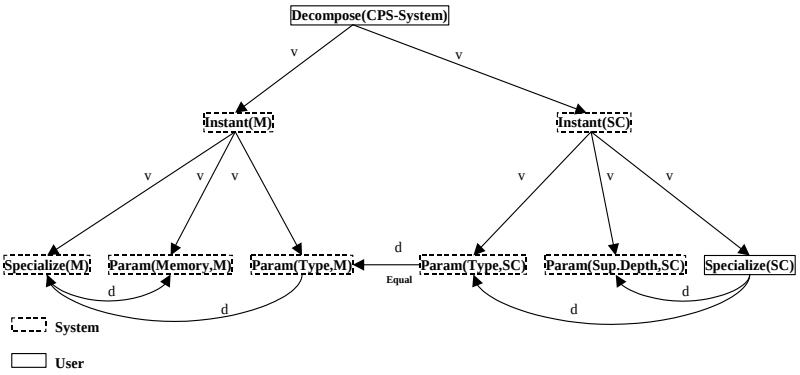

Fig. 5. A Configuration

conflict, i.e. the case when the memory is 100 and the supervision depth is 10 (see Figure 6). In general when a node has two incoming *d-edge* dependencies (like Specialize(M), Param(Type,M), Param(Type,SC), Specialize(SC)) a conflict at this node can probably occur.[5]

The general idea is, when a configuration model is given, all possible dependencies can be computed in advance and independently of a given task. Thus, a dependency graph can be directly computed from a configuration model, and can be used for dependency analysis. For operations that are related to a specific task (like "give me the parts of the model, that are considered for this specific product") the necessary parts of the dependency graph can be activated. The prerequisites (*v-edges*) are considered as conditions for the activation of a sub graph. Thus, for a given task specification those conditions can or cannot be fulfilled, which leads to an activation or non-activation of the following sub graph. Furthermore, patterns of conceptual constraints are used as prerequisites in the dependency graph. This is presented in detail in the next section.

[5] Not considering user interactions which can cause conflicts at any node.

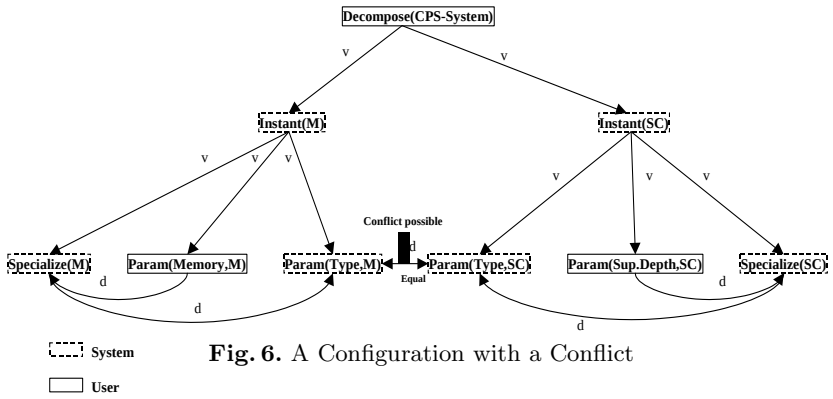

Fig. 6. A Configuration with a Conflict

4.2 Dependencies in Structure-based Configuration

Dependencies between configuration steps and constraints are listed and explained in detail in the following.

Between configuration steps: In this section for each type of configuration step the dependencies are listed. A configuration step type, the dependency type, and the dependent configuration step type is given.

1. Decomposing an aggregate is a prerequisite for existence of the parts as an instance.
2. Decomposing an aggregate determines the specialization of the aggregate.
3. Existence of an instance is a prerequisite for the parameterization of the parameters.
4. Existence of an instance is a prerequisite for the specialization of the instance.
5. Existence of an instance is a prerequisite for the decomposition of the instance.
6. Existence of an instance is a prerequisite for the integration of the instance.
7. Specialization of an instance to a concept type c is a prerequisite for further specializing the instance.
8. Specialization of an instance to a concept type c determines parameterization of all parameters which are new or changed in c.
9. Specialization of an instance to a concept type c determines decomposition of all relations which are new or changed in c.
10. Parameterizing a parameter p of an instance determines the specialization of the instance.
11. Integration of a part in an aggregate determines the decomposition of the aggregate.
12. Integration of a part in an aggregate determines the specialization of the aggregate.

Between constraints: At a first glance, dependencies between configuration steps and conceptual constraints could be considered by depending instantiation

steps with the patterns that match the related instance (Figure 7 upper part). But because only when all patterns of a conceptual constraint are fulfilled the related constraint relation is instantiated, one can summarize all patterns of a conceptual constraint to one node (as it is done in Figure 7 lower part). This node is a prerequisite for the constraint relation. Thus, dependencies between configuration steps and conceptual constraints can be computed by looking at all conceptual constraints. The rules between configuration steps and conceptual constraints are:

1. Instantiating a concept c is a prerequisite for all conceptual constraints, which contain a pattern that subsumes c.
2. Parameterizing a parameter p of an instance of type c is a prerequisite for all conceptual constraints, which contain a pattern that subsumes c and contains p.
3. Decomposing an aggregate via relation r is a prerequisite for all conceptual constraints, which contain a pattern that subsumes c and contains r.

Between conceptual constraints and constraint relations the rule is:

1. All patterns of a conceptual constraints are prerequisites for all constraint relations specified in the constraint call!

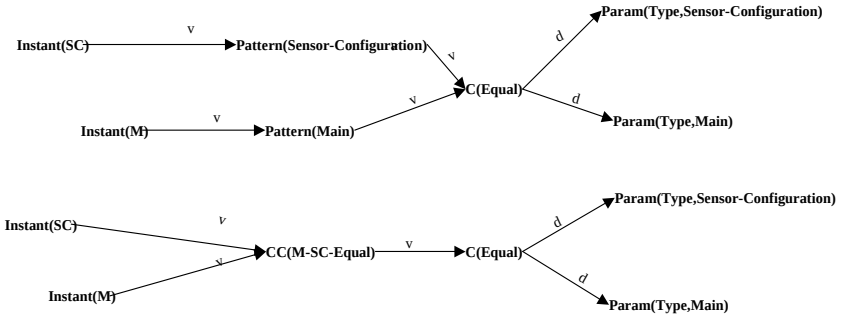

Fig. 7. Representation alternatives for dependencies between instantiation configuration steps and conceptual constraints, only the lower graph is actually necessary

Constraints always restrict parameters, or relations of diverse concepts, which is done by introducing constraint variables in the constraint net. Those variables can be identified with the appropriate configuration step, thus, leading from a situation given in Figure 8 lower part to the situation given in Figure 8 upper part.

Between configuration steps and constraint relations:

1. Parameterization of a parameter p constraint call contains variable referring to p.
2. Decomposition of a relation r constraint call contains variable referring to r.

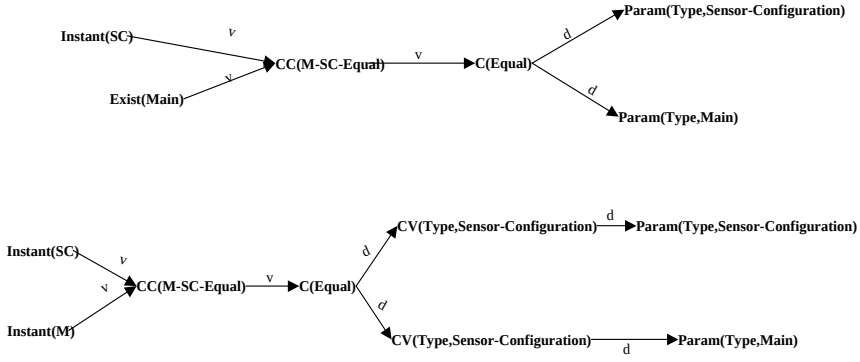

Fig. 8. Example: Identifying Constraints and Configuration Steps

3. Specialization of an instance constraint call contains variable referring to $instance - of$.

Constraints themselves can be uni-directed or multi-directed. Therefore, the determination dependencies between the constraint variables are defined accordingly:

1. Uni-directed Constraint of the form: $e_1 \dots e_n \rightarrow e_m \dots e_z$:
   $e_1 \dots e_n$ determines $e_m \dots e_z$.
2. Multi-directed Constraint of the form: $e_1 \dots e_n \leftrightarrow e_m \dots e_z$:
   e_1 determines $e_2 \dots e_z$
   e_2 determines $e_1, e_3, \dots e_z$
   etc.

In Figure 9 the complete dependency graph for the example introduced above is given.

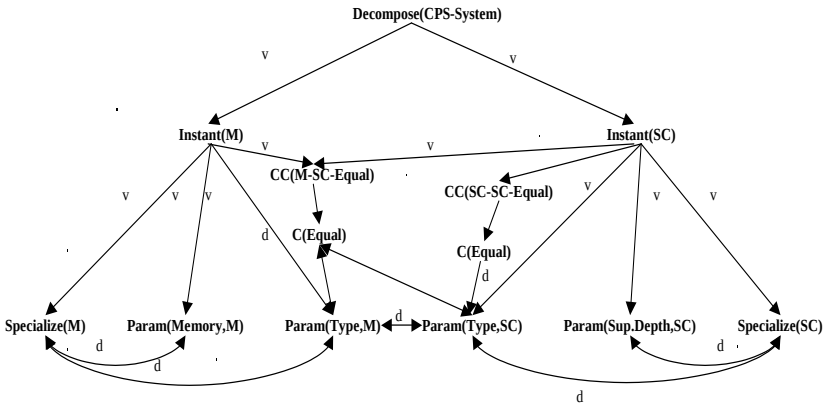

Fig. 9. Dependency graph for the example

Value-Related Dependencies: Up to now, only dependencies without considering values are taken into account. But also value-related dependencies like optional, alternative, multiple parts, dependencies related to a specific parameter value, number restricting constraints, and alternatives for integration steps (or-operator in part-of) have to be taken into account. For each part which is mentioned in any decomposition a graph is generated containing all dependencies related to that part - i.e. a sub graph for that part is created. In the dependency graph the *v-edge* to this graph is indicated with the number restriction of the decomposition, e.g. $[0..1], [1..1], [0..n]$.

Multiple instances of one concept can occur when conceptual constraints with patterns filtering the same concept. In such a case:

- Also only one partial graph for each concept is generated, e.g. only one for Sensor-Configuration!
- Constraints between different parameters of those concepts are indicated with further *d-edges*
- Constraints between the same parameters (e.g. Type) are indicated with numbers on *d-nodes*

Dependencies for constraints that contain number restrictions can be identified with decomposition steps. This can be done because with those constraints the number of parts of an aggregate is determined and the control module instantiates the parts according to those numbers. Thus, the behavior in this case is like a decomposition step.

4.3 Computing Dependencies

Given a configuration model in a declarative syntax (e.g. in CKML or in a tool-specific language like EngCon's XML notation [1] or BHIBS of KONWERK [3]), the dependencies are already present in the configuration model and can be transformed to an appropriate representation, e.g. a graph data type. This can be done by parsing the configuration model by taking the declarative syntax into account and using the previously explained rules.

Transforming the declarative model into a dependency graph In the previous section for each situation (i.e. configuration step, conceptual constraint and constraint relation) the dependencies are identified. By using those rules and one of the following methods, a complete dependency graph can be computed.

A further opportunity is to compute the dependencies by configuration, thus, the dependency graph is "configured" e.g. by using KONWERK. This can be done by once defining a model which describes the dependency rules in terms of concepts, i.e. defining a dependency meta model. While configuring with such a dependency model the concepts of the configuration model (e.g. of the configuration model of Figure 1) are inspected with inspection methods. Inspection methods typically compute information on objects themselves, here on concepts themselves, like all parameters of concept A, all patterns of conceptual constraint

C etc. Configuration with such a meta model would lead to a dependency graph which represents the dependencies of the configuration model. The dependency graph can again be used as a configuration model which is activated during configuration with the configuration model activating a task-specific dependency graph.

Because procedural knowledge defines known dependencies between configuration steps, this knowledge can be incorporated when the dependency graph is computed. Thus, when computing the dependency graph procedural knowledge can be used for more restricting the resulting graph. This topic has not yet been taken into account.

5 Using Dependencies for Evolution

In the previous sections we have discussed how the dependency analysis is realized. In this section we show how the known dependencies can be used to build complex change operations. The CPS example from Figure 1 in Section 2 will be extended to show how modifications to the configuration model can lead to inconsistencies. Therefore, complex change operations are build to prevent such situations.

We assume that exactly one Sensor-Configuration of type One-Sensor exists – this one is an Ultrasonic Sensor (see Figure 10).

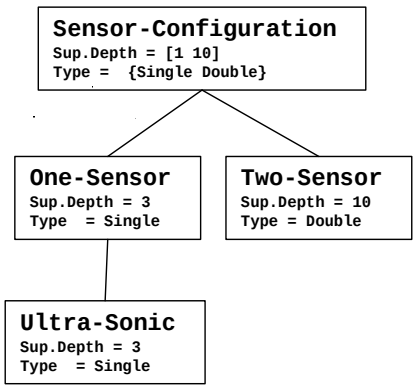

Fig. 10. The CPS Example with Ultrasonic Sensor

In our example, a new type of sensor is created – a Short-Range Radar. There also exists a Sensor-Configuration with exactly one of these sensors. Therefore we introduce SR Radar as a new Specialization of One-Sensor. The short range radar has a supervision depth of 5 meters (which is not a subset of the corresponding parameter value in One-Sensor). Thus, the value for the parameter Sup.Depth of the One-Sensor has to be corrected accordingly. Figure 11 shows the correct new version of the configuration model.

To avoid the inconsistency mentioned above, complex change operations are built from the basic change operations *add concept* and *enlarge parameter value*. This results

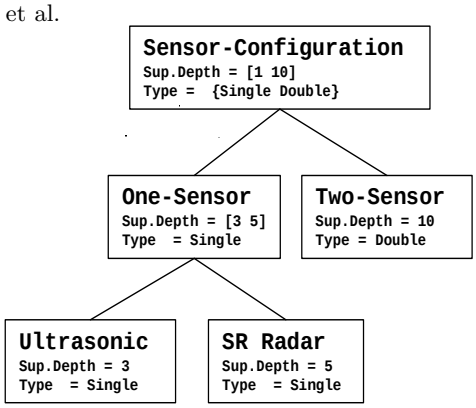

Fig. 11. The CPS Example with Ultrasonic and SR Radar Sensors

in the following complex change operations:

add concept(Ultrasonic) → *add concept(Ultrasonic), apply superconcept(Ultrasonic, One-Sensor)*

add concept(SR Radar) → *add concept(SR Radar), apply superconcept(SR Radar, One-Sensor), enlarge parameter value (One-Sensor Sup.Depth [3 5])*

This is of course a very simple example. But it should make clear how the dependency analysis can be used to guarantee consistency for evolving configuration models. The properties (Type and Sup.Depth) e.g. are defined in the basic concept Sensor-Configuration and therefore can be automatically processed. The three types of consistency declared in Section 2 have to hold and can be processed separately. Furthermore, the dependency analysis can be used to inspect other relations and restrictions (that have been added to more specific concept definitions). This means, when changes are given, the computed dependent parts of the model have to be considered for corrections and / or remodeling.

6 Summary

We have shown how dependencies can be computed from the declarative model in structure-based configuration. Rules according to the interpretation of the configuration model with the inference machine have been defined. They can be computed with simple graph algorithms and the computed dependencies can be used for computing impacts of changes of the configuration model. Short implementation experiences show that the dependency analysis can be easily implemented. However, a full implementation and integration in the configuration process is not yet available and will be done in future work.

Given a dependency graph for a configuration model, this graph can be used for diverse operations and presentations. If, for example, this information is made accessible during the configuration process it could help in reducing configuration effort. Furthermore, dependency graphs can be used for training purposes, e.g. by clarifying the relations and dependencies in the model.

Further topics are conceivable as potential application areas for the dependency analysis shown in this paper. Knowing and therefore being able to predict the impacts of changes, innovative configuration[6] can be realized. [5] Another application area is reconfiguration: knowing the modifications to a configuration model and their temporal order and knowing with which version of the configuration model a product has been configured with, it is possible to reconfigure it – e.g. for adapting new development or bug fixes. These topics have not yet been addressed by our research group and therefore present future work.

Acknowledgments

This research has been supported by the European Community under the grant IST-2001-34438, ConIPF - Configuration in Industrial Product Families.

References

1. V. Arlt, A. Günter, O. Hollmann, T. Wagner, and L. Hotz, 'EngCon - Engineering & Configuration', in *Proc. of AAAI-99 Workshop on Configuration*, Orlando, Florida, (July 19 1999).
2. A. Günter, *Wissensbasiertes Konfigurieren*, Infix, St. Augustin, 1995.
3. A. Günter and L. Hotz, 'KONWERK - A Domain Independent Configuration Tool', *Configuration Papers from the AAAI Workshop*, 10–19, (July 19 1999).
4. A. Hein, J. MacGregor, and S. Thiel, 'Configuring Software Product Line Features', in *Proc. of ECOOP 2001 - Workshop on Feature Interaction in Composed systems*, Budapest, Hungary, (June, 18 2001).
5. L. Hotz and T. Vietze, 'Innovatives Konfigurieren in technischen Domänen', in *Proceedings: S. Biundo und W. Tank (Hrsg.): PuK-95 - Beiträge zum 9. Workshop Planen und Konfigurieren*, Kaiserslautern, Germany, (February 28 - March 1 1995). DFKI Saarbrücken.
6. M. Klein and N.F. Noy, 'A Component-based Framework for Ontology Evolution', in *Proceedings of the Workshop on Ontologies and Distributed Systems, IJCAI-03*, Acapulco, Mexico, (2003).
7. T. Krebs, L. Hotz, C. Ranze, and G. Vehring, 'Towards Evolving Configuration Models', in *Proc. of 17. Workshop, Planen, Scheduling und Konfigurieren, Entwerfen (PuK2003) – KI 2003 Workshop*, pp. 123–134, Hamburg, Germany, (September, 15-18 2003).
8. S. Thiel, S. Ferber, T. Fischer, A. Hein, and M. Schlick, 'A Case Study in Applying a Product Line Approach for Car Periphery Supervision Systems', in *Proceedings of In-Vehicle Software 2001 (SP-1587)*, pp. 43–55, Detroit, Michigan, USA, (March, 5-8 2001).

[6] A configuration process is called innovative, when a solution is computed that has not previously been covered by the configuration model