Reasoning Methods for Knowledge-based Product Derivation

Lothar Hotz[1] and Thorsten Krebs[2]

[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

Abstract. Because of the possibly large variability in software systems and the complex dependencies between individual software components, product derivation in the context of software product families is not a trivial task. In this paper, we present reasoning methods known from structure-based configuration. These methods have been successfully applied to product derivation in technical domains and are deemed suitable also for software-intensive domains. Starting with i) a model describing the variability of already realized software components and ii) a concrete task specification for a specific product, during a knowledge-based product derivation process a description of the needed software components is derived. This description is to be used for realizing, i.e. assembling the desired product. The applicability and usefullness of the reasoning methods are shown by configuring features of a car periphery system.

Key Words: Software Variability Modeling, Feature Modeling, Product Derivation, Knowledge-based Configuration, Reasoning Methods, Logical Inferencing

1 Introduction

One goal of software product lines is to enable and enhance planned reuse of software components [1, 2]. For a given *task specification*, which describes product capabilities needed for realizing specific customer requirements, it is desirable to select the necessary software components that realize these customer requirements. Furthermore, when automating this process at suitable places several aspects like *completeness* (i.e. selecting all necessary components) and *correctness* (i.e. selecting only compatible components) must be ensured. This can be achieved by using models of features [3] (*feature models*), software components (*component models*) and their relations describing all variability in the product line. In our work we explore the possibility to use such kind of knowledge for an automated product derivation process using methods of knowledge-based configuration especially *structure-based configuration* developed in Artificial Intelligence [4, 5]. Thus, we call our approach a *knowledge-based* product derivation process (kb-pd).

In this paper, we present the basics of the knowledge-based product derivation process which is used to derive a product specific for a given set of requirements. Those basics are given by reasoning methods which are discussed in the following. Furthermore, a case study taken from an automotive domain shows how an implementation of the proposed automated derivation process based on the described semantic aspects can be realized. Syntactic issues are not considered, see [6] for a Lisp-like notation, [7] for an XML-notation and [8] for an UML-notation.

The remainder of this paper is organized as follows: first we give a short introduction in the examples domain (Section 2). In Section 3 we present preconditions which have to be fulfilled for using inference mechanisms that are discussed in more detail in Section 4. In Section 5 we shortly present our experiments which demonstrate the applicability of the previously discussed reasoning methods and conclude in Section 7.

2 Example Domain: Car Periphery Supervision

In this paper, for illustrating we take examples from the Car Periphery Supervision (CPS) domain. CPS systems monitor the local environment of a car. They comprise a family of automotive systems that are based on sensors installed around a vehicle. The recording and evaluation of sensor data enables different kinds of applications (product capabilities). These can be grouped into safety-related applications like pre-crash detection, blind spot detection and adaptive control of airbags and seat belt tensioners, and comfort-related applications like parking assistance and adaptive cruise control [9]. Different applications need different kinds of sensor data. For instance, while for parking assistance the range of 2.5 meters surveyed by using ultrasonic sensors is sufficient and also gives higher accuracy in that range, for pre-crash detection a longer distance has to be monitored and short range radar is required.

Applications realizing CPS-systems consist of a number of hardware and software components resulting in *software-intensive* systems. Furthermore, for realizing different customer needs a wide variability of hardware and software components are present, out of which the appropriate components must be selected to fulfill the needs. Thus, those domains give raise to probably *automatic* product derivation support.

3 Preconditions for Logical Inferencing

In order to be able to use logical inferencing, the following preconditions have to hold:

– A *model* of all components usable for product derivation (the *asset store*) has to be set up (this process is called *modeling*). The language we develop for specifying knowledge about variability in the product line is logic-based and domain independent. Thus, models contain logical concepts describing features of a system as well as software and hardware components (see Section 3.1) [10,11]. The language used in our approach is called *Configuration knowledge modeling langauge* (CKML).
– *Consistency* of the model has to be guaranteed. Basically consistency means, that no contradicting forms are included in the model. For ensuring consistency, several checks are presented in Section 3.2.

– For a specific product, the customer requirements are abstracted by the *task specification* (see Section 3.3). This task specification describes features and components that are desired by the customer, thus, which have to be selected from the configuration model as a minimum. The goal is to infer all needed components for realizing the application by using automatic reasoning. To guarantee *completeness* of the solution, the process has to select all necessary components that together realize the required functionality.

In the knowledge-based product derivation process – i.e. a process starting from the task specification and using the configuration model – a *configuration description* is derived. This is done by starting with an *initial* configuration which represents the selected features and components from the task specification, then going via several *partial* configurations which describe the intermediate states, to an *end* configuration that can be used to retrieve appropriate software components from the asset store.

3.1 Modeling for Knowledge-based Product Derivation

We distinguish between the *conceptual model* for describing all variability of features and components in a product line and the *procedural model* for describing the process of derivation, thus basically, the order of configuration decisions. In the following, the modeling facilities and their semantics in the kb-pd process are presented. Furthermore, the meaning realted to the end configuration is discussed.

Conceptual model A *conceptual model* describes basic product entities by means of *concepts*. Each concept c is specified by a name $id(c)$, parameters $P(c)$ and relations $R(c)$. The set $Prop(c)$ is called *properties* of concept c and is the union of its parameters and relations $P(c) \bigcup R(c)$. Three relation types are defined: *structural relations*, *specializations* and restrictions between concepts and their properties expressed by *constraints*.

– A *concept name* $id(c)$ of concept c specifies the concept type and is used for referring to this concept from various places in the model. The semantics of a name with respect to the kb-pd process is an abbreviation for all properties which are defined within that concept. In the resulting configuration description the name specifies the type or class of the configured component.
– Each parameter $p \in P(c)$ of a concept c consists of a *name* $id(p_c)$ and a *value range* $v(p_c)$ specified by an *object descriptor*. The set of all parameter names of concept c is identified by $P^{id}(c) = \bigcup id(p_c)$. Object descriptors can be symbols (strings), numbers, intervals and sets of symbols, numbers or intervals. During the kb-pd process, each parameter is set to a *terminal value*[3] which lies in the pre-defined value range. In the resulting configuration description the parameter describes an attribute of the configured component.

[3] A terminal value is a value that is completely specified with respect to the task specification. It can be a single value or a set.

– A specialization relation is defined by relating a concept sub to exactly one superconcept sup, vice versa, a superconcept can have several subconcepts. All properties of the superconcept are inherited by all subconcepts. Properties of the superconcept can be overridden in the subconcept description, but all values of properties in subconcepts have to be *subsets* of the related property values of the superconcept (see also Equation 1).
– Each structural relation $r \in R(c)$ of a concept c is described by a relation description consisting of a name $id(r_c)$ and a *relation descriptor* $v(r_c)$. A relation descriptor is defined by giving a *specifier* as a common type for all parts (i.e. the superconcept of all parts) $v^{type}(r_c)$ and a specifier for each related part $v^{par_i}(r_c)$. This defines a one (aggregate) to n (parts) relation. Each specifier consists of a type s^{type}, a minimum number s^{min} and a maximum number s^{max}. The type refers to concept names applicable for this relation. The number restrictions specify the cardinality of one specific type (e.g. optional parts are represented by $s^{min} = 0$ and $s^{max} = 1$).
  Parts of a relation description ($parts(r_c) = \bigcup_{i=1}^{n} v^{par_i}(r_c)$) have to be *disjoint* – i.e. no two parts of the same type or subtype can participate in one relation. The main semantics of a structural relation with respect to the kb-pd process is: when the aggregate exists in the current partial configuration, the related parts also have to exist according to the provided cardinalities. For the other way round, i.e. if a part exists, what happens with the aggregate. A structural relation can have different characteristics, these are *aggregation*, *composition* and *set-composition*:
  Aggregation An aggregation defines a structural relation where for the existing aggregate the parts have to be instantiated according to their cardinalities. But if one part exists the aggregate has not to be existent. The existence of a part is a *necessary* condition for the aggregate but not a *sufficient* one.
  Composition A composition is a stronger form of an aggregation. In this case the parts cannot exist without the corresponding composite. This means that relations from this type are processed in both directions – i.e. the composite also has to be instantiated when one of the parts exists. The existence of a part is a *necessary* and *sufficient* condition for the aggregate.
  Set-Composition A set-composition is a special form of a composition. Here the composite is only instantiated when all of its parts exist.
– Restrictions between properties of arbitrary concepts are expressed by *constraints*. During the kb-pd process constraints ensure a consistent labeling of the properties, where *consistent* means that for each value of a property, a value for all related property can be computed – i.e. values of properties participating in a constraint relation can be computed by the constraint. Thus, constraints can be used to describe and ensure interrelations in the resulting configuration description, e.g. for describing compatibility (requires, excludes, etc.). Constraints *restrict* value ranges of properties, i.e. they do not extend those ranges, to ensure the basic reasoning mechanism.

Conceptual models can be used to describe product entities like features, hardware components and software modules. An example concept definition is given in Figure 1. `Parking Assistance` is a concept derived from the superconcept `Application` and is $appliaction - of$ a `CPS System`. The value range $v(has - sensors_{Sensor})$ is specified by the $v^{type}(has–sensors_{supervision}) = [Sensor\ 2\ 4]$ with $s^{type} = Sensor$, $s^{min} = 2$, and $s^{max} = 8$ and two $v^{par_i}(has - sensors_{supervision})$: $[Front\ 2\ 4]$ and $[Rear\ 0\ 4]$. This means that a `Parking Assistance` has at least two and at most 4 `Front Sensors` and optionally up to 4 `Rear Sensors`.

Concept

Name: `Parking Assistance`

Superconcept: `Application`

Parameters:

```
Range [10 25]
```

Relations:

```
application-of CPS-System
has-sensors
  [Sensor 2 8] :=
  [Front 2 4]
  [Rear  0 4]
```

Fig. 1. A concept definition with parameters, specialization and decomposition relations.

Procedural model A *procedural model* describes the order in which configuration decisions are processed. It mainly consists of a description of *strategies*. A strategy *focuses* on a specific part of the conceptual model. E.g. a strategy can focus on features, on software components or on the system as a whole. Furthermore, conflict resolution knowledge is used for resolving *conflict situations*[4] (e.g. by introducing explicit backtracking points).

The order of configuration decisions is not significant with respect to the solution. This means, when the same value is chosen for each configuration step in two different ordered configuration procedures with same decisions this does not affect the configuration solution. Thus, this kind of knowledge is not important for further discussion on inferencing, because it influences only the *order* of the inferences made, not the *result*.

3.2 Consistency of the Model

After modeling, besides syntactical checks the following consistency checks can be performed on the conceptual model:

– For the specialization relation between two concepts $sub \rightarrow sup$ the subset relation for each property must be fulfilled:

[4] A conflict is defined as a state in which the partial configuration is not sound with respect to the configuration model and / or the task specification.

$$sub \subset sup \Leftrightarrow \forall p \in Prop(sup) \;\; if \; \exists p' \in Prop(sub) \\ \{id(p') = id(p) \Rightarrow v(p') \subseteq v(p)\} \quad (1)$$

In the subconcept not every property of the superconcept must be defined. In this case the property of the superconcept is inherited, thus, it hold also for the subconcept.

The subset relation for all parameter types (integer, float, sets and ranges) is well defined in commonly known math – for relation descriptors it is defined in Equation 2. There, the number restrictions and the concept descriptions of the parts are tested. If a structural relation with a specific id is specified in both, the superconcept sup and the subconcept sub, the relation descriptor of the subconcept $v(r_{sub})$ has to be a subset of the corresponding relation description in the superset $v(r_{sup})$. This is defined as follows:

$$v(r_{sub}) \subseteq v(r_{sup}) \Leftrightarrow \\ v^{type}(r_{sub}) \subseteq v^{type}(r_{sup}) \wedge \forall s_i \in parts(r_{sup}), \\ \forall t_j \in parts(r_{sub}) \; with \; t_j^{type} \subseteq s_i^{type} \\ \{\sum_{j=1}^{n} t_j^{min} \geq s_i^{min} \wedge \sum_{j=1}^{n} t_j^{max} \leq s_i^{max}\} \quad (2)$$

This means, if a relation is defined in both, the superconcept sup and the subconcept sub, first the common type of the superconcept relation description $v^{type}(r_{sup})$ has to be a superset of the subconcept relation description $v^{type}(r_{sub})$. Furthermore, the sum of all number restrictions (t^{min} and t^{max}) specified in the subconcept that match a certain specifier of the superconcept have to be in the $\geq$ (for minimum) and $\leq$ (for maximum) relation as indicated in Equation 2. All parts of one relation description in a concept have to be disjoint. However, two concepts can be related via multiple distinct relations.

A specialization relation can be seen as a shortcut, which allows the factorization of common information in a superconcept. If a property is specified in both, the superconcept and a related subconcept, the value of this property (i.e. the object descriptor) of the subconcept has to be a subset of the related property in the superconcept. With the specialization relation a *specialization hierarchy* is constructed.

– For the structural relation in the aggregate a and for each part p_i a relation is specified. In the aggregate a $has{-}related{-}concepts$ relation has to be specified with a relation description describing the parts (for Figure 1 $has{-}sensors$ is defined as a $has{-}related{-}concepts$ relations (not shown)) and in each part a $related{-}to$ relation is specified, which refers to the name of the aggregate it is related to ($id(a)$).

– Consistency according to constraints is determined by boundary checks: a constraint only *reduces* value ranges specified in the conceptual model. Furthermore, each constraint is related to some properties of concepts by *constraint variables*. Those are restricted (*constrained*) by the constraint – i.e. the related value ranges are reduced. Thus, by invoking each constraint separately with the value ranges taken from the concept definitions, the result of each constraint variable has to be equal to or a subset of the related properties defined in the conceptual model. If a superset or the empty set is computed for some constraint variable, the constraint is not consistent with the conceptual model.

3.3 Task Specification

The task specification describes the configuration goal – i.e. the features, components and restrictions a product must have. For this, a set of goal concepts GC is pre-defined in the configuration model.

More *complex specifications* can include further details about the desired configuration solution [12, 13]. Such additional specifications can be the integration of already existing (partial) configurations, the selection of additional goal concepts or the specification of concept properties. Examples for *complex task specifications* might be the selection of a `Parking Assistance` and a set of Front `Sensors` or the selection of a `Sensor` with a given value for the attribute `Range`.

4 Knowledge-based Product Derivation

In the previous section we have shown how to model variability for product derivation and how this configuration model is kept consistent. In this section we proceed with describing how such configuration models can be used for our knowledge-based product derivation approach.

4.1 Configuration Steps and Related Reasonings

To distinguish between the reusable conceptual model and a configuration task, partial configurations consisting of concept instances are used for representing the current state of the configuration for a specific product derivation. Initially the concept instances i_c are generated according to the task specification and contain the parameter value ranges $v(p_c)$ and relation descriptions $v(r_c)$ of the related concept c. Those value ranges are successively restricted during the kb-pd process.

In general, a configuration step describes "setting a property of a concept instance to a more specific value". Each subset of the current value range is a more specific value. Four different types of configuration steps are supported. For each configuration step a *new value* is determined by asking the user or using computational methods like functions or default values. For each configuration step this new value has different impacts:

Parameterization A *parameter* is an attribute of a concept instance - i.e. a tuple consisting of a name and a value. Thus, a parameterization step is setting the attribute value such that the new value is more specific than before (i.e. a subset of the previous value range).

Specialization Each concept instance belongs to one concept definition determining the *concept type*. With a *specialization*, the concept type of a concept instance is changed to a more specific concept type (in a lower level of the specialization hierarchy). Typically more specific concepts contain more specific and new attribute definitions, those are *inferred* when a specialization is performed. Furthermore, properties which are only specified in the superconcept and not in the subconcept are inherited from the superconcept. This inference is based on modus ponens:

sub	chosen in the specialization step
$sub \rightarrow sup$	given in the model
sup	inferred by modus ponens.

A further inference step concerning specialization is given by *automatic specialization*. If all property values of a concept instance i_c of type c are set to subsets of related properties of a concept c' (i.e. $v(p_{c'}) \subset v(p_c) \; \forall \, p_{c'} \in Prop(c'), \; p_c \in Prop(c)$) and c' is a subconcept of c (i.e. $c' \rightarrow c$), and there is no such $c'' \rightarrow c$ for which $v(p_{c''}) \subset v(p_c) \; \forall \, p_{c''} \in Prop(c''), \; p_c \in Prop(c)$ holds, then the concept type of i_c is automatically changed from c to c'. This means, the instance i_c is automatically specialized to be an instance of c'.

Decomposition Configuring structural relations is done by the configuration step called *decomposition*. The relation description as it is described in the concept is used as a starting point for the new value. Existing instances are compared with this new value for using them in the decomposition. If no such instances exist they are instantiated and included in the current partial configuration. Thus, in a decomposition step the parts are generated according to their cardinalities – if they are not already present in the partial configuration. The relations between the aggregate and its new parts are established in the instances.

Integration In an *integration* step, an already existing concept instance is integrated into the appropriate aggregate. The new value is either given by an instance representing the aggregate or by a concept. In the last case it is necessary to instantiate this concept, i.e. a new instance for representing the aggregate is generated. Like in the decomposition step here relations between the part and the (probably new) aggregate are established, too.

4.2 Generating the Initial Configuration

When selecting a concept c from the set of goal concepts GC as a task specification, a *subtree* of instances is generated via structural relations. This subtree includes instances of all mandatory parts of the goal concept c and its successors, which can be transitively reached via structural relations. These instances build up the *initial configuration*. The optional parts are included if they are selected by the user or through automated mechanisms like taxonomic inferences or constraints (see decomposition step in Section 4.1).

For instance the concept `Parking Assistance` from Figure 1 can be chosen as a task specification. In this case, the initial configuration also comprises the mandatory `Front Sensors`.

When giving *several* concepts c_i from the set of goal concepts GC the transitive closure is generated from *all of them*. However, instances of c_i must not necessarily be connected to each other. This is done in the integration step described in Seciont 4.1.

4.3 The Configuration Cycle

The kb-pd process is cyclic; for each non-terminal property taken from the conceptual model a new value is determined, introduced in the partial configuration, possible inferences are made and potential conflicts are handled (see [14]). After this, optionally *global mechanisms* are executed for inferring system-wide impacts of the new value, i.e. several concept instances can be affected by single configuration steps. With global mechanisms it is possible to include external mechanisms, which do not use the configuration model. However, they can be used for verifying the current partial configuratiuon. Examples are *compiling the configured software components* or *simulating the product's behavior* in an external simulation module.

An important global mechanism is the constraint mechanism [15]. With constraints, relations between the properties of arbitrary concepts can be specified in the conceptual model. Because properties can be referred to in multiple constraint relations, a constraint *net* is generated. When a property is set by some configuration step, the constraint net is *propagated* and new values for the related properties are recursively computed.

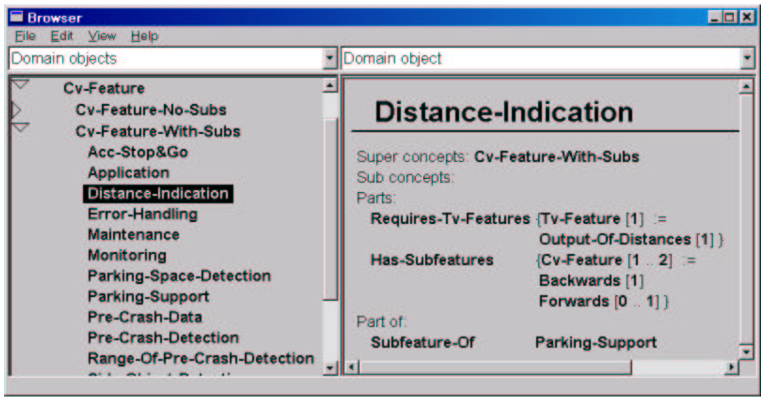
5 A Case Study

For approving the applicability of the previously described logical reasoning methods in product derivation we use the configuration tool KONWERK [16] (which deals as a blueprint for the commercial tool EngCon [17]) and model a feature tree as the input knowledge base. The feature model, which describes features of the CPS domain (see Section 2) [18], was given as a DOORS[5] model by Robert Bosch GmbH[6]. These tools provide a modeling language which has the semantics described in Section 3 and Section 4. This language was used to model the CPS feature model. Attributes of and relations between features like *optional*, *requires*, *excludes* and *recommended* (see [3]) are modeled.

The feature model consists of approximately 170 features with 100 relations and was divided into two views, a customer-related view describing product capabilities in customer-understandable terms and a technical view describing product attributes necessary for realizing these product capabilities. In Figure 2 an example for modeling a requires relation between a customer view feature and a technical view feature is given ($Requires–Tv–Features$) as well as the relation $has–subfeature$ between customer view features. Furthermore a part of a specialization hierarchie is shown on the left.

By using the configuration tool it is directly possible to automatically derive a description of all necessary product components and their attributes when a set of needed product capabilities is given. This is depicted in Figure 3 where the user decides step 1 and 2 and thus selects $ACC{-}Stop\&Go$ as an application. By using the model, the system infers further required customer features and technical features. The white boxed features indicate further decisions that have to be made.

[5] http://www.telelogic.com
[6] http://www.bosch.de



```
Cv-Feature
  Cv-Feature-No-Subs
  Cv-Feature-With-Subs
    Acc-Stop&Go
    Application
    Distance-Indication
    Error-Handling
    Maintenance
    Monitoring
    Parking-Space-Detection
    Parking-Support
    Pre-Crash-Data
    Pre-Crash-Detection
    Range-Of-Pre-Crash-Detection
```

Distance-Indication

```
Super concepts: Cv-Feature-With-Subs
Sub concepts:
Parts:
  Requires-Tv-Features {Tv-Feature [1] :=
                        Output-Of-Distances [1] }
  Has-Subfeatures      {Cv-Feature [1 .. 2] :=
                        Backwards [1]
                        Forwards [0 .. 1] }
Part of:
  Subfeature-Of         Parking-Support
```

Fig. 2. Relations Example (Screenshot of the tool KONWERK.)

6 Related Work

In [10], modeling of configurable products is introduced which is done with a similar language as the one described in this paper. However, we focus on reusing methods from structure-based configuration that can be used for product derivation and where such a model is given. We describe how feature models can be used for product derivation, similar to the FODA approach [3] that emphasizes on the the modeling aspect. With our case study it is shown how such a feature model can be mapped to a configuration language and used for product derivation.

7 Conclusion

The product derivation process in variation-rich domains is a difficult task when a large number of components and interrelations between those are present. In this paper, we show how a model consisting of components and features can be used to support the product derivation process by automatically computing interrelations of user-selected decisions. A precondition for this inference mechanisms is a logically consistent model. We outline a representation language for modeling components by means of concepts, properties (parameters and structural relations), a strict, subset-based specialization relation, and constraint relations. Given such a model diverse inference steps can be computed automatically for getting a description of software components that together build up a product. This description can be used in subsequent steps for gathering the appropriate software components (e.g. libraries, files).

The basic reasoning mechanisms are implemented in the systems KONWERK [16] and EngCon [7] and the use of those tools are shown with a feature model conmprising multiple levels of abstraction.

How this approach can be further applied to software-intensive systems is currently examined in the ConIPF project [19, 20].

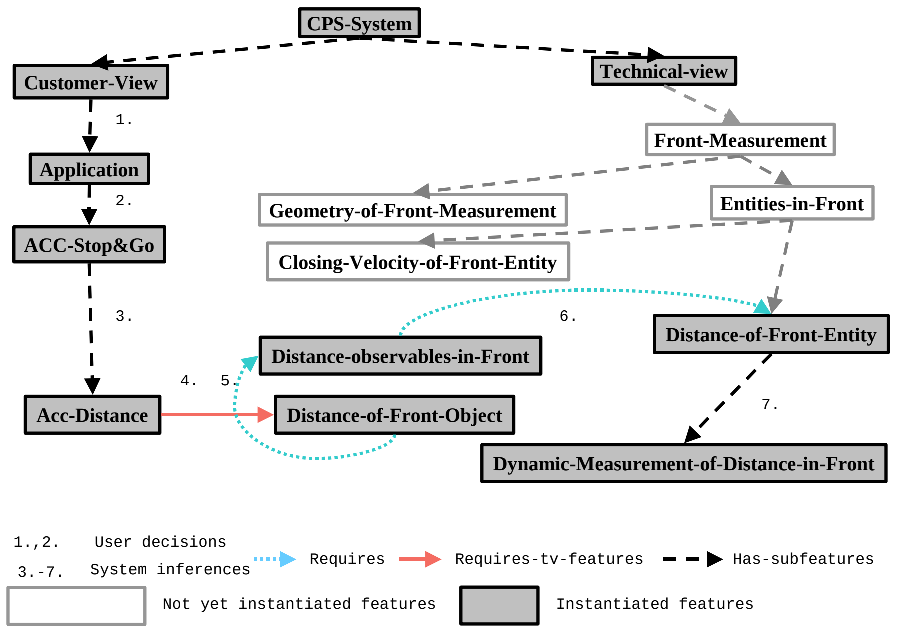


Fig. 3. Example Reasoning Path

Acknowledgments

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

References

1. Jacobsen, I., Griss, M., Jonssen, P.: Software Reuse: Architecture, Process and Organization for Business Success. ACM Press / Addison-Wesley (1997)
2. Hallsteinsen, S., Paci, M.: Experiences in Software Evolution and Reuse. Springer Verlag (1997)
3. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021 (1990)
4. Soininen, T., Tiihonen, J., Männistö, T., Sulonen, R.: Towards a General Ontology of Configuration. Artificial Intelligence for Engineering Design, Analysis and Manufacturing (1998), 12 (1998) 357–372
5. Stumptner, M.: An Overview of Knowledge-based Configuration. AI Communications **10(2)** (1997) 111–126
6. Günter, A.: Wissensbasiertes Konfigurieren. Infix, St. Augustin (1995)
7. Arlt, V., Günter, A., Hollmann, O., Wagner, T., Hotz, L.: EngCon - Engineering & Configuration. In: Proc. of AAAI-99 Workshop on Configuration, Orlando, Florida (1999)

8. Felfernig, A., Friedrich, G.E., Jannach, D.: UML as Domain Specific Language for the Construction of Knowledge-based Configuration Systems. International Journal of Software Engineering and Knowledge Engineering **10** (2000) 449–469
9. Thiel, S., Ferber, S., Fischer, T., Hein, A., Schlick, M.: A Case Study in Applying a Product Line Approach for Car Periphery Supervision Systems. In: Proceedings of In-Vehicle Software 2001 (SP-1587), Detroit, Michigan, USA (2001) 43–55
10. Männistö, T., Soininen, T., Sulonen, R.: Modeling Configurable Products and Software Product Families. In: Proc. of the International Joint Conference on Artificial Intelligence (IJCAI-2001) - Workshop on Configuration, Seattle, USA (2001)
11. Krebs, T., Hotz, L., Günter, A.: Knowledge-based Configuration for Configuring Combined Hardware/Software Systems. In Sauer, J., ed.: Proc. of 16. Workshop, Planen, Scheduling und Konfigurieren, Entwerfen (PuK2002), Freiburg, Germany (2002)
12. Krebs, T., Hotz, L.: Needed Expressiveness for Representing Features and Customer Requirements. In Riebisch, M., Coplien, J., Streitferdt, D., eds.: Proc. of ECOOP 2003 - Workshop on Modeling Variability for Object-Oriented Product Lines, Darmstadt, Germany (2003)
13. Thäringen, M.: Wissensbasierte Erfassung von Anforderungen. In Günter, A., ed.: Wissensbasiertes Konfigurieren. Infix (1995)
14. Hotz, L., Krebs, T.: Supporting the product derivation process with a knowledge-based approach. In: Proc. of Software Variability Management Workshop at ICSE 2003, Portland, Oregon, USA (2003)
15. Guesgen, H.W., Hertzberg, J.: Some Fundamental Properties of Local Constraint Propagation. Artificial Intelligence Journal **36** (1988) 237–247
16. Günter, A., Hotz, L.: KONWERK - A Domain Independent Configuration Tool. Configuration Papers from the AAAI Workshop (1999) 10–19
17. Ranze, K., Scholz, T., Wagner, T., Günter, A., Herzog, O., Hollmann, O., Schlieder, C., Arlt, V.: A Structure-based Configuration Tool: Drive Solution Designer DSD. 14. Conf. Innovative Applications of AI (2002)
18. Hein, A., MacGregor, J., Thiel, S.: Configuring Software Product Line Features. In: Proc. of ECOOP 2001 - Workshop on Feature Interaction in Composed systems, Budapest, Hungary (2001)
19. Hotz, L., Günter, A.: Using Knowledge-based Configuration for Configuring Software? In: Proc. of the Configuration Workshop on 15th European Conference on Artificial Intelligence (ECAI-2002), Lyon, France (2002) 63–65
20. Hein, A., MacGregor, J.: Managing Variability with Configuration Techniques. In: Proc. of the Workshop on Software Variability Management at the International Conference on Software Engineering, Portland, Oregon, USA (2003)