A Knowledge-based Product Derivation Process and some Ideas how to Integrate Product Development

(Position paper)

Lothar Hotz and Andreas Günter
HITeC c/o Fachbereich Informatik
Universität Hamburg
Hamburg, Germany 22527
Email: hotz@informatik.uni-hamburg.de

Thorsten Krebs
Fachbereich Informatik
Universität Hamburg
Hamburg, Germany 22527
Email: krebs@informatik.uni-hamburg.de

***Abstract*— In this position paper, a product derivation process is described, which is based on specifications of known customer requirements, features, artifacts in a knowledge base. In such a knowledge base a model about all kinds of variability of a combined software/hardware systems are represented by using a logical-based representation language. Having such a language, a machinery which interprets the model is defined and actively supports the product derivation process e.g. by handling dependencies between features, customer requirements, and artifacts. Because the adaptation and new development of artifacts is a basic task during the derivation process where a product for a specific customer is developed, the evolution task is integrated in the proposed knowledge-based derivation process.**

I. Introduction

The product line approach makes the distinction between a domain engineering part, where a common platform for an arbitrary number of products is designed and realized, and an application specific engineering part, where a customer product is derived (*product derivation process*) [1], [3]. In this position paper, a product derivation process which includes both the selection and assembling of artifacts out of a platform and their adaptation, modification, and new development for customer specific requirements is presented.[1]

The main underlying assumption is based on the existence of a descriptive model for representing already developed artifacts and their relations to features and customer requirements as well as the underlying architectural structure with its variations. All kinds of variability are represented (described) in such a model. Thus, variability is made explicit while the realization of the variability in the source code is still separate. This model is called *configuration model*. It is specified in a *knowledge base*. Thus, we speak of a *knowledge-based* product derivation process (*kb-pd-process*). Furthermore, it is assumed, that such a model is necessary to manage the increasing amount of variability in software-based products. Such a configuration model can be used for automatically configuring technical systems, where "configuring" means selecting, parameterizing, constraining, decomposing, specializing, and integrating components of diverse types (e.g. features, hardware, software, documents, etc.).

A configuration model describes all kinds of variability in a software system. Thus, it describes all potentially derivable products. But this is done on a descriptive level: when using a configuration model with an inference engine, only a description of a product is derived, not the product itself. But it is intended to use the description for collecting the necessary source code modules and realizing (implementing, loading, compiling etc.) the product in a straight forward manner. Furthermore, a configuration model is *not* a model to be used for *implementing* a software module, e.g. it does not describe classes for an object-oriented implementation.

In the following, we first describe some distinct levels of abstraction which we have to deal with when describing system entities (Section II). In Section III, we present the language entities as well as their interplay in the product derivation process. Evolution aspects are included in Section IV. A short discussion of some related work is given in Section V.

II. Levels of abstraction

We can identify three kinds of work to be done on distinct levels of abstraction for exploring a knowledge-based product derivation process:

1) **Language for specifying the knowledge base – What is used for modeling?**
   This level describes what can be used for modeling the general aspects of the process and the domain specific part. This is done by specifying a language, that can be used to describe the necessary knowledge. Furthermore, a machinery (inference engine) for interpreting this description is specified and realized in a tool. Basic ingredients of the language are concepts, relations between concepts, procedural knowledge and a specific task description (see [7], [9] for an example of such a language and a suitable tool). Entities of this language are further described in Section III.
2) **Aspects of the process – What are the general ingredients of a product derivation process?**

[1] We only consider engineering aspects of the process, we exclude economical aspects. As roles we simply see a team of software developers, which have to do both: developing a commonly used platform for all products and customer specific products.

On this level, general aspects that have to be modeled for engineering and developing products are specified. This level determines, which entities for the kb-pd-process have to be described. This is intended to be a description for a number of kb-pd-processes in distinct business units or companies, ideally for development of combined hardware/software systems in general. The description of a *specific* domain is done on the next level. Specification is done on a textual basis as well as on a model basis by using the language.
Following aspects of the kb-pd-process are currently taken into account:

- **Customer requirements:** A description of known and anticipated requirements expressed in terms which can be understood by the customer.
- **Features:** A description of the facilities of the system and its artifacts.
- **Artifacts:** A description of the hardware, software components and textual documentations to be used in products.
- **Phases of the process:** A description of general phases of the process, e.g. "determine customer requirements", "select appropriate features", "select and adapt necessary artifacts".
- **Reference configurations:** A description of typical combinations of artifacts (cases), which can be enhanced or modified for a specific product.

For each aspect an *upper model* with e.g. decompositions (e.g. sub-features) and relations of aspects is expressed. The upper model describes common parts of domain specific models. Upper models are used to facilitate the domain specific modeling. An example of an upper model is given in Figure 1. Two different views on features (i.e. customer-view (`cv-feature`) and technical-view (`tv-feature`)) are shown. Both specialize to a concept which has sub-features and one which doesn't (`cv-no-subs`, `cv-with-subs`). The dotted arrows indicate places where the domain specific models come in. Lines indicate specialization relations and arrows decomposition relations. This example shows how conceptual work done in [5], [10], [11], [16] can be used for specifying an upper model, which in turn can be used for automatic product derivation.
Each aspect of the process is modeled by using the language. Thus, it is described how e.g. customer requirements and their relations can be represented by using concepts and concept relations. In this paper, we do not further elaborate on this topic.

3) **Domain specific level – What is modeled for a specific domain?**
On this level a domain specific model is specified by using the language and the upper model. By interpreting the model with a machinery (given by a tool), this model is used for performing the process. For developing software modules (i.e. on a file, source code, developer model level) development tools and software management tools are integrated. In this paper, we do not further elaborate on this topic.

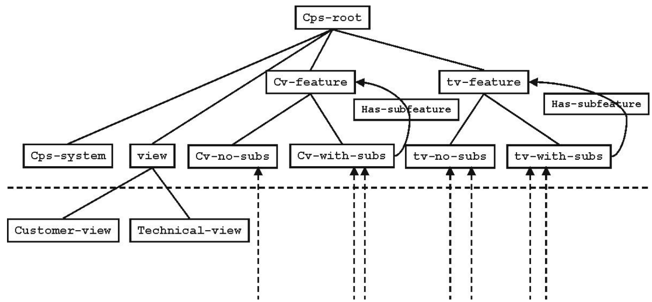

Fig. 1. Example of an upper model

III. Entities of the Knowledge Based Model

Basic entities of the model and the process are listed as follows:

1) A **concept model** for describing concepts by using names, parameters and relations between parameters and concepts. Main relations are decomposition relations, specialization relations, and n-ary relations between parameters of arbitrary concepts expressed by constraints. Such concept models can be used to describe properties and entities of products like features, customer requirements, hardware components, and software modules.
2) **Procedural knowledge** mainly consists of a description of strategies. A strategy focuses on a specific part of the concept model. E.g. a strategy focuses on features, another one on customer requirements and a next one on software components, or on the system as a whole. Furthermore conflict resolution knowledge which is used for resolving a conflict (e.g. by introducing explicit backtracking points) is described.
3) A **task specification** which describes a priori known facts, a specific product has to fulfill.

Strategies are performed in *phases*. In each phase one strategy is used, which focuses on a specific part of the model. After selecting this part, in a phase all necessary decisions (i.e. *configuration steps*) are determined by looking at the model. Each configuration step represents *one* decision, e.g. the setting of a parameter value, or processing a decomposition relation. Possible *configuration steps* are collected in an *agenda*, which can be sorted in a specific order, e.g. first decomposing the architecture in parts, then selecting appropriate components, and then parameterizing them. Decisions can be made by using distinct kinds of methods including automatic or manual ones. Each decision is computed by a *value determination method*, which yields to a specific value representing the decision. Examples for value determination methods are: "ask the user", "take a value of the concept model" or "invoke a given function". Thus, in a configuration step the decisions to be made are described and after applying some kind of value

determination method the resulting value is stored in the *current partial configuration*. A partial configuration represents all decisions made so far and their implications, which are drawn by the mechanisms described in the following.

In a cyclic practice, after each configuration step more global (i.e. systemwide) mechanisms are (optionally) executed. Examples are:

- **Constraint propagation:** For computing inferences followed by a decision and for validating the made decisions, constraints defined in the knowledge model (i.e. constraints represent relations between parameters of concepts) are propagated, based on some kind of constraint propagation mechanism.
- **External mechanisms:** For performing an external method, which does not use the concept model but only the currently configured partial configuration external techniques can be applied. Examples are:
  - simulation techniques: a simulation model is derived from the partial configuration and a separated module (like matlab) is called for this task. Some specific kind of simulation in the area of software product derivation is "compiling the source files".
  - optimization techniques: the current partial configuration is used to compute optimal values for some parameters of the configuration.
- **Further logical inferences:** Methods, which perform logical inferences that are not performed using the decision process but use the concept model, can be invoked (e.g. taxonomic inferencing, description logic etc.).

The objective of global mechanisms is to compute values for not yet fixed decisions or to validate the already made decisions. Those mechanisms (if more than one is present) are processed in an arbitrary order but repeated until no new values are computed by those mechanisms, i.e. until a fixed point is reached. If this validation is not successful or the computed value for a parameter is the empty set, a *conflict* is detected. An example would be, if the compilation of the source files fails. A conflict means that the task description, the subsequent decisions made by the user, and their logical impacts are not consistent with the model. For resolving a conflict, diverse kinds of *conflict resolution methods* (e.g. backtracking) can be applied to make other user-based decisions (see [9]). On the other side, one could also try to change the model, because if a conflict is detected, with the given model it is not possible to fulfill the given task descriptions and user needs. This gives raise to evolution, i.e. to modify or newly develope artifacts and include them in the model, so that the needs can be fulfilled (see Section IV).

Summarizing the kb-pd-process performs the following (slightly simplified) cycle:

Until no more strategy is found:

1) Select a strategy
2) Compute the agenda according to the focus
3) Until the agenda is empty or a termination criteria of the strategy is satisfied:
   - Select an agenda entry
   - Perform a value determination method
   - (Optionally) execute the global mechanisms
   - If a conflict occurs, evaluate conflict resolution knowledge.

IV. Including evolution aspects in the process

Above a well-known configuration process is described (see [4], [6]). The changing of artifacts and further development of new components (i.e. *evolution*) can be included in this process as described in the following subsections. The aspect of evolution can be seen as a kind of *innovative configuration*. We see innovative configuration not as an absolute term but as a relative one – relative to a model. A model describes a set of configurations which can be configured by using it. Innovation related to this model is given, if the configuration process computes a configuration which does not belong to this set. For supplying a product derivation process where evolution of artifacts are a basic task, we expect to apply methods known in innovative configuration to be used. A survey on innovative configuration is given in [9], [12].

A. Points of evolution

Following situations which come up in the process described in Section III indicate the necessity for evolution:

1) Pro-active, foreseen evolution, more general models: Instead of narrowing the model, broader value ranges for parameters and relations can be modeled a priori. For example, the sub-models describing customer requirements or features can represent more facilities than the underlying artifacts can realize. If during the derivation process such a feature is selected by the task description or inferred by the machinery, it gives raise to evolution of an artifact.
2) Conflicts which cannot be resolved by backtracking, i.e. by using the current model, indicate places where evolution can take place. For example, if two artifacts are chosen which are incompatible, a resolution of such a conflict would be to develop a new compatible artifact and include it into the model.
3) Points set by the user: Instead of selecting a value at a given point, the evolution of the model can be started by the developer for integrating a new or modified artifact in the partial configuration. Another example is given when the user does not accept the automatically made decisions. Thus, an evolution process is explicitly started by the user to change the model for making another decision than the model indicates. Thus, evolution as a kind of value determination method is introduced.
4) A further point is given when evolution is seen as a further global mechanism. Thus, it is included after a decision is made. Some conditions are tested on the partial configuration when evolution should be started. One trivial condition is given when the user does not accept the automatically made inferences. Thus, transparency must be given to make such a decision. If the evolution changes existing descriptions, the partial configuration must be adapted and the other global mechanisms must be invoked to find a new fixed point.

B. Evolve the model

All dependencies of the new concept (features, artifacts, customer requirements) to existing ones must be specified. Having a model, the context where a new concept will be included, can be computed on the base of the model. For instance, the related constraints of an depending aggregate or a part-of decomposition hierarchy can be presented to the developer for considering during the evolution of the model.

C. Supporting the evolution of features, customer requirements and artifacts by a knowledge-base approach

By analyzing the knowledge base, following information used for development, can be presented to the developer. The underlying idea is to present those parts of the model, which can be used in special development situations, to the developer.

- Present the already defined concepts with their parameters and relations.
- Present the specialization relation of all, of some selected or of some depending concepts. In the last case subgraphs, which describe a specialization context of a given concept are computed, e.g. the path to the root concept with direct successors of each node.
- Present the decomposition relation of a given relation of all, of some selected or of some depending concepts. In the last case subgraphs which describe the decomposition context of a given concept are computed, e.g. all aggregates, which the concept are part-of and all transitive parts which the concept has.
- Given a concept, present all concepts which are in relation to it by analyzing the constraints, i.e. also a subgraph is computed. Because constraints relate parameters of concepts the subgraph presents not only concepts but also relations between parameters.
- Given a concept, present all strategies where a parameter or relation of the concept is configured.
- Given a new concept description (with parameters and relations), compute a place in the specialization hierarchy for putting the concept into.

Knowledge modeling can be seen as a specific kind of evolution. If no given model exists, knowledge modeling is an evolution of the always given upper model. The mentioned services can be used for bringing up the first model of the existing artifacts, features and customer requirements. Thus, by supporting the evolution task, the task of knowledge modeling is also be supported.

D. Conflict resolution with an evolved model

When the model is changed, e.g. because new artifacts are included, the changes must be consistent with the ordinary model and the already infered impacts stored in the partial configuration. What kind of resolution techniques are useful have still to be developed. One trivial approach is to start the total process again with the new model and the old taks, and make all decisions of the user automatically. Thus, test the new model with the user needs, if they are consistent. This can be done automatically, because all user inputs are stored as such in the partial configuration, only the impacts have to be computed again, based on the new model. Another approach is to start some kind of reconfigration or repair technique, which changes the partial configuration according to the new model.

E. Evolve the real components

Last but not least the new components have to be build. The new source code can be implemented by using existing tools for developing and changing software systems.

F. The kb-pd-process with the evolution task included

Summarizing the kb-pd-process where evolution is included looks like:

Until no more strategy is found:

1) Select a strategy
2) Compute the agenda according to the focus
3) Until the agenda is empty or a termination criteria of the strategy is satisfied:
    - Select an agenda entry
    - Perform a value determination method or start evolution
    - (Optionally) execute the global mechanisms, included the evolution task
    - If a conflict occurs, evaluate conflict resolution knowledge.

V. Related Work

There are some approaches which try to automate software processes [14], [15]. The main distinction to the approach proposed in this paper is the different kind of knowledge representation. Instead of using rule-based systems, which have deficiencies when used for big systems [6], [8], [17], a basic concern of the language we propose is to separate distinct types of knowledge (like conceptual knowledge for describing components and their variability and procedural knowledge for describing the process of derivation). A requirement which is e.g. not followed in [2], where information about components is mixed with information about binding times in UML diagrams. One has to distinguish the knowledge representation and the presentation of the knowledge to the user. For presenting it might be useful to mix some knowledge types at certain situations (as described in IV-C). But for maintainability and adequacy reasons it is of specific importance to separate them.

In [13] a support for human developers, which is not based on automated software processes, is proposed. E.g. representations are mainly designed for human readability instead of machine interpretation. As a promising approach, structured plain text based on XML notations are considered. Thus, the combination of formal structured knowledge and unstructured knowledge should be achieved. On the one hand XML is only a mark-up language, where the main problem is to create a document type definition, which describes the documents to be used for representing software. One could see the language described in Section III as a specification for such a DTD. Thus, in our opinion for formally describing configuration knowledge in a structured way the necessary type definitions are already known. On the other hand, if unstructured knowledge should be incorporated one should also define tools which can handle them in a more than syntactic way (e.g. similarity-based methods or data-mining techniques), to get a real benefit of those kinds of representations.

VI. CONCLUSION

Making knowledge about features, customer requirements, and artifacts explicit in a model and a tool-based usage of such a model yields to an automatic product derivation process.[2] It was shown, how such a product derivation process can be defined. Furthermore, the evolution of artifacts is introduced in the process and can be supported by using the knowledge which is explicit in the model.

REFERENCES

[1] J. Bosch, *Design & Use of Software Architectures: adopting and evolving a product line approach*, Addison-Wesley, 2000.

[2] M. Clauss, 'Generic modeling using uml extensions for variability', in *DSVL 2001*. Jyvaskylae University Printing House, Jyvaskylae, Finland, (2001).

[3] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2002.

[4] R. Cunis, A. Günter, I. Syska, H. Peters, and H. Bode, 'Plakon - an approach to domain-independent construction', in *Proc. of Second Int. Conf. on Industrial and Engineering Applications of AI and Expert Systems IEA/AIE-89*, (1989).

[5] A. Ferber, J. Haag, and J. Savolainen, 'Feature interaction and dependencies: Modeling features for re-engineering a legascy product line', in *Proc. of 2nd Software Product Line Conference (SPLC-2)*, Lecture Notes in Computer Science, pp. 235–256, San Diego, CA, USA, (August 19-23 2002). Springer Verlag.

[6] A. Günter and R. Cunis, 'Flexible control in expert systems for construction tasks', *Journal Applied Intelligence*, **2(4)**, 369–385, (1992).

[7] A. Günter and L. Hotz, 'Konwerk - a domain independent configuration tool', *Configuration Papers from the AAAI Workshop*, (1999).

[8] A. Günter and C. Kühn, 'Knowledge-based configuration - survey and future directions', in *XPS-99: Knowledge Based Systems, Proceedings 5th Biannual German Conference on Knowledge Based Systems*, ed., F. Puppe, Springer Lecture Notes in Artificial Intelligence 1570, (1999).

[9] A. Günter (Hrsg.), *Wissensbasiertes Konfigurieren*, Infix, St. Augustin, 1995. in german.

[10] 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).

[11] A. Hein, M. Schlick, and R. Vinga-Martins, 'Applying feature models in industrial settings', in *Proc. of First Software Product Line Conference - Workshop on Generative Techniques in Product Lines*, Denver, USA, (August, 29th 2000).

[12] L. Hotz and T. Vietze, 'Innovatives Konfigurieren in technischen Domänen', in *S. Biundo (Hrsg.), 9. Workshop Planen und Konfigurieren*, Kaiserslautern, Germany, (1995). DFKI Saarbrücken. in german.

[13] R. Kneuper, 'Supporting software processes using knowledge management', in *Handbook of Software Engineering and Knowledge Engineering*, volume 2, Singapore, (2002). World Scientific.

[14] L. Osterweil, 'Software processes are software too', in *Proceedings of the 9th International Conference on Software Engineering (ICSE9)*, (1987).

[15] H. D. Rombach and M. Verlage, 'Directions in software process research', in *Advances in Computers*, volume 41, (1995).

[16] M. Schlick and A. Hein, 'Knowledge engineering in software product lines', in *Proc. of ECAI 2000 - Workshop on Knowledge-Based Systems for Model-Based Engineering*, Berlin, Germany, (August, 22nd 2000).

[17] E. Soloway and al., 'Assessing the maintainabiliy of xcon-in-rime: Coping with the problem of very large rule-bases', in *Proc. of AAAI-87*, pp. 824–829, (1987).

[2] "Automatic" does of cause not mean totally automatic, task descriptions and user interactions are still possible, but logical impacts can be drawn by the inference engine.