Mapping Problem-Space to Solution-Space Features: A Feature Interaction Approach

Frans Sanen Eddy Truyen Wouter Joosen

DistriNet, Department of Computer Science,
Katholieke Universiteit Leuven
Celestijnenlaan 200A, 3001 Heverlee, Belgium
{frans.sanen,eddy.truyen,wouter.joosen}@cs.kuleuven.be

Abstract

Mapping problem-space features into solution-space features is a fundamental configuration problem in software product line engineering. A configuration problem is defined as generating the most optimal combination of software features given a requirements specification and given a set of configuration rules. Current approaches however provide little support for expressing complex configuration rules between problem and solution space that support incomplete requirements specifications. In this paper, we propose an approach to model complex configuration rules based on a generalization of the concept of problem-solution feature interactions. These are interactions between solution-space features that only arise in specific problem contexts. The use of an existing tool to support our approach is also discussed: we use the DLV answer set solver to express a particular configuration problem as a logic program whose answer set corresponds to the optimal combinations of solution-space features. We motivate and illustrate our approach with a case study in the field of managing dynamic adaptations in distributed software, where the goal is to generate an optimal protocol for accommodating a given adaptation.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed Applications; D.2.11 [*Software Engineering*]: Software Architectures

General Terms Algorithms, design, performance, reliability

Keywords Software product line engineering, configuration knowledge, problem-solution feature interactions, default logic, DLV, distributed runtime adaptation

1. Introduction

Mapping problem-space features into solution-space features is a fundamental configuration problem in software product-line engineering (SPLE) [16]. In SPLE [36], product configuration [11] often boils down to selecting the required features and subsequently instantiating a software product from a set of implementation artefacts based on configuration knowledge [15]. In other words, a configuration problem is defined as generating the most optimal combination of software features given a requirements specification and given a set of configuration rules. Current approaches however provide little support for complex mappings that support incomplete requirements specifications. A detailed account of these approaches is provided in Section 6. The need for more complex mappings such as support for n-ary configuration rules that support tolerating incomplete requirements specifications is motivated in Section 2.

In this paper, we propose an approach to model complex configuration rules based on a generalization of the concept of problem-solution feature interactions [42]. These are interactions between solution-space features that only arise in specific problem contexts. In order to precisely specify the interaction between the software features, we need to take into account specific properties of the problem domain.

Our proposed approach is based on four key principles. First of all, we separate the software system description into (1) a requirements specification, (2) solution-space features and (3) problem-solution feature interactions. Secondly, we use feature models [23] to model both the requirements specification and the solution-space features. The first feature model represents the different requirements and characteristics that are relevant for a particular end product, the second feature model represents the different key design decisions that can be applied to instantiate the resulting software. Thirdly, the problem-solution feature interactions serve as an intermediate step to map the requirements specification to the solution-space artefacts. Fourthly, we use default logic [39] to model and reason about problem-solution feature interactions. As a result, the approach supports reasoning about complex mappings from problem-space to solution-space features.

The main contribution of the paper is an approach that combines the concepts of configuration knowledge and feature interaction, and the provisioning of automated support for applying that approach in complex mappings. We therefore discuss the use of an existing tool to support our approach: the DLV [30, 1] answer set solver is used to express a particular configuration problem as a logic program whose answer set corresponds to the optimal combinations of solution-space features.

We motivate and illustrate our approach with a case study in the field of distributed runtime adaptation where the goal is to generate an optimal adaptation protocol for a given adaptation request. Distributed runtime adaptation is the problem of adapting a distributed system at runtime in a safe and coordinated way. More concretely, it concerns binding or unbinding multiple interdependent components [48] with a distributed application at runtime while preserving global state consistency [33]. Consider for instance the addition or removal of a fragmentation service in a messaging application [51] that consists of a fragmenting component at the client side that fragments large messages that exceed a certain threshold and a re-

GPCE'09, October 4–5, 2009, Denver, Colorado, USA.

assembling component at the server side which reassembles these messages again. Mainly as a consequence of the interdependence between these components, these must be added or removed together in an atomic fashion without causing message omissions.

The rest of this paper is structured as follows. Section 2 describes the running example we use throughout the remainder of the paper, motivates the need for complex mappings in configuration knowledge and sets the scope for our work. Section 3 then makes the case for problem-solution feature interactions as configuration knowledge and default logic as a conceptual frame. Section 4 discusses our approach. Subsequently, we apply the approach to our case study in Section 5. Finally, related work is discussed in Section 6 and we conclude the paper in Section 7.

2. Background

This section introduces DyReS as a case study, positions our work in a software product line context and motivates the need for complex mappings as part of the configuration knowledge.

2.1 DyReS

DyReS[1] is an adaptation support system that we have developed [51] based on the vision and principles of the NeCoMan middleware [21]. Currently, DyReS extends two industrially-used middlewares (Spring and JBoss) with runtime support for implementing distributed adaptations of a client-server based application in a coordinated and safe way. DyReS enables the developer to implement a distributed runtime adaptation in a coordinated way executing an adaptation protocol between the nodes involved in the adaptation. An adaptation protocol typically consists of a sequence of four reconfiguration tasks: installation (installing new components), finishing (bringing old components to a safe state), activation (unbinding any old and binding any new components) and removal (removing old components). Each reconfiguration task is structured as a sequence of one or more reconfiguration actions for creating/removing, binding/unbinding, starting/stopping, interrupting/resuming components or imposing a safe state on them.

The main idea behind DyReS is as follows. A client-server application typically exhibits a number of application-specific characteristics and desires some specific requirements with respect to how the adaptation protocol should coordinate the adaptation. The design of DyReS consists of a generic adaptation protocol, that works in all cases, and a set of optional customization strategies that optimize the generic adaptation protocol for a specific application based on the application-specific characteristics and requirements. Generally, an adaptation protocol that is customized towards an application its requirements and characteristics can perform more efficiently (cfr. also [20, 51]).

Application-specific requirements are quality requirements of importance, such as responsiveness, availability and reliability of the distributed application. For example, when the responsiveness of the application may not be affected by the adaptation, a temporary service disruption can not be tolerated. Obviously, these quality requirements have an influence on what will be the most optimal configuration of solution-space features.

Application-specific characteristics are an equally important part of the description of the problem space, since the application and its components may have specific characteristics that can be exploited to further optimize the resulting configuration of the adaptation protocol, or, on the contrary, forbid certain optimizations. A stateless session model of the distributed application is an example characteristic that can allow for a customization strategy of executing different reconfiguration tasks in parallel [21]. Such a customization strategy results in less service disruption and therefore satisfies the responsiveness requirement better.

In general, a customization strategy can concern different matters: composition order, alternative implementation components, sequence of execution, configuring complex components the right way, etc. In our work, we model customization strategies as incremental refinements of the generic adaptation protocol. Their implementation into reusable implementation components can be done using aspect-oriented programming [51].

2.2 Software product line engineering

In our work, we aim to develop a feature-based software product line for the design of a family of adaptation protocols as depicted in Figure 1. A specification of an adaptation protocol by a human reconfigurator is in the DyReS product line a configuration file that is directly parsed into a customized adaptation support system. A feature can be defined as a characteristic of a system that is visible to the end-user [23] or as a distinguishable characteristic that is relevant to some stakeholder [15]. Our domain analysis has resulted in a set of application-specific requirements and characteristics, i.e. problem-space features. The domain design consists of a generic adaptation protocol in combination with customization strategies, i.e. solution-space features.

We use feature models [23] to model both these problem-space and solution-space features. A feature model represents the common and variable features of a family of systems in a specific domain and the relationships between them. Features can be related hierarchically through an and/or relationship in combination with cardinality constraints to represent alternative or optional features. Crosscutting the hierarchy, features can require or exclude other features too. Although our approach does not prescribe a certain syntax to be used, we use trees[2].

Separating problem and solution space is an important trend in SPLE, e.g. the work in [32] enables the separation of concerns between product line and software variability. Software variability refers to the "ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context" [47]. Product line variability [36, 24] is specific to SPLE and describes the variation between the systems that belong to a product line in terms of properties and qualities, like features that are provided or requirements that are fulfilled. For the family of adaptation protocols in DyReS, product line variability is defined by the application-specific requirements and characteristics, while customization strategies describe the software variability.

The rest of this paper will focus on the configuration problem that is indicated by the dashed rectangle in Figure 1. If we consider the requirements and characteristics of a specific application as problem-space features and the customization strategies as solution-space features, our configuration problem boils down to generating the most optimal adaptation protocol given a requirements specification of the requirements and characteristics of the particular application.

2.3 Complex mappings

To handle the configuration problem, we adopt a generative programming (GP) view. GP is a software engineering paradigm based on modeling software system families such that, given a particular requirements specification, a highly customized and optimized intermediate or end product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge [15].

[1] DyReS stands for Dynamic Reconfiguration Support. Source code and contact information are available at http://www.cs.kuleuven.be/∼distrinet/projects/DyReS.

[2] More specifically, we use single-rooted directed acyclic graphs as defined in [44].

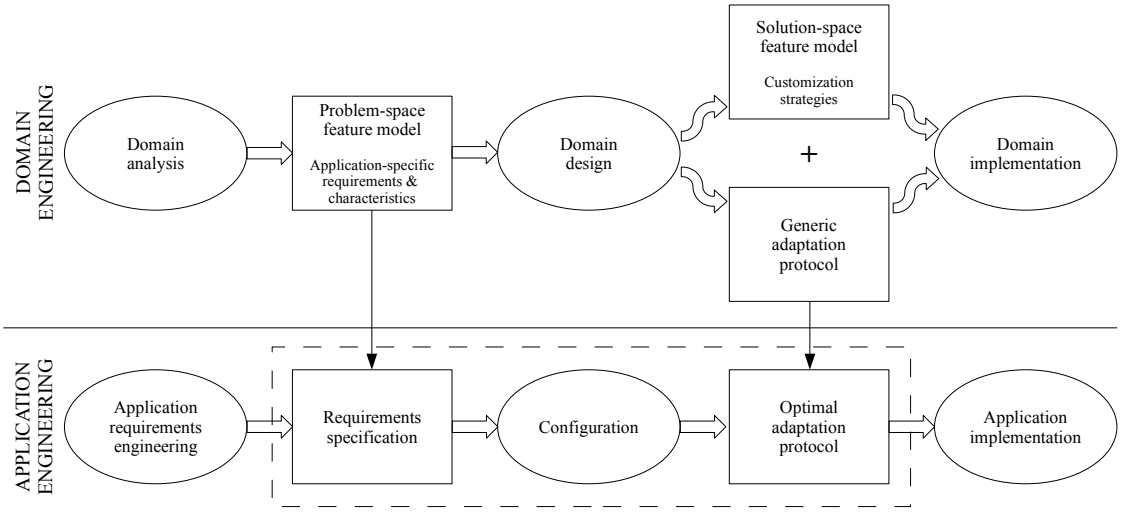

Figure 1. SPLE for the design of adaptation protocols. An oval is used to model an activity, a rectangle represents an artefact in DyReS.

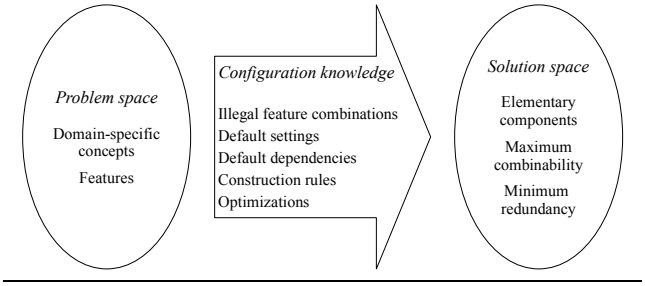

Figure 2. Elements of configuration knowledge. [15]

Figure 2 shows the different elements of configuration knowledge. As one can see, this involves illegal feature combinations in the problem or solution space, default settings, default dependencies, construction rules and optimizations. This paper is mostly concerned with the problem of (i) modeling and reasoning over construction rules while respecting feature interactions and (ii) enabling optimization while supporting defaults in the case of incomplete requirements specifications.

In our case study, configuration knowledge hence needs to support mapping a requirements specification into a correctly configured end product. Furthermore, the configuration knowledge needs to relate the requirements and characteristics of the application under reconfiguration to the most optimal set of customization strategies that can be applied to the generic adaptation protocol.

This configuration knowledge becomes quite complex in DyReS. First of all, the choice for a solution-space feature can depend on different elements from both the problem and solution space [42]. Therefore, simple one-to-one mappings do not suffice to specify complex construction rules in which the selection of a solution-space feature depends on both problem-space and solution-space features. On the contrary, we need more complex, n-ary mapping mechanisms. Secondly, incompleteness of requirements specifications [38, 43] is an important problem in DyReS. Often human reconfigurators, who are in charge of maintaining and evolving a running application, may not be able to precisely know how to instantiate the problem-space feature model of DyReS into a precise requirements specification. This is for example the case when they are not familiar with the notions involved or do not have access to the source code. Even simple one-to-one mappings from a problem-space to a solution-space feature possibly don't work if the knowledge about the problem-space feature is incomplete or non-existing to begin with. In Section 3.2, we present a concrete example of such a complex, n-ary mapping that may suffer from an incomplete requirements specification.

Hence, the proposed approach needs to offer a means for expressing complex, n-ary relationships between features that cross the problem and solution space and provides a means to handle potentially incomplete requirements specifications.

3. Feature interactions

This section motivates our approach and illustrates our point of view that complex configuration rules can be seen as an extension of the feature interaction problem. We elaborate on our rationale incrementally. First, we summarize the more traditional view on modeling and reasoning about feature interactions. Secondly, we present a concrete complex mapping between the problem and solution space in DyReS and show that this mapping can be seen as a problem-solution feature interaction. Thirdly, we define this concept and integrate it with default logic to be able to generate the most optimal adaptation protocol while using defaults in the presence of incomplete requirements specifications.

3.1 Traditional view

In our approach, we propose to extend the established feature interaction framework with problem-solution feature interactions and use this as the conceptual basis for modeling complex configuration knowledge. A feature interaction traditionally is defined as the situation where two features that work correctly in isolation (when composed with the base system) do not work correctly any longer when they are combined together and composed with the same base system [26, 12]. In other words, a feature interaction is some way in which a feature modifies or influences another feature in defining overall system behavior (taken from [55]). Note that this definition allows for both positive and negative interactions. Because interactions often are implicit in feature composition, they are difficult to understand. Therefore, a sound logic for modeling and reasoning about feature interactions is necessary, e.g. for finding a correct combination of features as in [8].

3.2 Complex mappings

We now present an example of a complex mapping that involves two solution-space features and one problem-space feature relevant in our case study that has been introduced in Section 2.1. For the sake of completeness, note that the mapping described below is only valid in the context of a replacement adaptation, but we make abstraction of this in the paper.

- **Finish before activate** This solution-space feature represents the customization strategy of executing the finishing and activation reconfiguration tasks in sequence, i.e. finishing any old components before new components become activated. As described in [42], the downside of this solution is that service disruption may occur (cfr. [51]) due to driving involved components to a quiescent or frozen state.
- **State synchronization** This solution-space feature represents another customization strategy that allows us to transfer execution state from the old to the new components.
- **Persistent** This problem-space feature is used to specify that execution state of one or more of the components involved in an adaptation is persistent (surviving more than one client request). This is the opposite of transient state.

Informally, the mapping between these three features needs to behave as follows. When *finish before activate* becomes selected, components are brought to a state where processing of all remaining client requests has completed. Therefore, all transient state (related to processing a single client request) is gone. Persistent state however survives execution of multiple client requests and may need to be transferred to new components. However, many client-server applications currently operate under a stateless session model where *persistent* state is not maintained in server components. Under these circumstances, an optimal adaptation protocol thus does not include $statesychronization$.

In other words, *finish before activate* depends on the *state sychronization* solution-space feature, but only in the specific problem context where the adaptation involves components that entail *persistent* state. Clearly, such knowledge is key to solving the configuration problem at hand and crosses both the problem and solution space. We experienced that specifying this kind of configuration knowledge precisely enough is far from trivial.

Problem-solution feature interactions Our approach proposes to use problem-solution feature interactions for expressing configuration knowledge consisting of complex, n-ary relationships between features that cross the problem and solution space. We define a problem-solution feature interaction as an interaction between two or more solution-space features that only arises based on one or more problem-space features. In order to manage problem-solution feature interactions, our approach enables modeling these interactions so we can enforce them (in case of positive interactions) or resolve them (in case of negative interactions).

Logic formulae already have been considered elsewhere as an appropriate means to model feature interactions. E.g. in [8], propositional formulae are used to model constraints where a feature requires or excludes another feature through the use of an *implies* relationship. Note that specifying the example interaction as a simple dependency between solution-space features only, e.g.

$$finish\ before\ activate \rightarrow state\ synchronization \quad (1)$$

or as a simple one-to-one mapping between a problem-space and solution-space feature, e.g.

$$persistent \rightarrow state\ synchronization \quad (2)$$

is not good enough. More specifically, *finish before activate* only depends on *state synchronization* if the application has persistent state. Indeed, an application without persistent state would benefit from an optimized adaptation protocol that skips state synchronization. Therefore, we can specify the complex mapping as follows:

$$finish\ before\ activate \wedge persistent \rightarrow state\ synchronization \quad (3)$$

3.3 Default logic

We however argue that formula (3) is not sufficient either for our purposes as listed in Section 2.3. It perfectly captures the problem-solution feature interaction in the case where we know if the components involved in the adaptation contain persistent state or not. Unfortunately, it is only realistic to assume that the requirements specification can be incomplete [38, 43] or nothing might be known about the components containing persistent state or not. Under such circumstances, the configuration knowledge still should be able to generate a complete and correct configuration of solution-space features, although maybe less optimal: we take *state synchronization* just to be sure. Hence, we need to be able to rely on default information. As a consequence, classical logic formulae (as e.g. in formula (3)) indeed are not sufficient if we want to model complex mappings such as the one in Section 3.2. In the rest of this section, we discuss the use of default logic as a means to implement this kind of configuration knowledge and reason about feature interactions [41].

Default logic has been originally proposed by Reiter as a non-monotonic logic to formalize reasoning with default assumptions [39]. It allows making plausible conjectures when faced with incomplete information and draw conclusions based upon assumptions [5]. As an intuitive example of what can be expressed, consider the well-known principle of justice in our Western culture that, in the absence of evidence to the contrary, we assume that the accused is innocent. Next, we overview both the syntactic sugar and semantics (informally) of default logic by applying it to our example interaction from the previous section. We subsequently also discuss the relevance of using default logic to support our approach.

A default theory T is a pair (W, D) consisting of a set W of logic formulae (background theory or facts of T) and a set D of default rules. The default rule explicitly representing our example interaction is presented below.

$$\frac{finish\ before\ activate\ :\ persistent}{state\ synchronization} \quad (4)$$

According to formula (4), if we know that *finish before activate* is true and *persistent* can be assumed, we can conclude *state synchronization*. The three parts of a default rule are called the prerequisite φ, justifications ψ_i and conclusion χ respectively. Hence, the general explanation of any default rule is given by "if we believe that prerequisite is true, and the justification is consistent with our current beliefs, we also believe the conclusion". In other words, given a default $\varphi : \psi_1, \psi_2, .../\chi$, its informal meaning is: if φ is known, and if it is consistent to assume $\psi_1, \psi_2, ...$ then conclude χ. It is consistent to assume ψ_i iff the negation of ψ_i is not part of the background theory W [10]. At this point, it is important to realize that classical logic is not appropriate to model this situation. The problem with such a rule (cfr. (3)) is that we have to definitely establish if there is *persistent* state involved or not (basically because of the closed world assumption [18]). As a consequence, the *state synchronization* feature never would become selected if nothing is known about persistence.

The semantics of default logic typically is defined in terms of extensions. Intuitively, an extension seeks to extend W (back-

ground theory) with "reasonable" conjectures based on the applicable default rules. More formally, a default $\varphi : \psi_1, \psi_2, .../\chi$, is applicable to a deductively closed set of formulae E iff $\varphi \in E$ and $\neg\psi_1 \notin E, \neg\psi_2 \notin E, ...$ You can think of E as the context in which φ should be known and with which ψ_i should be consistent.

Discussion on the relevance of defaults for our approach We now discuss our default rule (4) together with its meaning into more detail. Intuitively, the rule states that the $state\ synchronization$ feature needs to be selected if the $finish\ before\ activate$ feature is selected, unless there is no persistent state contained within one of the components involved in the adaptation. The default rule represents the following scenarios correctly.

- If the $finish\ before\ activate$ solution-space feature becomes selected as a result of applying the configuration knowledge and nothing is known about $persistent$ state, then the background theory W will include $finish\ before\ activate$ as a fact and say nothing about persistence. Because of default rule (4), only extensions indicating to also select the $state\ synchronization$ feature are valid ones. This is caused by $finish\ before\ activate$, the prerequisite, being true and the justification $persistent$ being not inconsistent with what is currently known. Similarly, the same conclusion will be drawn if the $persistent$ feature is selected as problem-space feature as part of the requirements specification.
- On the other hand, if the requirements specification indicates that components involved in the adaptation entail persistent state, W will include $persistent$. When there is no persistent state, W includes $\neg\ persistent$. If the latter is known, valid extensions will not include $state\ synchronization$ because default rule (4) no longer can be applied. If the former is known, the default rule remains applicable.

This way, the specific problem context on which our example interaction depends is made explicit via the justifications in default rule (4). Hence, we were able to express that the $finish\ before\ activate$ solution-space feature only requires the $state\ synchronization$ solution-space feature when the $persistent$ problem-space feature is selected as part of the requirements specification. More generally, this rule can be used to arrive at the most optimal combination of solution-space features, based on the specific problem context indicated by problem-space features.

Default rules are perfectly capable to express the more simple, intuitive binary interactions between two features: the justification part becomes true. Alternatively, default logic can be used in combination with classical logic rules, too. We refer the reader to Section 5 for concrete examples of such interactions.

4. Approach

This section summarizes our approach. Section 4.1 presents the high-level overview of the approach. Subsequently, Section 4.2 integrates the proposed approach with default logic. Finally, Section 4.3 discusses the use of DLV to support the approach.

4.1 Overview

A schematic overview highlighting the key artefacts of the proposed approach is depicted in Figure 3.

- Product line variability is modeled through a problem-space feature model. Features in this model express requirements and characteristics of a specific application.
- The instantiation of the problem-space feature model is a particular requirements specification.
- A solution-space feature model is used to model the software variability. Our solution-space features represent key design decisions that can be applied to instantiate the resulting software.
- The instantiation of the solution-space feature model models the optimal configuration of solution-space features. An optimal configuration of solution-space features satisfies the application-specific requirements and maximally exploits the characteristics of the application.
- Problem-solution feature interactions are used as a means to model the configuration knowledge. They enable us to map the problem-space features to an optimal configuration of solution-space features.

The proposed approach solves our configuration problem by enabling us to generate the most optimal instantiation of the solution-space feature model given a particular instantiation of the problem-space feature model and given a set of problem-solution feature interactions. We model and reason about the configuration knowledge by formalizing the problem-solution feature interactions in default logic.

4.2 Integration with default logic

Figure 3 also shows how each of the artefacts in our approach integrates with default logic. First of all, both the problem-space and solution-space feature model and feature interactions within these models can be represented in standard propositional or predicate logic, e.g. as proposed in [7] by Batory et al. Secondly, we use default rules to model problem-solution feature interactions, as explained in Section 3.3. The complete set of default rules gives us D. Next, we model the concrete instantiation of the problem-space feature model, i.e. a requirements specification, as our background theory W. Finally, the solution to our configuration problem, i.e. the optimal instantiation of our solution-space feature model, is given by the extension E of (W,D).

In our default rules, we distinguish between three predicates.

1. $kdd(X)$ indicates a key design decision. The parameter matches with the name of the solution-space feature that expresses the concrete key design decision.
2. $eq(X)$ defines an application-specific requirement. The parameter contains the name of the problem-space feature that matches with the concrete requirement.
3. $char(X)$ models a characteristic of the application that can be exploited. The concrete characteristic that matches with the name of the corresponding problem-space feature is included as parameter.

Below, the abstract structure of a default rule is shown. Rule (5) generalizes over the example interaction from Section 2.1 and states that $kdd(a)$ requires $kdd(b)$ if we can assume $char(k)$. In rule (6), the justification part includes an assumption w.r.t. the application-specific requirements or its negation.

$$\frac{kdd(a)\ :\ char(k)}{kdd(b)} \tag{5}$$

$$\frac{kdd(a)\ :\ \neg req(l)}{kdd(b)} \tag{6}$$

To compute an extension E, given a background theory, W, and given a set of default rules, D, we adopt answer set programming (ASP) [31]. ASP is a powerful paradigm from the field of non-monotonic reasoning [18] in artificial intelligence. It is based on the stable model [35] semantics of logic programming. An answer set solver is a program for generating stable models or answer

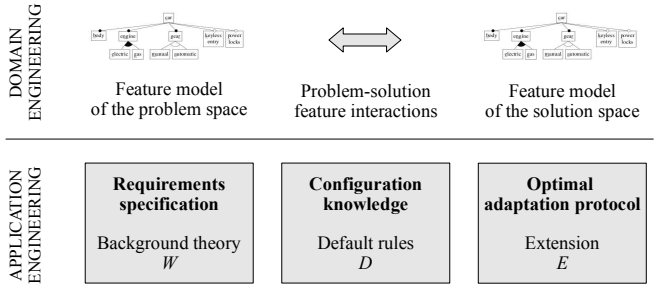

Figure 3. A schematic overview of our approach.

sets. In essence, we use ASP to express a particular configuration problem as a logic program whose answer set corresponds to the optimal combinations of solution-space features. We declaratively specify what a solution to the configuration problem should look like and leave it for the answer set solver to try to find a solution that solves the problem. The results of this computation are called models: they are consistent explanations of the world (the resulting configuration of solution-space features) as far as the solver can derive it. By default, most solvers generate all possible answer models. However, if the logic program representing the configuration problem contains inconsistencies, there simply is no model.

4.3 Supporting our approach in DLV

Our approach uses the DLV [30, 1] answer set solver to compute correct and optimal solutions for our configuration problem. Our choice for DLV as a tool is based on its systemic support for answer set programming and its ability to express disjunctions (the logical *or*), negation as failure (indicated by `not` and in contrast with true negation, which is indicated by the `-` sign) and default rules. DLV is able to implement default rules because the deductive closure of an answer set for a logic program with negation as failure is a consistent extension for the default theory of that same program, and vice versa [31]. The equivalent implementations of default rules (5) and (6) in DLV are shown below to illustrate how this can be done.

```
kdd(b) :- kdd(a), not -char(k).
kdd(b) :- kdd(a), not req(l).
```

The prerequisite and the conclusion of the default rule become the body, respectively the head of the rule in DLV. The justification is negated by negation as failure and copied in the rest of the body. Intuitively, `not -char(k)` is true if we have no information on `-char(k)` or in other words when we can assume `char(k)`, i.e. the first justification in default rule (5). Similar reasonings apply for believing the negation of `req(l)`. Note that a default without justifications matches a rule without negation as failure and that a fact can be represented by a default rule with both the prerequisites and justifications being true.

In DLV, we accomplish our goal of generating an optimal configuration of solution-space features with facts (W) and rules (D): facts are the input data and rules can be used to derive one or more solutions of the configuration problem. We already pointed out that problem-solution feature interactions represent important configuration knowledge which maps problem-space into solution-space features. In summary, the rules in our logic program express relationships between solution-space features or relationships that cross the problem and solution space. Facts, on the contrary, designate which application-specific requirements and characteristics have to be taken into account for a particular configuration problem.

5. Case study

In this section, we provide further details on DyReS, our case study of distributed runtime adaptation. We start from the motivating example introduced in Section 2 and work through the generation of the most optimal configuration of software features. For the sake of understandability, bear in mind that the requirements specification modeled via problem-space features corresponds to the application-specific requirements and characteristics of the application under reconfiguration, while the solution-space features model key design decisions that represent different strategies for customizing the generic adaptation protocol.

5.1 Expressing the application-specific requirements and characteristics

In DyReS, a feature model of 39 features is used to model both the requirements and characteristics of the application that will be adapted. A subset of this feature model is shown at the left of Figure 4 and distinguishes between three main dimensions: quality of service requirements, characteristics of the application itself and the components. A detailed description of this model already has been described elsewhere [42].

Quality of service requirements We consider responsiveness, availability and reliability to be the most relevant quality attributes as perceived by the end user during an adaptation. The service disruption tolerated, graceful service degradation tolerated and robustness features respectively concretize these requirements. An example instantiation of the problem-space feature model, i.e. a requirements specification in DLV looks as follows.

```
req(robustness).
req(small).
```

The robustness QoS feature demands from the adaptation protocol that client requests never will be rejected by the server. Furthermore, it is stated that a small service disruption is tolerated (e.g. when the application gets frozen in the course of driving components to a safe state). Clearly, these QoS requirements have an effect on how a distributed runtime adaptation should be performed. In order to know in which way an execution of a distributed adaptation affects the quality attributes of a particular application, the characteristics of the application must also be described.

Characteristics of the application and its components Characteristics that can be exploited to optimize a particular adaptation protocol are situated both on the level of the application itself and its components. An example description in DLV of component characteristics with an effect on the adaptation protocol is given below. It states that the adaptation involves an old component with persistent (cfr. our example) and externally dependent execution state. In addition, the component does not provide state transfer support on its own.

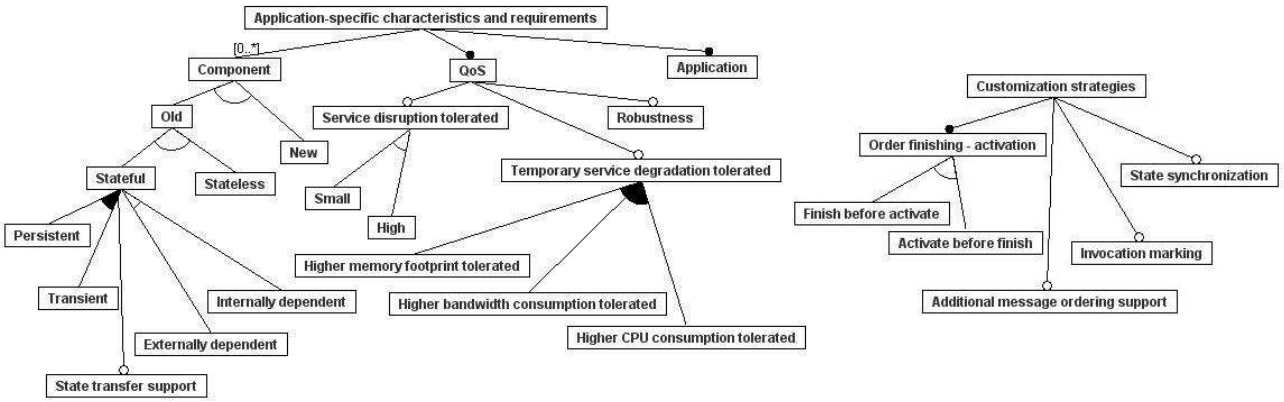

Figure 4. Feature model of application-specific requirements and characteristics (left) and customization strategies (right) in DyReS.

```
char(persistent).
char(externally_dependent).
-char(state_transfer_support).
```

5.2 Expressing the customization strategies

The different customization strategies are modeled via a second feature model of 31 features, of which a subset is depicted in Figure 4 at the right side. A detailed description of the complete diagram can be found in [42]. For now, it suffices to know that each customization strategy represents a key design decision on how to shape the concrete adaptation protocol. An important variability point of an adaptation protocol in DyReS relates to the order between finishing and activation. The downside of the generic protocol that finishes any old components before activating new ones (called *finish before activate*) is that service disruption may occur due to driving involved components to a quiescent or frozen state. By already activating new components before old ones are finished, we can benefit from a reduced service disruption. Note that this *activate before finish* customization strategy involves more than simply swapping the order in which DyReS executes the finishing and activation actions respectively [51]. We modeled this variability point as follows in DLV.

```
kdd(order_fin_act) :-
    kdd(customization_strategies).
kdd(finish_before_activate) v
kdd(activate_before_finish) :-
    kdd(order_fin_act).
-kdd(activate_before_finish) :-
    kdd(finish_before_activate).
-kdd(finish_before_activate) :-
    kdd(activate_before_finish).
```

It says that for a set of customization strategies, the order between finishing (`fin`) and activation (`act`) needs to be established. This order can be either finish before activate or activate before finish but not both. Obviously, other customization strategies can be represented in a similar fashion.

5.3 Expressing the configuration knowledge

Configuration knowledge in DyReS needs to enable us to generate a correct and the most optimal set of customization strategies given a description of the application-specific requirements and characteristics. We discuss multiple categories of configuration knowledge. First, we provide some extra examples of complex mappings in the DyReS case study. Subsequently, we discuss two special cases of such complex mappings: simple and weaker mappings. We combine the rules expressing problem-solution feature interactions with rules modeling the more simple relationships between solution-space features only. The latter can be seen as semantic constraints to which any solution for the configuration problem should adhere.

Complex mappings In our approach, complex mappings are modeled as problem-solution feature interactions. Section 3.2 already presented an example of an interaction between two solution-space features which only occurred depending on a problem-space feature. Another example of a complex mapping is illustrated by the problem-solution feature interaction and its DLV equivalent below. The details of this equivalence already have been presented in Section 4.3.

$$\frac{kdd(activate\ before\ finish)\ :\ char(ordered)}{kdd(additional\ message\ ordering\ support)} \tag{7}$$

```
kdd(additional_message_ordering_support) :-
    kdd(activate_before_finish),
    not -char(ordered).
```

This mapping expresses that if we know that activate before finish is an applicable customization strategy and if we can assume $char(ordered)$, additional message ordering support also will get selected as an applicable customization strategy.

Simple mappings We model simple, binary one-to-one mappings in which a problem-space feature directly can be mapped to a solution-space feature as a default rule with its justification being true. E.g. the default rule

$$\frac{req(robustness)\ :\ true}{kdd(finish\ before\ activate)} \tag{8}$$

is implemented in DLV as follows:

```
kdd(finish_before_activate) :-
    req(robustness).
```

The rule expresses that the robustness QoS requirement requires the finish before activate customization strategy.

Weaker mappings The last category of configuration rules involves weaker mappings. The idea of a weaker mapping is that we map problem-space into solution-space features giving priority to knowledge inferred via one of the other categories. For example,

$$\frac{req(small) \ : \ kdd(activate\ before\ finish)}{kdd(activate\ before\ finish)} \quad (9)$$

says that if only a small service disruption is tolerated, we conclude $kdd(activate\ before\ finish)$ if we can assume it. On the contrary, if the rest of our configuration knowledge leads to the deduction of $-kdd(activate\ before\ finish)$, this rule never will fire. Otherwise, we further can optimize the generic adaptation protocol according to the activate before finish customization strategy. The DLV version of the default rule implementing this weaker mapping is shown below.

```
kdd(activate_before_finish) :-
    req(small),
    not -kdd(activate_before_finish).
```

5.4 Generating the most optimal configuration

After describing the feature models for the application-specific requirements and characteristics and the different customization strategies respectively, we listed some example configuration rules. A semi-complete overview of all relevant configuration knowledge in DyReS and concrete example adaptation protocols already have been described elsewhere (in [42] and [51] respectively).

Experiments show that we are able to calculate the most optimal configuration of solution-space features, i.e. the most optimal set of customization strategies, for a given requirements specification. Our DLV implementation consists of 34 rules prescribing how the customization strategies are related to one another on the one hand and 39 rules enabling 28 mappings between problem-space and solution-space features on the other hand.

For example, suppose that we start from all the facts that have been described in Section 5.1. In other words, DLV has `req(robustness)`, `req(small)`, `char(persistent)`, etc. as part of the background theory W. W also includes the axioms implementing the solution-space feature model, e.g. as illustrated in Section 5.2. Remember that D contained the problem-solution feature interactions. DLV's first answer to one of our experiments was the following correct configuration of solution-space features (and thus a valid element of the extension set E).

```
 1  {kdd(customization strategies),kdd(adaptation_
 2  type),kdd(order_fin_act),kdd(finishing_or_not)
 3  ,kdd(activation_or_not), kdd(state_synchroniza
 4  tion),kdd(finish_before_activate),-kdd(activat
 5  e_before_finish),-kdd(invocation_marking),-kdd
 6  (additional_message_ordering_support),kdd(repl
 7  acement),kdd(finishing),kdd(activation),kdd(st
 8  ate_transfer),kdd(distributed),kdd(interrupt),
 9  kdd(impose_safe_state),kdd(coordinated),kdd(in
10  vocation_queuing),kdd(monitoring)}
```

The simple mapping in rule (8) results in `kdd(finish_before_activate)` being selected (cfr. line 4). Consequently, the complex mapping expressed in default rule (4), selects `kdd(state_synchronization)` (cfr. line 3) as another solution-space feature. Note that, although `req(small)` is true, the weaker mapping expressed in rule (9) can not be applied under these circumstances since it is no longer consistent to assume `kdd(activate_before_finish)`. We make abstraction of the other predicates but it should be clear that the set of all the predicates in the depicted DLV answer denotes a complete and correct instantiation of the solution-space feature model.

5.5 Discussion

We finally discuss some of our experiences with applying the proposed approach to the domain of distributed runtime adaptation.

First of all, combining simple and weaker mappings provides us with a way of prioritizing between configuration rules: conclusion X of a rule implementing a simple mapping can turn a rule representing a weaker mapping (with the negation of that same conclusion X) unapplicable. E.g., rule (9) no longer can be applied in our case study if rule (8) fires. However, if we have two simple mappings or two weaker mappings with contradicting conclusions, it currently is impossible to have one take precedence over the other. DLV's built in support for taking care of priorities normally can help us to accommodate this issue.

Secondly, the result of applying the configuration knowledge can result in multiple answers in the answer set for a particular configuration problem. Typically, the less is known from the requirements specification, the more configurations of solution-features are possible. Currently, our approach does not support a way to evaluate these different solutions and select one[3]. However, all generated solutions are valid and optimal w.r.t. the initial configuration problem. On the other hand, it is also possible that DLV cannot generate any solutions. If there are no answers at all, we still are unclear about how to proceed. We could for instance try to find the minimal set of application-specific characteristics and requirements that can be omitted so we can find a correct configuration of solution-space features.

Finally, there are some limitations of rule-based systems in general, such as problems concerning knowledge acquisition, consistency checking, modularity and maintenance [19]. Obviously, the severeness of these problems largely depends on the scale of the domain in which one tries to solve a particular configuration problem. In future work, we plan to integrate our approach with description logics (DL) [17], e.g. using OWL [40]. DL techniques typically support consistency checking and offer modularity mechanisms.

6. Related work

As stated in Section 2, we take the approach of Metzger et al. [32] distinguishing between product-line (problem space) variability and software (solution space) variability models as the starting point of our work. They connect both kinds of models by means of generic cross links named *x-links*. Models and x-links are formally represented so that consistency between both models can be checked as a satisfiability problem, similar to existing feature model analysis approaches [8]. As such, this work focuses on the automation of crucial consistency checks during SPLE, both at early stages of the development as well as during product line evolution. The focus of our work rather is on the automatic derivation of optimal software configurations from requirement specifications; therefore, both approaches are complementary. We believe that x-links and our concept of problem-solution feature interactions are quite similar in nature. However, by using default logic we have generalized our concept into generic configuration rules that are able to deal with incomplete requirement specifications. The latter is a key issue in the application-specific configuration of dynamic adaptation support systems and middleware in general. In the remainder of this section, we first discuss the relation to other approaches that focus on automating the application configuration phase starting from feature models. Thereafter, we position our work in a more broadly space of related work.

Automated configuration In Beuche's pure::variants approach [11], component implementations are annotated with formulae in propositional logic to indicate for which combinations of features these components are suited best. A Prolog-based constraint solver is used to automatically deduce the optimal set of components

[3] Architecture evaluation tools such as SonarJ [3], Lattix [2] or alike might be useful in this regard.

given a feature model instantiation. Similarly, Czarnecki et al. [14] directly annotate model templates with feature presence conditions based on which the model template can be automatically instantiated given a feature model instantiation. In comparison to our approach, these approaches do not represent configuration knowledge in a separate modeling artefact. Our approach separates problem and solution space and relates both through explicit problem-solution feature interactions. As a result, we focus on reasoning over the key design decisions in the solution space without being restricted to a certain software development paradigm (instead of enforcing any architectural style or modeling template beforehand).

To our knowledge, two other approaches represent configuration knowledge in a separate artefact. First, Van der Storm et al. [53] map problem-space feature models to software artefacts in a separate modeling step. In particular, n-to-1 mappings (expressed in propositional logic) are supported in order to select software artefacts when certain combinations of problem features are selected. Our approach tackles the opposite problem, namely the inclusion of combinations of solution features depending on a selection of specific problem-space features. Secondly, Tun et al. [52] have recently extended the approach of Metzger et al. [32] by integrating it with the Jackson-Zave framework for requirements engineering. Based on this conceptual basis, Tun et al. define a general procedure for deriving an optimal software configuration given a requirements specification. Although x-links in Metzger's approach are in principle generic n-to-n mappings, Tun et al. only give simple examples of one-to-one mappings. A strong point of Tun's approach is that its derivation process can take into account quantitative constraints. The major difference between our approach and the above two approaches is that by expressing the mappings in default logic, our approach can deal with incomplete requirements specifications.

As we use the DLV answer set solver, our approach is also strongly related to software configuration approaches that are inspired by artificial intelligence (AI) research on physical product configuration (i.e. structure-oriented configuration tasks of assembling mechanical products from parts) [19, 46]. Krebs et al. [28] explore the usage of the configuration system Konwerk [19] for mapping customer requirements to configuration of software components in product lines for embedded systems. Similar to our approach, Myllärniemi et al. [34] use the Smodels answer set solver [35] for automatically finding a software component composition that satisfies given functional and non-functional requirements [13]. A major difference with our approach is that we focus on the configuration of (solution-space) feature models whereas these existing AI approaches focus on configuring component models with classification and aggregation relationships. Feature modeling however focuses on capturing choices (e.g., alternative and optional features) rather than different kinds of relationships between components [4]. For this reason and on a more detailed level of comparison, our approach prefers to use the DLV system over Smodels because DLV supports disjunction and classical negation whereas Smodels does not. On the other hand, the Smodels inference engine is more efficient than DLV but this is not a problem in our approach because the computational complexity of our configuration problem is relatively low.

Broader context In this last part of the related work section, we discuss the position of our work in the broader field of software product line engineering and model-driven development. Recently, several researchers are interested in simplifying the mapping between problem and solution space in order to alleviate the feature traceability problem by means of exploring ideas like feature-oriented programming [37], feature-oriented software development [25], aspect-oriented programming [29] and multi-dimensional separation of concerns [49]. Various SPLE approaches support the (meta-)modeling and traceability of variability across the different phases of SPLE [9, 50, 6, 45, 36]. These approaches give guidance on bridging the gap between high-level requirements and detailed design through providing essential documentation, design, and assessment practices. These approaches form thus a sensible starting point for applying the above automated software configuration approaches. Design spaces [9] guide the application developer at stating requirements, preventing incomplete or inconsistent requirements. Finally, an existing model driven development approach for product lines [54] combines model weaving and model transformation in order to support powerful transformations from problem to solution space while taking into account functional and non-functional requirements. Functional and non-functional requirements are modeled as separate feature models that are mapped to problem-space and solution-space models respectively. This mapping is performed either using a negative variability approach (similar to [14]) or by a positive variability approach (which performs aspect weaving [27] on the model level [22]). Orthogonal to this, model-to-model transformations are leveraged for automatically obtaining solution-space models from problem-space models. The approach is powerful and targets the full SPLE process while our approach only targets the application configuration phase. Possibly our approach can be used to deal with interactions between functional and non-functional feature models in their approach, but it is not clear whether there is a case for such interactions in this model-driven SPLE approach.

7. Conclusion

Mapping problem-space features into solution-space features is a fundamental configuration problem in software product-line engineering. In this paper, we have proposed an approach to model complex configuration rules based on the concept of problem-solution feature interactions. These are interactions between solution-space features that only arise in specific problem contexts. The approach allows us to generate the most optimal configuration of solution-space features, given a requirements specification and a set of configuration rules. We have proposed to use default logic to model complex mappings between problem-space and solution-space features. We also have discussed the use of the DLV answer set solver to support our approach by expressing a particular configuration problem as a logic program whose answer set corresponds to the optimal combinations of solution-space features. The advantages of the approach are twofold. First, we can express complex mappings that involve multiple features that cross the problem and solution domain. Secondly, incomplete requirements specifications do not keep us from generating optimal configurations of solution-space features. We have motivated and illustrated our approach with a case study from the field of managing dynamic adaptations in distributed applications.

References

[1] The dlv tutorial. http://www.dbai.tuwien.ac.at/proj/dlv/tutorial/.

[2] Lattix. http://www.lattix.com.

[3] Sonarj. http://www.hello2morrow.com/products/sonarj.

[4] M. Antkiewicz and K. Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In M. G. Burke, editor, *ETX*, pages 67–72, 2004.

[5] G. Antoniou. A tutorial on default logics. *ACM Comput. Surv.*, 31(4):337–359, 1999.

[6] F. Bachmann, M. Goedicke, J. C. S. do Prado Leite, R. L. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A meta-model for representing variability in product family development. In F. van der Linden, editor, *PFE*, volume 3014 of *LNCS*, pages 66–80. Springer, 2003.

[7] D. Batory. Feature models, grammars, and propositional formulas. In *SPLC 2005*, 2005.

[8] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *ACM*, 49(12):45–47, 2006.

[9] L. Baum, M. Becker, L. Geyer, and G. Molter. Mapping requirements to reusable components using design spaces. In *RE 2000*, page 159, Washington, DC, USA, 2000. IEEE Computer Society.

[10] P. Besnard. *Introduction to Default Logic*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1989.

[11] D. Beuche. *Composition and Construction of Embedded Software Families*. PhD thesis, 2003.

[12] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115 – 141, 2003.

[13] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Springer, October 1999.

[14] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE*, volume 3676 of *LNCS*, pages 422–437. Springer, 2005.

[15] K. Czarnecki, U. Eisenecker, and K. Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

[16] S. Deelstra, M. Sinnema, and J. Bosch. Product derivation in software product families: a case study. *Journal of Systems and Software*, 74(2):173–194, 2005.

[17] F. M. Donini, M. lenzerini, D. Nardi, and A. Schaerf. Reasoning in description logics. pages 191–236, 1996.

[18] M. L. Ginsberg, editor. *Readings in nonmonotonic reasoning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[19] A. Günter and C. Kühn. Knowledge-based configuration: Survey and future directions. In *XPS*, pages 47–66, London, UK, 1999. Springer-Verlag.

[20] J. Hillman and I. Warren. An open framework for dynamic reconfiguration. pages 594–603, 2004.

[21] N. Janssens. *Dynamic Software Reconfiguration in Programmable Networks*. PhD thesis, Department of Computer Science, K. U. Leuven, Leuven, Belgium, 2006.

[22] J.-M. Jézéquel. Model driven design and aspect weaving. *Software and System Modeling*, 7(2):209–218, 2008.

[23] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990.

[24] K. Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *Software, IEEE*, 19(4):58–65, Jul/Aug 2002.

[25] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. Featureide: A tool framework for feature-oriented software development. In *ICSE*, pages 611–614. IEEE, 2009.

[26] D. Keck and P. Kuehn. The feature and service interaction problem in telecommunications systems: a survey. *Software Engineering, IEEE Transactions on*, 24(10):779–796, Oct 1998.

[27] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. L., and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, volume 1241 of *LNCS*, 1997.

[28] T. Krebs, L. Hotz, and A. Gnter. Knowledge-based configuration for configuring combined hardware/software systems. In *in Proc. of 16. Workshop, Planen, Scheduling und Konfigurieren, Entwerfen (PuK2002*, pages 10–11, 2002.

[29] K. Lee, K. C. Kang, M. Kim, and S. Park. Combining feature-oriented analysis and aspect-oriented programming for product line asset development. *SPLC*, 0:103–112, 2006.

[30] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, 2006.

[31] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:2002, 2002.

[32] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. *RE 2007*, pages 243–253, Oct. 2007.

[33] K. Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed systems*. PhD thesis, 1999.

[34] V. Myllarniemi, M. Raatikainen, and T. Mannisto. Using a configurator for predictable component composition. pages 47–58, Aug. 2007.

[35] I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal lp. In *LPNMR*, pages 421–430, London, UK, 1997. Springer-Verlag.

[36] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, September 2005.

[37] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443, 1997.

[38] R. P. D. Redondo and others. Supporting software variability by reusing generic incomplete models at the requirements specification stage. In *ICSR*, volume 3107 of *LNCS*, pages 1–10. Springer, 2004.

[39] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.

[40] F. Sanen, E. Truyen, and W. Joosen. Managing concern interactions in middleware. In *Distributed Applications and Interoperable Systems*, volume LNCS 4531, pages 267–283, 2007.

[41] F. Sanen, E. Truyen, and W. Joosen. Modeling context-dependent aspect interference using default logics. In *Fifth workshop on Reflection, AOP and Meta-data for Software Evolution*, July 2008.

[42] F. Sanen, E. Truyen, and W. Joosen. Problem-solution feature interactions as configuration knowledge in distributed runtime adaptations. In *10th International Conference on Feature Interactions in Telecommunications and Software Systems*. IOS Press, 2009.

[43] M. Schlick and A. Hein. Knowledge engineering in software product lines. In *ECAI 2000 - Workshop on Knowledge-Based Systems for Model-Based Engineering*, 2000.

[44] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. pages 139–148, Sept. 2006.

[45] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. Covamof: A framework for modeling variability in software product families. In *SPLC*, volume 3154 of *LNCS*, pages 197–213. Springer, 2004.

[46] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming*, pages 195–201, Stanford, USA, March 2001. AAAI Press.

[47] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, 2005.

[48] C. Szyperski. *Component Software*. Addison-Wesley, 2002.

[49] P. Tarr, H. Ossher, W. Harrison, and Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE*, pages 107–119, New York, NY, USA, 1999. ACM.

[50] S. Thiel and A. Hein. Systematic integration of variability into product line architecture design. In G. J. Chastek, editor, *SPLC*, volume 2379 of *Lecture Notes in Computer Science*, pages 130–153. Springer, 2002.

[51] E. Truyen, N. Janssens, F. Sanen, and W. Joosen. Support for distributed adaptations in aspect-oriented middleware. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development*, pages 120–131. ACM, 2008.

[52] T. T. Tun, Q. Boucher, A. Classen, A. Hubaux, and P. Heymans. Relating requirements and feature configurations: A systematic approach. In *SPLC*, 2009.

[53] T. van der Storm. Generic feature-based software composition. In *Software Composition*, LNCS 4829, pages 66–80. Springer, 2007.

[54] M. Völter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC*, pages 233–242. IEEE Computer Society, 2007.

[55] P. Zave. An experiment in feature engineering. In *Programming methodology*, pages 353–377. Springer-Verlag NY, Inc., 2003.