Support for Collaborative Feature-Based Product Configuration in Software Product Lines

Marcilio Mendonca, Donald Cowan
David R. Cheriton School of Computer Science, University of Waterloo
Waterloo, Ontario, Canada
{marcilio,dcowan}@csg.uwaterloo.ca
http://csg.uwaterloo.ca/~marcilio

Technical Report CS-2007-30, August 2007

Abstract

In Software Product Lines (SPLs), product configuration is a decision-making process in which a group of stakeholders indicate the features desired for a particular product (software). A feature model is normally used to represent the spectrum of available configuration decisions and thus works as a guide to the configuration process. Although in practice product configuration is seen as a collaborative activity that involves satisfying stakeholders with divergent interests and skills, current configuration technology is essentially single-user-based in which user requirements are interpreted and translated into configuration decisions by a single role referred to as the application engineer. As a consequence, product configuration becomes time-consuming and inaccurate especially in the case of large product lines. This technical report discusses a doctoral research proposal on *Collaborative Product Configuration* (CPC). The research aims at investigating the major challenges of realizing CPC in SPLs and, subsequently, at developing an approach that explicitly addresses the problems identified. CPC concepts, algorithms, and tool support are discussed and some preliminary experimental results are shown. The research is expected to bring contributions to the SPLs field by paving the way for a deeper understanding of collaborative product configuration, and ultimately by fostering the development of newer and better approaches in the future.

1 Introduction

Software Product Lines (SPLs) [1][3] is a product-family approach to software development that capitalizes on reusable assets as a means to improve software quality while reducing production costs and shortening time-to-market [2]. Unlike traditional start-from-scratch software development approaches where new artifacts are produced for every new product, SPLs foster a reuse-driven construction process based on two major phases namely *product-line engineering* (or domain engineering) and *product engineering* (or application engineering). At product-line engineering a set of reusable core assets (e.g. UML class diagrams, source code templates, test cases) are developed to address a whole problem domain rather than a single problem, and thus expose some variation points, i.e., points for customization. In the product-engineering phase available core assets are customized according to user requirements in order to produce individual software products. Currently, SPLs is a very active area of research where the combined effort of the academia and the industry has proven highly beneficial to its development.

Product configuration is a key product engineering activity in SPLs in which the features desired for a product are chosen. Features are commonly arranged in hierarchical structures known as *feature models* [6] that provide a convenient place for storing product configuration decisions. After a set of configuration decisions are made a product specification is produced that will commonly serve as input for automated product generation tools.

In practice, the spectrum of product configuration decisions represented in the feature model commonly spans over several technical and non-technical knowledge domains thus demanding decision makers with different backgrounds (e.g. customer, product manager, software engineer, database administrator) to actively participate in the configuration process. Furthermore, collaborative product configuration scenarios may also have to enforce a specific authority scheme in which, for example, the decisions of a particular configuration role (e.g. product manager or customer) should prevail over other roles' decisions (e.g. technical decisions of a database manager or a software engineer). In fact, as well pointed by Krueger [4], "*the decision-making role for product creation may be an engineering role such as an application engineer ... or it may be a non-engineering role such as a product marketer, a sales person, or the customer. The role should be given to the person who can make the best decisions at the best time*". As a consequence, coordination of activities becomes a major issue in product configuration in order to minimize decision conflicts and enforce the correctness of product specifications. For instance, decision conflicts arise when different *configuration actors*, i.e., the people directly involved in the configuration decision-making, make decisions that can not hold together typically because they violate one or more configuration constraints.

Furthermore, large product lines may exhibit thousands of features (sometimes refereed to as *feature explosion* [7]) thereby requiring about the same number of configuration decisions to be dealt with. Teamwork in such scenarios is highly desirable to cope with the general complexity of the configuration process. In addition, other factors such as *proper authority* and *specialized knowledge* may require people with complementary backgrounds to participate in the configuration process. In other scenarios products may be configured in multiple stages [8] in which in each stage a partial configuration of the product is produced as in the case of software supply chains [8]. The rationale for splitting product configuration in multiple stages can be related to time, roles and targets [8].

As a consequence, understanding how to proper support collaborative work in the context of product configuration turns out to be a critical issue in SPLs especially with regards to its adoption by software organizations that already have a high demand for efficient and coordinated teamwork. Similarly important is to understand what makes current SPLs technology inappropriate for collaborative product configuration and how to provide useful extensions.

1.1 Problem Statement

Although in practice product configuration may be seen as a collaborative process where people with different expertise and authority levels actively contribute in building a single and consistent product specification, current product configuration approaches in SPLs do not explicitly support collaborative configuration. In fact, current configuration technology is essentially single-user-based where user requirements are interpreted and translated into configuration decisions by a distinct configuration role generally known as *application engineer*. This process is error-prone and time-consuming especially when large feature models with hundreds or thousands of features are considered. In addition, stakeholders' participation in the process is essentially *passive*, i.e., limited to providing requirements to application engineers and hoping that useful features are included in the product. As a clear evidence of the problem several product configuration approaches [6][24][25][26][27] and tools [14][15][16][34][28][30] in SPLs heavily rely on feature models to support product configuration yet there is still a considerable knowledge gap on how to use feature models in a multi-user-configuration context. Indeed, while most of the approaches incorporate abstractions related to variability such as features very rarely teamwork concepts are taken into consideration. As a consequence, effective tool support for collaborative configuration is missing.

Some SPL research works have attempted to tackle configuration decision-making in different ways but in general the approaches adopted are either just high-level descriptions of collaborative configuration scenarios with no tool support or based on too restrictive assumptions such as assuming that configuration tasks can be split into fully independent sets, which is generally not the case in practice.

In reality, the challenges to enable collaborative configuration are many including finding effective means to split and coordinate configuration tasks. For instance, configuration work needs to be split among multiple configuration actors who, guided by a plan, coordinate their actions to produce a single consistent product specification. The coordination problem becomes a real issue if we consider feature models with a complex network of feature constraints connecting different branches of the feature tree. Potentially, the higher the number of constraints the higher the degree of work coupling which in turn requires strategies for conflict resolution. In fact, coordinating configuration work is a critical issue to the success of collaborative product configuration approaches.

Finally, some technical issues make the development of software for collaborative configuration particularly challenging. Because collaborative configuration scenarios may involve connecting people distributed across different space and time dimensions, configuration software should ideally be thought of as a distributed system. Hence, aspects such as communication and group awareness become relevant in order to minimize decisions conflicts and facilitate work coordination.

The research problems we identified can be summarized by the following questions:

- How can current technology for product configuration be adapted or extended to explicitly address collaborative configuration demands?
- How can collaborative configuration scenarios be represented and validated? Is it possible to perform static analyses to validate such scenarios? How can scenario representations be made executable?
- What kind of tool support can be provided to support collaborative configuration? How scalable are such tools and major algorithms involved?
- What are relevant properties of collaborative product configuration? How can they be enforced?
- Can groupware concepts be borrowed to improve tool support for collaborative product configuration? Which concepts are worth borrowing?

1.2 Limitation of the State-of-the-Art

Some attempts to improve the process of configuring products have been made but generally none of them tackled collaborative configuration as a first-class problem.

Reducing the overall complexity of product configuration

Several works on product configuration have focused on providing mechanisms to cope with the inherent complexity of large configuration spaces. For instance, in large product lines that encompass thousands of features the process of specifying product variants can be non-trivial. Proposed alternatives to alleviate this problem have mostly concentrated on minimizing the mandatory number of decisions required to configure products. Some examples include the use of default values or assumptions regarding the selection state of features as well as the provision of propagation mechanisms. Furthermore, current product configuration tools provide some sort of mechanism to support the reuse of configuration decisions based on a existing configurations [14][15] or the specialization of partial specifications [16]. In practice, a base configuration can be represented by a partially configured feature model that serves as a starting point to configure new products. Reusing configurations can be particularly useful when the product line tends to derive very similar products. Nevertheless, producing a base configuration or adapting an old configuration can be a cumbersome process that may even requires teamwork in order to identify what should or not be reused. The trade-offs of using a base or old configuration to support product configuration were discussed in [18].

On the other hand, support for collaborative configuration in such approaches is virtually non-existent. Most of the configuration work is performed by reusing and adapting previous configuration decisions and there is a lack of understanding about how this process can be conducted by teams.

Product configuration as a Constraint Satisfaction Problem

Product configuration has also been addressed as a Constraint Satisfaction Problem (CSP) [19][20]. Product line configuration knowledge is described in terms of a component-port representation [21] that includes a set of constraints to restrict the components combinability. Constraints are normally expressed in a formal notation (e.g., logic predicates). Similarly, user requirements are translated to a formal representation allowing the problem to be solved by automated systems (also known as configurators). Configurators will attempt to find a set of values that satisfy both, the user requirements and the configuration constraints. Because multiple solutions can be found, configurators may have to incorporate *optimal solution* strategies to find a single final configuration. For instance, a configurator may use a cost-based strategy to distinguish among a group of components so that the lowest-cost one is chosen. Enhanced versions of the CSP approach were developed to support the notion of distributed configuration

[21]. Typically, the configuration problem was translated into a distributed constraint satisfiability problem (DisCSP) [22] in which the problem constraints and variables are fragmented over multiple configuration environments. Each environment is controlled by an intelligent software agent that works as a local configuration system. DisCSP approaches work on distributed algorithms to support software agents' communication (e.g., message passing mechanisms) and coordination (e.g., enforcement of local and global constraints).

The CSP and DisCSP approaches focus on developing algorithms and machinery support for solving constraint satisfaction problems. The assumption is that machines can quickly process thousands of instructions and perform efficient backtracking until a desirable solution is found. The involvement of humans in the process is limited to providing requirements to the configuration system in terms of logic formulas. In our approach, even though we plan to take advantage of CSP machinery and algorithms the goal is to assist human decision-making. Consequently, it is not natural to think of processing power in this context but rather means to boost human coordination. For instance, communication and awareness are two possible strategies humans can take advantage of to minimize decision conflicts. In this context, CSP-related algorithms and graphical user interfaces can be combined to assist humans in visualizing and resolving decision conflicts.

Staged configuration of products

The work on staged configuration [8][23] pointed out various scenarios in which product configuration can be performed in stages. The authors introduced two configuration techniques called specialization and multi-level configuration to support the idea of staged configuration. A case-study in the automotive industry was provided to exemplify how an embedded operating system for a vehicle could be configured in multiple stages. A high-level workflow illustration was given showing that in each stage it was possible to have multiple configuration actors configuring a subset of the feature model concurrently and have their specifications combined afterwards.

In our approach we look deeply at aspects related to collaborative configuration. Among other issues, we want to understand how configuration tasks can be properly split and assigned to roles, how to specify, validate and execute configuration scenarios, and how to provide effective support for decision conflict resolution. Additionally, we want to borrow groupware concepts that might prove valuable in the context of collaborative configuration such as coordination, communication, and awareness. Ultimately, the idea is to tackle collaborative product configuration in a broader and more explicit manner and provide appropriate tool support.

1.3 The Proposed Research at a Glance

To overcome the problems mentioned we propose an approach that builds upon well-established product configuration models to *explicitly* support *feature-based product configuration* or as we will refer from now on *Collaborative Product Configuration (CPC)*. Among the driving forces are the needs to better understand how configuration tasks can be split, represented, validated, and executed so that multiple configuration actors can actively participate in the product configuration process.

More specifically, the research aims at:

- Developing a model for collaborative configuration that extends feature-based product configuration abstractions to explicitly include teamwork concepts
- Proposing a representation to describe collaborative configuration scenarios including the semantic of its elements (e.g. merge operation)
- Developing algorithms to generate validation constraints to check the correctness of specified configuration scenarios
- Providing means to check collaborative configuration properties such as termination, backtrack-freeness, and deadlock-freeness.
- Developing tool support based on groupware concepts for the specification and execution of collaborative configuration scenarios
- Running case-studies, simulations, or both to check the feasibility and scalability of our approach

1.4 Expected Contributions

The expected contributions of our research could include:

- A model that explicitly represents collaborative product configuration abstractions
- A representation for describing collaborative configuration scenarios
- Algorithms to generate constraints to validate collaborative configuration scenarios
- Formal verification of collaborative configuration properties
- The development of supporting tools to build and execute collaborative configuration scenarios
- Case studies and simulations that shows the feasibility and scalability limits of the approach

We hope that by addressing the research problems discussed in Section 1 we will pave the way for a deeper understanding of collaborative product configuration and ultimately foster the development of newer and better approaches in the future.

1.5 Research Applicability

An important goal in our research is the widespread integration of the approach being developed with a variety of available methods, techniques, and tools for product configuration. For instance, the approach relies primarily on basic feature modeling concepts which allows for an improved compatibility with current feature-based configuration approaches [6][24][25] [26][27] and tools [14][15][16][28][30][34]. Consequently, configuration engineers may use their favorite feature modeling tool to create a feature models and afterwards use a CPC tool to enable the collaborative configuration of that models. For instance, the FeaturePlugin [16] configuration tool can be used as to author a feature model specification which, thereafter, can be imported and interpreted by a CPC tool. Furthermore, we expect our approach to work well with product configuration techniques such as specialization and multi-level configuration used in the context of staged-configuration [23]. In staged-configuration cases arise in which different parties are required to configure a single feature model concurrently and afterwards merge their decisions into a single consistent specification. The CPC approach can be helpful in this context by offering a safe environment, i.e., with well-known rules, validation algorithms, tool support, etc., to describe and perform collaborative product configuration.

The remainder of this document is organized as follows. Section 1 of this document presents the motivation of the research, the research problems identified, some related works that attempted to alleviate these problems, and an overview of the proposed research and the major expected contributions. Background and related work are presented in section 2. The research proposed is presented in more detail in section 3. The context of the research is highlighted, an overview of the research goal is presented followed by a discussion of the approach's components and some preliminary solutions envisioned. Section 3 discusses the limitations of the approach and possible validation alternatives. Section 4 discusses the current state of the research in terms of publications and tool support implementation. Future directions are presented in section 5 and references are provided in section 6.

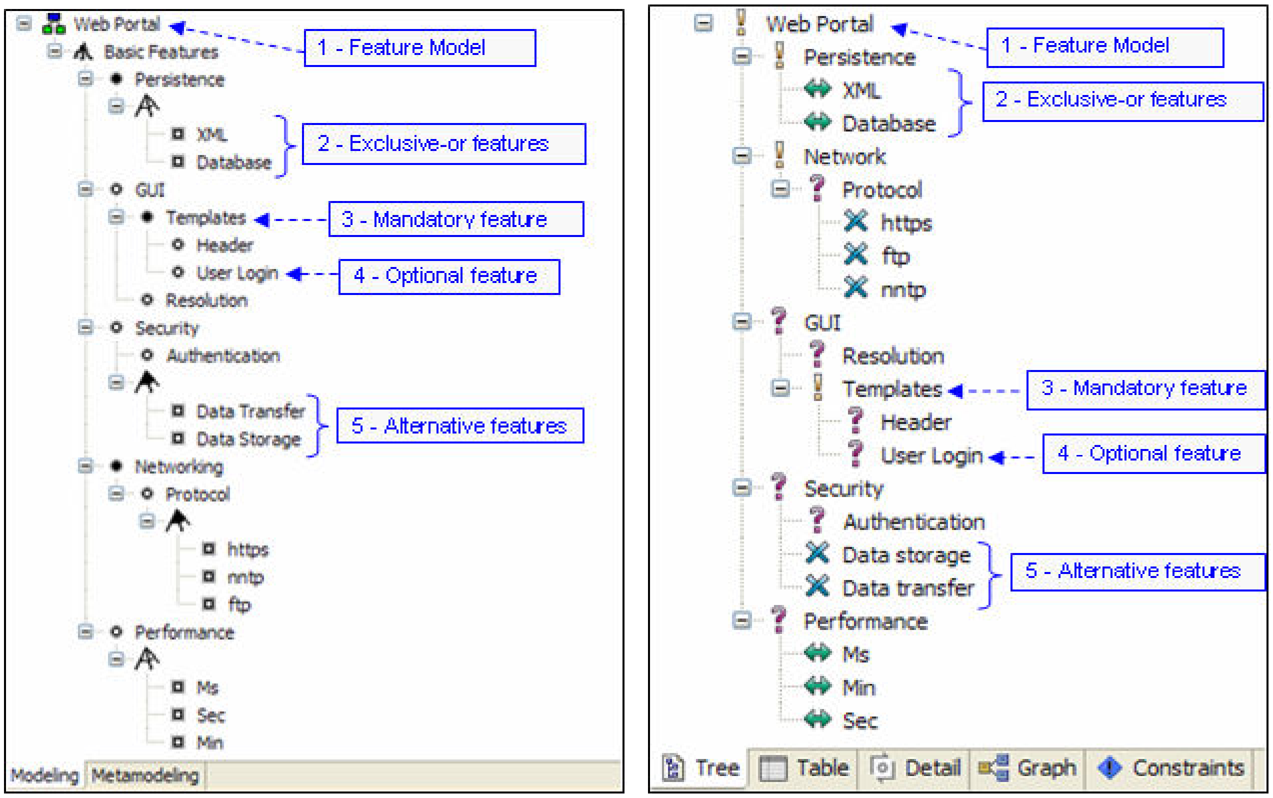
2 Background and Related Work

In this section we provide a background discussion on research topics related to our work.

2.1 Feature-based Product Configuration

Feature-based product configuration approaches have feature models as its central constituent. Originally proposed by the Feature-Oriented Domain Analysis (FODA) [6] method, feature models offer a powerful yet simple representation to represent commonalities and variabilities of a software product family. Since its inception, feature models have been widely supported by several SPL approaches [6][24][25][26][27] and tools [14][15][16][34][28][30]. Various enhancements have been proposed in attempts to improve its descriptive power and make it more appropriate for automated product generation. Examples of SPL methods that support the notion of feature models include FORM [24], FOPLE [25], FeatuRSEB [34], Alexandria [26], and generative programming [27]. In addition, the feasibility of using feature models to boost product derivation made them specially attractive to commercial and academic product configuration tools such as FeaturePlugin [16], CaptainFeature [30], pure::variants [15], Gears [14], xFeature [28]. While some tools are specifically designed to support product configuration (e.g., FeaturePlugin in Figure 1-A) others will also support fully automated product generation (e.g., pure::variants in Figure 1-B).

The major concept behind a feature model is that of a *feature.* According to Kang [6] a feature is "*a prominent or distinctive and user-visible aspect, quality, or characteristic of a software system or systems*". A slightly different definition by Czarnecki [27] states that a feature is "*a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in a family*". In the context of our research on collaborative configuration the feature definition from Czarnecki sounds more appropriate as he emphasizes the concept of stakeholders that can directly influence product configuration. For instance, stakeholders such as the customer, a sales manager, a database manager, a system administrator, or even a software developer may be directly involved in the product configuration decision-making thereby indicating the features desired for a particular software product. In our research, we support the notion of a feature model as repository of configuration decisions that need to be dealt with at some point during the product configuration process. The decisions should be made by skilled people based on their knowledge and authority in the process.

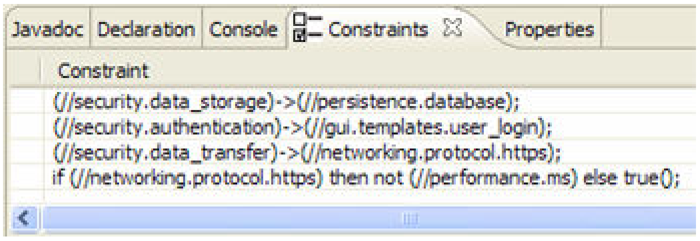


(A) **(B)**

Figure 1: A Web-Portal product line feature model drawn on FeaturePlugin (A) and pure::variants (B)

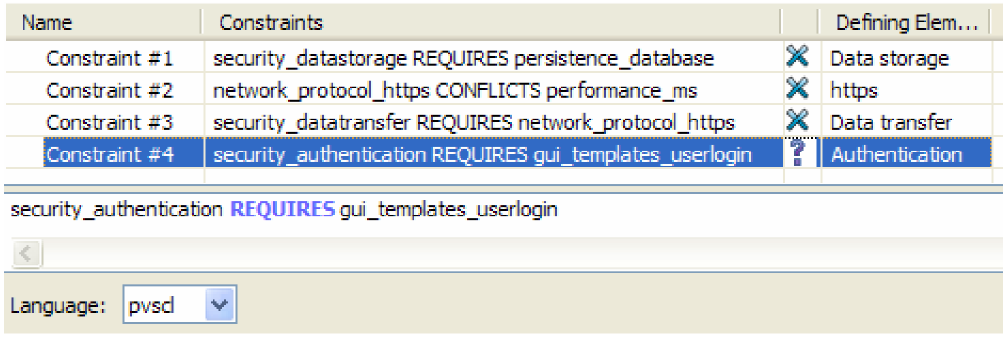
Figure 1 shows a feature model of a Web-Portal product line drawn with two distinct feature-based product configuration tools [16][15]. Mandatory and optional features as well as exclusive-or and alternative features are described (boxes 2, 3, 4, and 5 in Figure 1). These types of features form the core of feature models and hence are commonly supported by product configuration approaches. In the product line engineering phase a feature model is created to represent the available configuration options for a product family. At product engineering, a group of configuration actors will carry out decisions that will customize the product line core assets according to specific user requirements thus yielding to a product specification. Consequently, it is important to ensure that appropriately skilled people are involved in the configuration decision-making to avoid producing products that do not provide the appropriate value to their customers.

The use of additional constraints to restrict feature combinations increases the complexity of product configuration especially in a collaborative context. Yet constraints are at the core of the configuration problem and are indeed often used in practice. Without constraints the output of the configuration process could be a specification that does not represent a *valid product*, i.e., where all the components work well together and do not conflict with each other. An example of an *invalid product* would be an automobile with a powerful engine, let us say a *V6-engine*, and a simple *mini-chassis* that does not support large engines. In this case, we say that the *mini-chassis* feature conflicts with the *V6-engine* feature.



```
(//security.data_storage)->(//persistence.database);
(//security.authentication)->(//gui.templates.user_login);
(//security.data_transfer)->(//networking.protocol.https);
if (//networking.protocol.https) then not (//performance.ms) else true();
```

(A)



Name	Constraints		Defining Elem...
Constraint #1	security_datastorage REQUIRES persistence_database	✗	Data storage
Constraint #2	network_protocol_https CONFLICTS performance_ms	✗	https
Constraint #3	security_datatransfer REQUIRES network_protocol_https	✗	Data transfer
Constraint #4	security_authentication REQUIRES gui_templates_userlogin	?	Authentication

```
security_authentication REQUIRES gui_templates_userlogin
```

Language: pvscl

(B)

Figure 2: Web-Portal feature model constraints - FeaturePlugin (A) and Pure::variants (B) configuration tools

The use of configuration constraints is also a common practice and thus constraints are widely supported by product configuration tools. In Figure 2, we show constraints for the Web-Portal product line. Figure 2-A illustrates constraints using the FeaturePlugin XPath constraint language. Figure 2-B shows the same constraints written in pure::variants *pvscl* constraint language. Constraint #4 enforces that whenever feature *security_authentication* is selected so must be feature *gui_templates_userlogin*. In other words, if authentication is a requirement then a user login interface must be available to gather users' login and password information.

2.2 Feature Explosion

A common issue with the use of feature models to describe product lines variabilities is the *feature explosion* [7] problem. For instance, suppose the feature model sketched in Figure 1 contains thousands of features and a fairly large number of constraints. In this case, the configuration problem would gain a new dimension in terms of complexity. In a large feature model it may be hard to understand the exact impact of making a decision because of the intricate network of decision dependencies and automatic decision propagation mechanisms, to identify who is in charge of what features and how to coordinate the decision-making, and to be aware of other players' decisions and the corresponding impact in ones' decision space.

As we discussed previously, attempts to alleviate this problem have mostly focused on minimizing the need for decisions by fostering the reuse of previous decisions. As argued, the reuse of decisions is also a decision-making process itself that may require decisions makers to carefully analyze which decisions are worth reusing. Furthermore, the process of generating a database of reusable configuration requires collaborative work where domain experts will indicate the most appropriate configuration choices for different situations and configuration goals.

2.3 Feature Mappings

Research on *feature mappings* [31][29] has exploited means to link features to architectural software components as a means to support the automation of product generation. The challenge of feature mappings is due to the scattering and tangling nature of feature realization at architectural level. For instance, the decision to select feature *database* in the Web-Portal product line in Figure 1 may affect several architectural artifacts such as source code files, libraries, scripts, configuration files, etc. While feature mapping approaches allows for a rapid production process that fulfills time-to-market demands, maintainability can be a challenge since changes on feature models or architectural components may also require updating the mappings.

2.4 Feature Interaction

Feature interaction [31][32] is another interesting area of research involving feature models. The problem initially rose in the telecommunication domain and has also been object of research in SPLs. According to Zave, *"a feature interaction is some way in which a feature or features modify or influence another feature in defining overall system behavior*" [32]. Hence, feature interaction concerns the way features are composed to form a software service or product. While features enable the flexibility of offering personalized products to clients they also require a deep understanding of how their composition may affect each others' expected functionality at the cost of having "*software bugs, cost and schedule overruns, and unfortunate user experiences*" [33]. A very clear example of a real feature interaction problem in the telecommunication domain can be found in [33]. In our Web-Portal illustration, a feature interaction analysis would be required for example to include the user *authentication* as a feature. A user authentication policy in the Web-Portal would require other features, especially functional ones, to check whether the portal users have the right privileges to access specific resources. Feature interaction is considered a semantic problem where the addition of new features can interfere with existing ones. However, in our research we primarily focus on static dependencies among features expressed in terms of additional constraints. As we progress in our research we may eventually reconsider our position concerning feature interaction problems.

2.5 Feature Model Extensions

Several extensions have been proposed to enhance feature models descriptive power including cloning [38], references [38], attributes [38], relationships [34][35][36], feature categories and annotations [34][37], feature cardinality [38], modularization [39], as well as a variety of rendering styles and graphical notations and those discussed in [40] (e.g., see Figure 1 for two examples of tree-like graphical notations). A comprehensive discussion on feature model extensions can be found in [23] and [40]. In our approach to collaborative configuration we plan to initially support basic feature model abstractions and progressively incorporate new concepts whenever it makes sense.

2.6 Feature Model Formalization

The formalization of feature models has also been focus of research in SPLs. The work from Riebisch [43] discussed similarities among feature representation arrangements, proposed a categorization scheme for features (functional, interface and parameter features), and analyzed different feature relations and constraint conflicts. Batory's work [41] discussed a logic-based representation to express feature relations. By transforming feature models in propositional formulas the work enabled the use of existing logic-based tools such as logic-truth maintenance systems (LTMS) and CSP/SAT solvers. Major benefits include the ability to reason on and debug feature models, support for automatic decision propagation, and rationale support for configuration decisions. Benavides [42] also elaborated on the benefits of connecting feature models to formal logic. More specifically, he discussed the advantages of viewing feature-based product configuration as a constraint satisfiability problem including the ability to track the number of valid configurations available, to filter products features based on particular attributes, to verify the satisfiability of a feature model, and to find the *best* product configuration according to a given criterion.

2.7 Staged Configuration

Some preliminary research work has also supported the notion of teamwork on product configuration even though having analyzed the problem generally at a high-level. For instance, Czarnecki's work on staged configuration [23] provided various examples where product configuration is performed collaboratively in which configuration actors specialize and configure feature models in stages. The examples provided also showed that in certain situations *merge* operations are needed to compose resulting partial specifications. However, the work leaves open issues such as how configuration tasks can be arranged and validated, how to merge inconsistent specifications, what policies for conflict resolution can be used, etc.

2.8 Feature-Based Production Planning

Kang proposed an approach [45] that integrates feature models and the production plan document proposed by the SEI [5]. He suggested that feature models should be split in smaller independent units (*binding units*) so that product developers would be able to configure them separately. However, when feature models expose a complex network of feature dependencies identifying independent binding units may be very complex especially when tool support is not provided. In addition, partition strategies may require dealing with a large number of configuration decisions in diverse knowledge domains thereby requiring a clear strategy to cope with work dependency. In our research, we also suggest the splitting of feature models in smaller more manageable units but rather based on organizational factors such as *specialized knowledge*, *proper authority,* or other attributes that help identifying contributors to the configuration process, i.e., qualified configuration actors. Similarly, our approach provides means to explicitly specify, analyze, validate and represent work dependencies in collaborative product configuration allowing specified collaboration scenarios to be executed by an automated tool. In addition, we plan to borrow groupware ideas as a means to improve tool support for collaborative configuration.

2.9 Groupware and CSCW

The area of Computer-Supported Collaborative Work (CSCW) was initiated in the mid-80's with the purpose of studying how technology could help improve group work [48]. In the years that followed the community strived to define a coherent research agenda to the field [46][47][49]. While some research groups focused on the *CS* dimension, i.e., the development of computer systems to support collaborative work, other groups enforced the *CW* perspective, i.e., understanding the different types of human collaborative engagements so as to adequate the underlying technology accordingly. In the context of our research we are particularly interested in the technical dimension of CSCW, i.e., models, techniques and applications, and how we could capitalize on developed knowledge as a means to provide adequate support for feature-based collaborative product configuration in SPLs.

Collaborative and individual work pertains to the same work domain but represent "*different ways of doing the same*" [59]. In other words, collaborative work changes the means not the ends. For instance, in our approach to collaborative configuration the input is still a feature model and the outcome is a consistent product specification just as in the case of individual product configuration. Another interesting observation is that cooperative ensembles are normally transient and dissolve after reaching a goal [46]. That is mostly the case in collaborative product configuration when a group of configuration actors get together in a temporary work based on their skills, knowledge, authority or other attribute relevant to the problem. After the product is configured and derived the group normally disbands.

2.10 Work Coordination, Communication and Awareness

Another key aspect of collaborative work is work coordination. The coordination problem may be defined as the "*integration and harmonious adjustment of individual work efforts toward the accomplishment of a larger goal*" [62]. In formal collaboration scenarios, i.e., where it can be accurately described who is doing what and when, tools such as workflows can be very useful as a means to explicitly support work coordination. Conversely, in other informal collaboration scenarios participants are usually in charge of coordinating interdependent activities themselves by means of *communication*. In practice, it is observed that regulated and flexible scenario may occur intertwined.

Communication among participants of a collaborative engagement can be synchronous or asynchronous. In real-time interactions synchronicity is normally a requirement as people expect instant feedback. For instance, teleconferencing systems generally provide real-time audio and video communication mechanisms so as to allow participants to see and talk with each other synchronously. Drawbacks of synchronous communication include high infrastructural costs and scalability. In other cases, asynchronous communication may be more appropriate especially when participants are distributed across different time zones. That is, communication requirements are highly dependent on space and time constraints. A typical asynchronous communication mechanism is an e-mail system. E-mails are normally used when there is no sense of urgency otherwise a messenger system, for example, would be more appropriate. Another factor that has to be considered when choosing a communication mechanism is that the higher the work coupling the higher the communication requirements [60]. That is, tightly work coupling requires people to constantly communicate to coordinate their work as opposed to loose work coupling where communication can be sporadic. Depending on the case it may be convenient to provide more elaborate communication mechanisms, i.e., that adds value to basic features such as audio, video, file attachment, and so forth. For instance, in a collaborative design tool remote modelers resolving a conflict involving a UML class (say one wants to delete the class while the other wants to extend it) should be assisted by a tool with relevant data regarding the UML class, dependencies, the rationale for adding/removing the class, and so forth. Just allowing modelers to communicate may not be helpful enough.

Work coordination in groupware systems is also facilitated by the presence of awareness systems. Awareness can be defined as the understanding of who is working with you, what they are doing, and how your own actions interact with theirs [63]. Several CSCW works have shown the importance of group awareness in supporting work coordination. A study on the impact of awareness on open-source software development [61] showed that even when work partition is informal, i.e., when developers can contribute to any part of the code, awareness helped programmers to coordinate their activities effectively and minimize conflicts. Additionally, group awareness can be especially convenient as a means to reduce the need for communication as reported in the open-source study. The importance of group awareness in collaborative work is so evident that frameworks for evaluating distributed collaboration based on awareness have been proposed [60]. In general, they discussed the connection between awareness and other concepts such as work coupling, communication, and coordination.

2.11 The 3C Model: Communication, Coordination and Cooperation

CSCW has also proposed models to support groupware development such as the 3C model (see Figure 3) [51]. In this model, *collaboration* is defined as a combination of *cooperation*, *coordination* and *communication*. The model has been instantiated in a variety of domains (e.g., learningware [50]) and the experience gathered led to the proposal of a systematic process approach to groupware construction based on the Rational Unified Process (RUP) called 3C-RUP-Groupware [56]. A comprehensive discussion on the use of the 3C model to support groupware development can be found in [57].

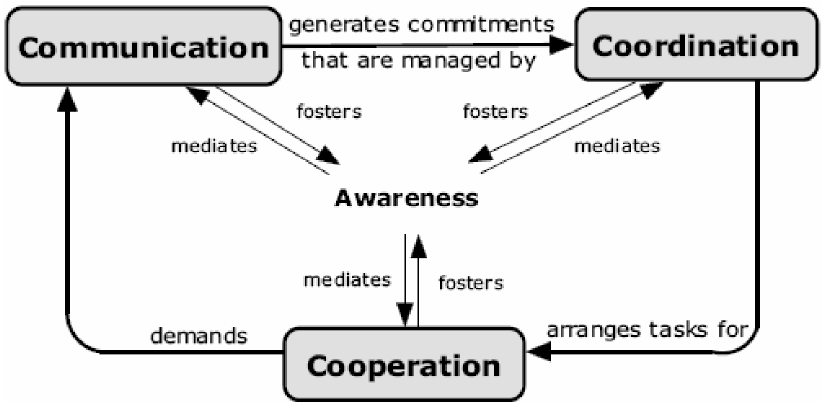


Figure 3: The 3C model instantiated for group work (figure from [50])

Figure 3 shows the 3C model instantiated for group work. The three main concepts, the 3Cs, are related by associations such as "*generates commitments that are managed by*", "*arranges tasks for*", and "*demands*". The 3C model along with other important issues such as space/time considerations in groupware development will serve as basis in our research to develop a model for collaborative work applicable in the context of feature-based product configuration. We understand that it is crucial to take CSCW work seriously into account in our approach as this community has made significant progress exploring models for collaborative work that also greatly support groupware development.

2.12 Collaborative Software Engineering

Software engineering is viewed as an inherent "*collaborative social practice*" [9]. In the context of Software Engineering, CSCW has been materialized as Collaborative Software Engineering (CSE) [55]. CSE aims at studying collaborative work within software development processes such as modeling, coding, and testing. Up to now, some important tools to support collaborative software engineering processes have been developed as in the case of software configuration management [11][10] and collaborative software design [12]. For an excellent annotated bibliography of collaborative software engineering works please refer to [13].

3 Proposed Research

This section overviews the proposed research, gives a good notion of how we intend to approach the research problems, and glances over the major components of the approach.

3.1 Context

The context of the research is depicted in Figure 4. The major research area is Software Product Lines, more specifically Collaborative Product Configuration as indicated by the dashed rectangle labeled "Software Product Lines – Collaborative Product Configuration". The general goal is to investigate how current configuration technology (e.g., models, algorithms, processes) can be extended to support the notion of collaborative product configuration. We intend to take advantage of some concepts and techniques from at least three other fields that seem applicable in the context of our research as indicated by the filled ellipses labeled "Logics", "Groupware/CSCW", and "Workflow". Recent research works [41][42] has built a connection between logics and feature models by translating feature trees into propositional formulas. This allowed off-the-shelf software components such as SAT solvers and CSP tools to be used in the context of product configuration (e.g. debugging feature models, checking for satisfiability).

The area of Groupware/CSCW studies how technology can help improve collaborative work and has raised important issues in the development of collaboration systems. Concepts such as awareness that encourage teamwork players to be aware of each others' activities can help in minimizing product configuration conflicts. For instance, someone making configuration decisions may avoid decision conflicts by being aware of other's configuration decisions. Workflows are very useful to represent organizational processes. Collaborative product configuration as a teamwork process will eventually require some sort of workflow to describe the sequence of steps of configuration processes. Process notations such as BPMN [58] may prove suitable to describe such collaborative configuration scenarios [66].

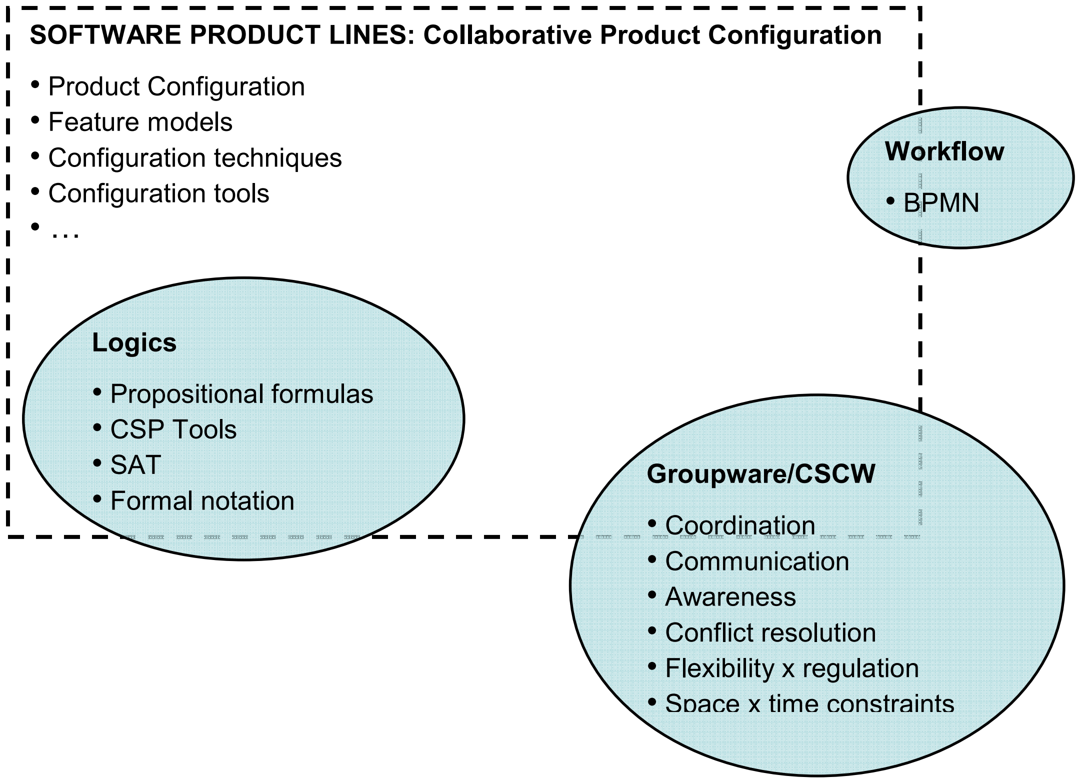


Figure 4: Research Context: Collaborative Product Configuration and Related Fields

3.2 Overview

Figure 5 illustrates how we intend to approach the research problems introduced in Section 1. Figure 5-A shows a product configuration scenario as it is commonly viewed, i.e., as a single-user activity. In this scenario, a feature model produced during product line engineering serve as basis for product configuration. A single configuration actor using a feature-based product configuration tool will select the desired features for a product based on particular user requirements. Stakeholders' involvement in the process is limited to providing requirements to the single configuration actor. Configuration conflicts caused by incompatible user requirements are normally not anticipated and handled in an *ad-hoc* manner. On behalf of the stakeholders configuration actors derive a product specification that will serve as input for automated product generation.

Figure 5-B illustrates how we intend to enhance *scenario A* to support multi-user configuration. Similarly to *scenario A*, a feature model is provided as input for the configuration process. However, before the actual product configuration takes place a *CPC scenario specification* activity is required to annotate feature models with teamwork abstractions. This activity involves identifying desired configuration roles, actual players for each role, and specifying the set of configuration decisions for each role. The goal of this activity is to produce a *CPC scenario description* that describes how the configuration actors are expected to participate in the process, the order of their tasks and the resolution of possible decision conflicts.

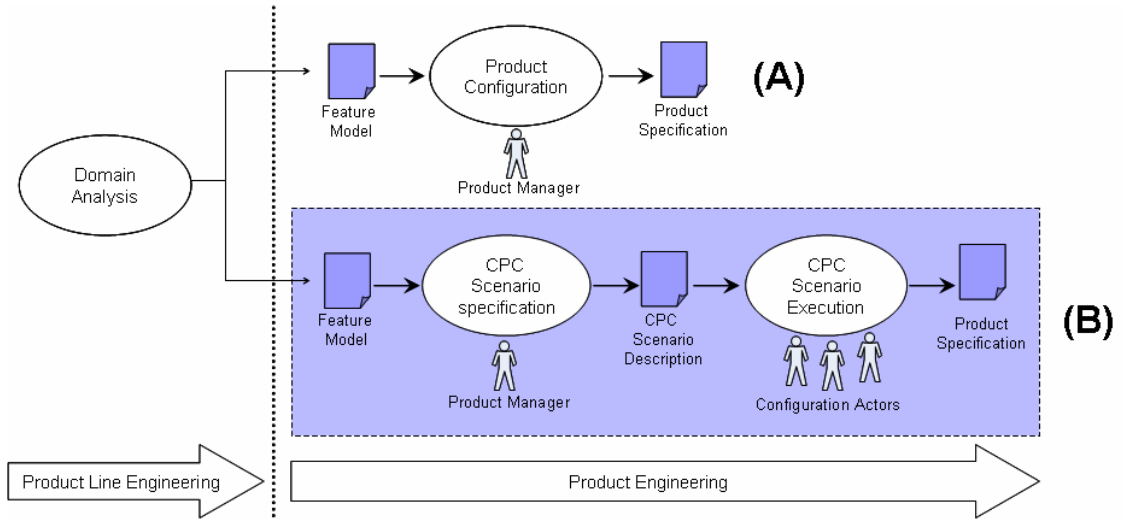


Figure 5: Traditional (A) and collaborative (B) feature-based product configuration scenarios

Yet on *scenario B* the *CPC scenario execution* activity supports collaborative product configuration based on a *CPC scenario description* provided as input. This activity is ideally supported by a collaborative tool that is capable of interpreting and executing CPC scenario descriptions. The tool enforces coordination constraints described in the process model, offers communication and awareness mechanisms, facilitates conflict resolution, and produces a valid product specification as the outcome.

In the following we discuss our approach to collaborative product configuration that aims at supporting the scenario illustrated in Figure 5-B. We also point out some research issues and questions that might arise in developing the approach.

3.3 Approach

In the following we provide a detailed discussion of the relevant issues and questions concerning our research work.

3.3.1 Specifying Collaborative Product Configuration Scenarios

The first step in our approach to CPC is to specify a collaboration scenario description that will guide the product configuration process and allow teams of configuration actors to make decisions in a proper coordinated manner. The person in charge of specifying the collaboration scenario description is ideally the one in charge of the product line and that also interfaces with the customer. What qualifies this person is her privileged view on the people involved in the product customization, what their skills and roles are, and how they could contribute to the process. We use to think of this person as playing the product manager or project manager role. From now on, we will refer to this role simply as the *product manager*. Among the major activities of the product manager role are the identification of the configuration roles and potential candidates to play them, the splitting of the configuration responsibilities, the analysis and refinement of the work dependencies, and the design of collaborative configuration scenarios. In the following, we discuss how we plan to tackle these activities and the major research issues involved.

3.3.1.1 Incorporation of collaborative product configuration abstractions

A common technique to integrate new and existing abstractions into a single model is to manipulate the abstract syntax representation of the model (e.g. meta-model). The first step consists in adding the new abstractions to the abstract syntax and subsequently to draw associations between previous and new abstractions. Several feature model meta-models have been proposed in the literature [27] [23] and described in terms of UML class diagrams which allows new abstractions to be easily incorporated. While some meta-models support basic feature model abstractions [27] such as mandatory, alternative, inclusive-or, and exclusive-or features (see Figure 6) others [23] also include concepts such as references, feature attributes, and feature cardinality. The incorporation of new abstractions into feature models meta-models is very likely to continue as an attempt to improve quality attributes such as expressiveness, succinctness, and naturalness [44].

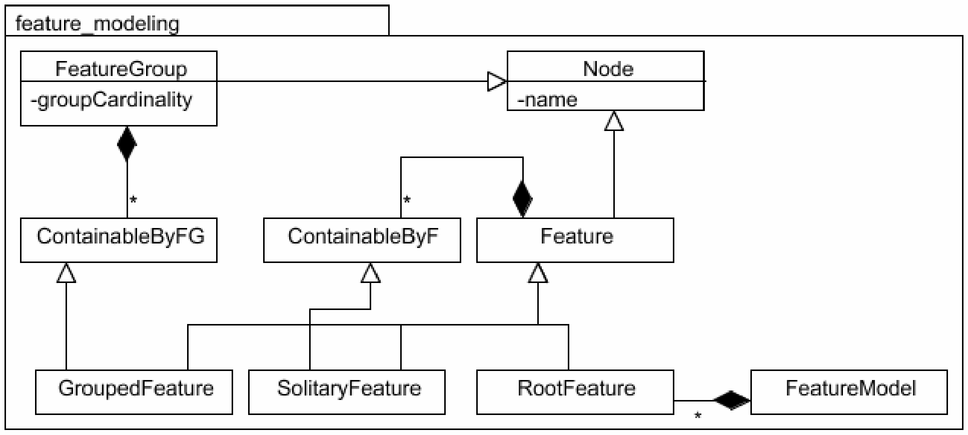


Figure 6: Feature model meta-model described in [23] simplified to describe only basic feature model concepts

In our approach we plan to integrate collaborative configuration abstractions with basic feature model abstractions. As our approach develops we may attempt to gradually support other advanced feature model abstractions as long as they make sense in the context of collaborative configuration. Furthermore, as we just mentioned it is very unlikely that there will be a definitive feature model meta-model that incorporates all the extensions proposed in the literature which kind of reflects the different research viewpoints on feature models.

Figure 6 shows a feature model meta-model proposed by Czarnecki [23] simplified to describe only basic feature model concepts. The meta-model specifies that a feature model has a single root feature and that features can contain other features building a hierarchical structure. Features can be solitary (*SolitaryFeature*) or grouped (*GroupedFeature*)in feature groups (*FeatureGroup*). Some parent/children constraints are enforced by the elements *ContainableByFG* and *ContainableByF*.

In Figure 7, we sketch out a *collaborative configuration* meta-model that contains three main abstractions: *configuration space, configuration role,* and *configuration actor*. These three abstractions are at the core of our approach as they enable teamwork in product configuration. Configuration spaces are the means to split the set of decisions in the feature model into simpler modular units. Currently, we have defined that configuration spaces correspond to sub-trees of the feature model (see the TreeConfigurationSpace element in Figure 7). Each configuration space is assigned to a single configuration role that is in charge of making configuration decisions within the space. The rationale to assign configuration spaces to configuration roles can be related to factors such as required knowledge or authority to make decisions within the space. Configuration actors are the actual people that play configuration roles.

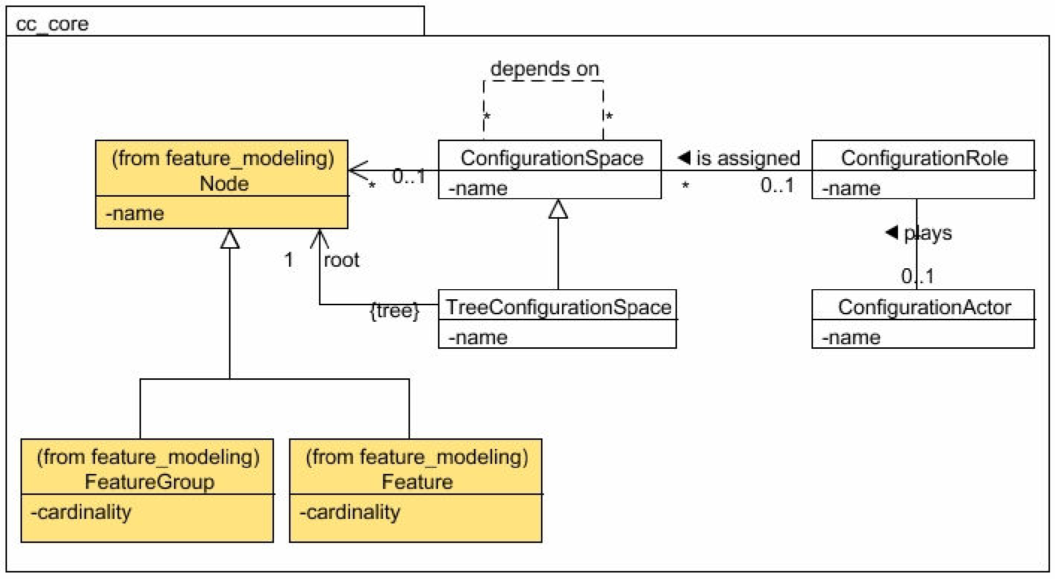


Figure 7: A collaborative configuration model and its relation to feature model concepts

The abstractions depicted in Figure 6 and Figure 7 allow us to draw decorated feature models as the Web-Portal feature model shown in Figure 8. In addition to features and feature groups the feature model also contain explicit references to configuration spaces (*W, P, G, S, N,* and *F*), configuration roles (*project manager, database manager, web designer,* and *security specialist)*, and configuration actors (*a1, a2, a3, a4, a5* and *a6*). Configuration spaces group configuration decisions based on a particular criterion and are later assigned to proper configuration roles. The space-role assignment process should enforce the selection of the best-fit role to make decisions taking into consideration the nature of decisions represented in the configuration space. For instance, in Figure 8 persistence-related decisions encompassed by configuration space *{P}* were assigned to the *database manager* role given his credentials to make decisions in the domain of knowledge.

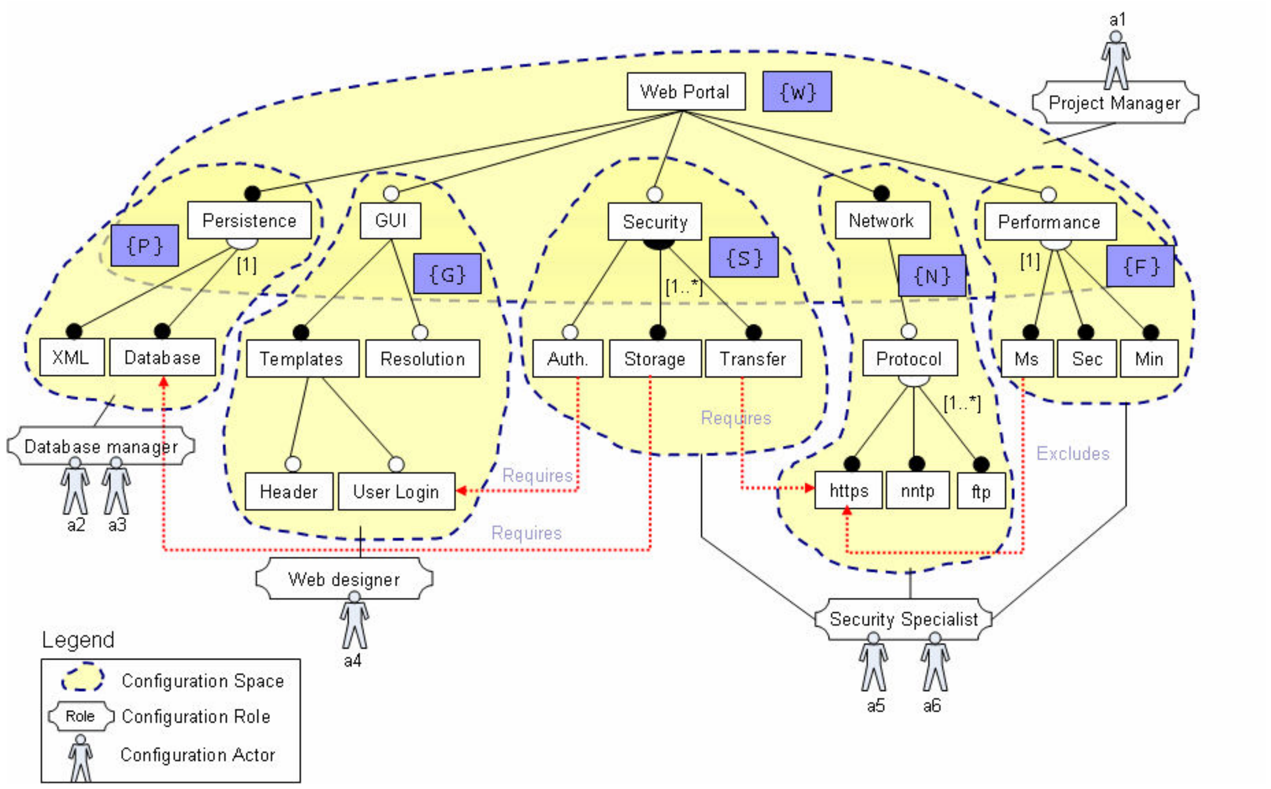


Figure 8: A Web-Portal feature model extended with configuration spaces, roles and actors.

The example shows that the *database manager* role is in charge of persistence configuration decisions while the *web designer* role is responsible for GUI-related decisions. Multiple configuration spaces can be assigned to a single configuration role as in the case of the *security specialist* role. The *project manager* role makes high-level decisions that may or may not require further specialized decisions. For instance, if feature *GUI* is not selected by the *project manager* role the *web designer* role has no decisions left and thus does not take part in the configuration process. The configuration spaces depicted in Figure 8 seem to follow a knowledge-based strategy in which each role is assigned a set of configuration decisions in a particular domain of knowledge (e.g., database, GUI, system security/networking). Likewise, decision power is differentiated as in the case of the *project manager* role that may cause other roles to be left out of the configuration process.

The red dashed arrows that connect two or more features highlight feature dependencies. In the example, a decision to select feature *storage* will require decision propagation to also select feature *database.* However, features *storage* and *database* belong to different configuration spaces and in the case of the Web-Portal example assigned to different configuration roles. Consequently, if feature *storage* is selected by the *security specialist* role feature *database* also has to be. However, the decision of the *database manager* role may be the opposite, i.e., to unselect feature *database* and thus unselect feature *storage*. This is known as a decision conflict and represents a major issue in collaborative configuration as we will discuss later in this document.

Operator	**Semantic**
tree(f)	Extracts a sub-tree of the feature model rooted by *f*
combine_tree(f1,f2,…,fn)	Extracts the smallest sub-tree enclosing the sub-trees rooted by f1,..., fn
cut_tree(f, {f1,f2,..,fn})	Extracts a sub-tree rooted by *f* but containing *f1, f2, ..., fn* as leaves
ind_tree(f)	Extracts the smallest ***independent*** sub-tree starting at feature *f* (i.e., a sub-tree that can be configured entirely independently)

Table 1: Preliminary list of operators to represent configuration spaces in feature models

Configuration Space	**Operator**
{W}	*cut_tree('Web-Portal', {'persistence','gui',''security','network','performance' })*
{P}	*tree('persistence')*
{G}	*tree('gui')*
{S}, {N}, {F}	*tree('security'), tree('network'), tree('performance')*

Table 2: Describing the configuration spaces in Figure 8 using the operators in Table 1

As we mentioned, configuration spaces contain a set of configuration decisions grouped according to particular criteria. They are the primary means for assigning configuration decisions to configuration roles. We understand that configuration spaces can be as simple as sub-trees in the feature model associated with a particular domain of knowledge, for example. Splitting a feature model into tree-like configuration spaces has several advantages. Firstly, the splitting process is relatively straightforward as it follows the natural hierarchical structure of the feature model tree. Secondly, it is natural to think of a feature model as a composition of other simpler feature models in the same sense that a product-line may be thought of as a combination of simpler product-lines (e.g. sub-systems, components). In this case, each component supplier could be responsible for configuring the branch of the feature tree that corresponds to the component they provide. Lastly, viewing configuration spaces as sub-trees of the feature model tree also simplifies establishing the order of execution of the configuration tasks basically by enforcing that parent configuration spaces have to be configured prior to their children. We propose some preliminary operators to help representing configuration spaces in feature model trees. A list with the operators is presented in Table 1. We could use the operators described in Table 1 to represent the configuration spaces depicted in Figure 8 as follows in Table 2.

3.3.1.2 Dependency of configuration tasks

As shown, feature models encompass a set of configuration decisions that need to be addressed in order to configure software products. To cope with the complexity of handling multiple configuration domains that potentially requires people with proper authority and/or specialized knowledge to participate in the product configuration decision-making we proposed a *divide-and-conquer* approach based on the concepts of configuration spaces and roles. That is, an initial feature model is partitioned in smaller components called configuration spaces which group inter-related configuration decisions and are subsequently assigned to multiple configuration roles. In this scenario, the complexity of the collaborative configuration process is substantially dependent on the arrangement of the configuration spaces and roles proposed. In fact, there is a trade-off between teamwork coordination and the complexity of the decision-making. For instance, in a fine-grained partitioning scenario where a large number of configuration spaces and roles are indicated we expect the decision-making to be facilitated as the number of configuration decisions per role ratio is reduced and the assignment of domain-specific decisions to skilled people is improved. On the other hand, the higher the number of configuration spaces/roles the higher the likelihood of increasing the degree of coupling among playing roles.

3.3.1.3 Validation of configuration spaces, roles, and actors

According to our definition configuration spaces must be sub-trees of the feature model. In the following we describe the rules for specifying valid configuration spaces, role, and actors.

- A configuration space must be a tree, i.e., each node must contain a single root feature except for the root node that may node have a parent
- The only allowed overlapping between two configuration spaces are the root/leaves features as in the case of parent/child or siblings configuration spaces
- The root feature of a configuration space represents a point of connection with other parent/sibling configuration spaces.
- All grouped features in feature group must be included as an integral part of a single configuration space.

- A given configuration space can only be assigned to a single role
- A configuration role may be assigned multiple configuration spaces
- A configuration role can be played by multiple configuration actors and a configuration actor can be play many roles
- Configuration spaces, roles and actors can be uniquely identified

The validation rules discussed can be expressed, for example, in terms of UML constraints attached to the collaborative configuration meta-model in Figure 7. It is important to notice that the validation rules proposed reflect the current state of our research and may be subject to change as the research progresses.

3.3.1.4 Representation of collaborative product configuration scenarios

Once configuration spaces and roles are specified and validated the next step is to design a process model to describe collaborative configuration scenarios. At the core of the process model is work coordination and a successful model is one that minimizes conflicting situations, optimizes parallel and independent work, and clearly describes the configuration roles and their responsibilities in the configuration process. In addition, a collaboration process model should ideally be executable, i.e., allow an external tool to parse and execute its operations thereby running an actual collaborative configuration scenario.

An intriguing question in our research regards the way we should describe collaborative configuration processes. It seems that some concepts can be borrowed from the area of process languages and workflows as they have been largely used as a means to describe activities carried out by teams of machines and/or humans. Coordination has been at the core of these languages to help coping with work interdependency and to enforce consistency. For instantce, by using a workflow language one is able to describe how individual and shared resources are handled by a group of people as they carry out their activities towards a specific goal. Activities can be performed in a sequence when they expose dependencies or in parallel when work can be performed independently. In some cases the outcomes produced by independent activities need to be merged into a consistent state. In the end of the process it is expected that a goal has been reached and some resources eventually produced.

The context of collaborative configuration is quite similar. A team of humans with special skills are assigned partial configuration tasks in order to produce intermediate resources that are consistently merged to produce a single final outcome, i.e., a valid product specification. Even though the use of workflows seem to be quite applicable in the context of our approach it will certainly require precise semantic definitions for particular operations. For instance, a *merge* operation is a generic operation that combines two or more inputs and produces a single output that represents a consistent combination of the input elements according to some validation rules. In practice, as in the case of feature-based collaborative configuration the semantics of the *merge* operation has to be defined precisely to avoid ambiguity and errors.

Operation/Symbol	**Name**	**Description**
$\longrightarrow$	Sequential flow operator	Arrange configuration sessions in a sequence
$\|$	Parallel flow operator	Parallelize configuration spaces decision-making
MERGE-DECISIONS - PRIORITY-MERGE - MINIMIZE-CHANGES-MERGE	Merge operator - Priority merge - Minimize changes	Merge configuration spaces decisions - Specify priorities for merging decisions - Minimize changes over decisions made
MANUAL-MERGE	Manual merge	Perform a real-time manual merge requiring configuration roles to resolve a conflict

Table 3: Possible operators of a collaborative product configuration process language

Manual *versus* Automatic Merge

- A manual merge is performed by humans with machinery assistance
- An automatic merge is fully automated based on a conflicting resolution strategy

Group Awareness

- When awareness is *on* configuration roles will be notified about decisions that may impact their configuration spaces and how their own decisions impact others
- When awareness if *off* configuration roles will work entirely independently from each other even when their collective decisions are inconsistent (solved by a merge later on)

```
⌂ Goal: Find a solution to a CSP problem that best approximates given sets of variable assignments
⌂ Parameters:
⌂   - P: A CSP problem
⌂   - C: additional constraints over P variables not yet added to P
⌂   - S: a current solution for P (not considering constraints C)
⌂   - H: an assignment over a sub-set of P variables
⌂ Pre-conditions:
⌂    - P is satisfiable
⌂    - S is a solution for P
⌂    - A is a sub-set of S (A ⊆ S)
⌂ Return: a solution R that satisfies P (including constraints C) and best approximates the assignments in A
MERGE-DECISIONS ( P , C : Constraints; S , A : Assignments ) → R : Assignments
1. begin
```

```
2.  add $C$ to the set of constraints of problem $P$
3.  ⌂ *if S satisfies problem P returns S*
4.  **if** SAT(P,S)
5.    **then** R ← S
6.    ⌂ goal: *find a solution that best approximates A*
7.    **else**
8.      **then** maxA = 0, maxS = 0
9.            ⌂ *check all solutions X that satisfies P*
10.           **for each** solution X where SAT(P,X)
11.             **if** (S = { })
12.                 **then** S ← X
13.                 ⌂ *temporarily saves the solution that best approximates A*
14.                 **else if** count(X ∩ A) > maxA
15.                       **then** S ← X
16.                            maxS ← count(X ∩ S)
17.                            maxA ← count(X ∩ A)
18.                       ⌂ *enforce that the solution also minimizes changes in S*
19.                       **else if** count(X ∩ A) = maxA
20.                             **then if** count(X ∩ S) > maxS
21.                                   **then** S ← X
22.                                        maxS = count(X ∩ S)
23. **end**
```

Example function calls:

Priority Merge = MERGE-DECISIONS(P_1 U P_2, C_{1-2}, S_1 U S_2, S_1)

Minimized Overall Changes Merge = MERGE-DECISIONS(P_1 U P_2, P_{1-2}, S_1 U S_2, S_1 U S_2)

Currently, we have identified some elements of the collaborative product configuration process language typically control flow and merging operators as described in Table 5. A proof-of-concept implementation of the merge operation is shown in the previous page and supports both priority and minimize-changes merges. The algorithm starts declaring the merge function Merge-Decisions. The parameters are: P: a CSP problem; C: an additional constraints over P variables not yet added to P; S: a current solution for P (not considering constraints C); H: an assignment over a sub-set of P variables. The objective of the function is to find a solution for problem P including constraints C as so to minimize the changes in S based on the assignments on H. Line 2 adds constraints C to problem P. If S satisfies P then the algorithm stops (lines 4 and 5). Lines 10-17 search for a solution for P that best approximates the variable assignments in H. Lines

19-22 enforce that the solution should ideally minimize changes in S too. The result assignment R contains a solution for P that minimizes changes in H, primarily and in S, secondarily. The function uses a brute-force algorithm that tests all solutions for P and thus may be limited in practical situations. However, the function provides the right interface we are looking for in the approach to merge configuration specifications based on priority or minimization of changes. A goal in the approach is to know the scalability of the function and try to optimize it later, even though our research is not directly related to algorithms and optimizations.

What makes describing collaborative configuration particularly challenging is its dynamic nature. For instance, an anticipated dependency between two configuration spaces, say C_a and C_b, that led them to be arranged and executed in sequence may have disappeared during the configuration process as a consequence of previous decisions suggesting now that the configuration spaces should be executed in parallel. The question of whether it is possible to optimize the collaborative configuration process dynamically by analyzing the network of dependencies among the various configuration spaces and roles available is a future research target.

Currently, the approach to describe collaborative configuration is based on the concepts of configuration spaces and configuration roles as well as on analyzes of their dependencies. We have considered two basic relationships between configuration spaces: *order* and *dependency*. A *dependency* relationship occurs when two configuration spaces have decisions that may affect each other. For instance, if the selection of feature F_a in the configuration space C_a *requires* the selection of feature F_b in configuration space C_b we say C_a *depends on* C_b, and vice-versa. If configuration spaces C_a and C_b are assigned to different configuration roles then a potential conflicting scenario is characterized thus requiring a proper coordination strategy to be put in place. As mentioned, a possible solution would be to arrange the configuration spaces in a sequence thus avoiding conflicts. However, a clear disadvantage of sequences is that chaining configuration spaces may delay decisions that could be made earlier in the process as they do not expose any dependencies. In other words, if the number of dependencies is low then ideally the configuration spaces should be handled in parallel and merged in the end. In the eventual case of a conflict during the merge a special negotiation phase would take place. All decisions regarding the arrangement of the configuration spaces/roles might also consider additional factors such as organizational hierarchy and time-to-market constraints.

The *order* relationship is a strong type of dependency between two or more configuration spaces. In fact, we view *order* as a parent-child relationship where *parent* configuration spaces encompass decisions that may either activate or deactivate the *children* configuration spaces. Activating a child configuration space means the decisions within the space must be handled by a configuration role. Contrarily, deactivating a configuration space means that all decisions within the space were automatically made to a particular state, for example, all features were deselected. The *order* relationship can be derived from the hierarchical order of the configuration spaces in the feature model considering a scenario where configuration spaces are always structured as trees. In Figure 8, we say that configuration space *{W}* is handled before configuration spaces *{G}*, *{S}* and *{F}* considering that decisions in *{W}* may or may not require further decisions to be handled on *{G}*, *{S}* or *{F}*.

3.3.1.5 Validation of collaborative product configuration scenarios

Another intriguing point in collaborative configuration regards the validation of configuration scenario descriptions. That is, the order in which configuration decisions are made is fundamentally important to avoid inconsistencies or other undesirable scenarios such as deadlocks. A possibility to support the validation of configuration decisions arrangements is to specify constraints to enforce certain properties regarding the *order* and *dependency* of configuration spaces. In fact, configuration spaces encompass configuration decisions that can be either internal (do not affect any other roles' configuration space) or external (may affect other roles' configuration spaces).

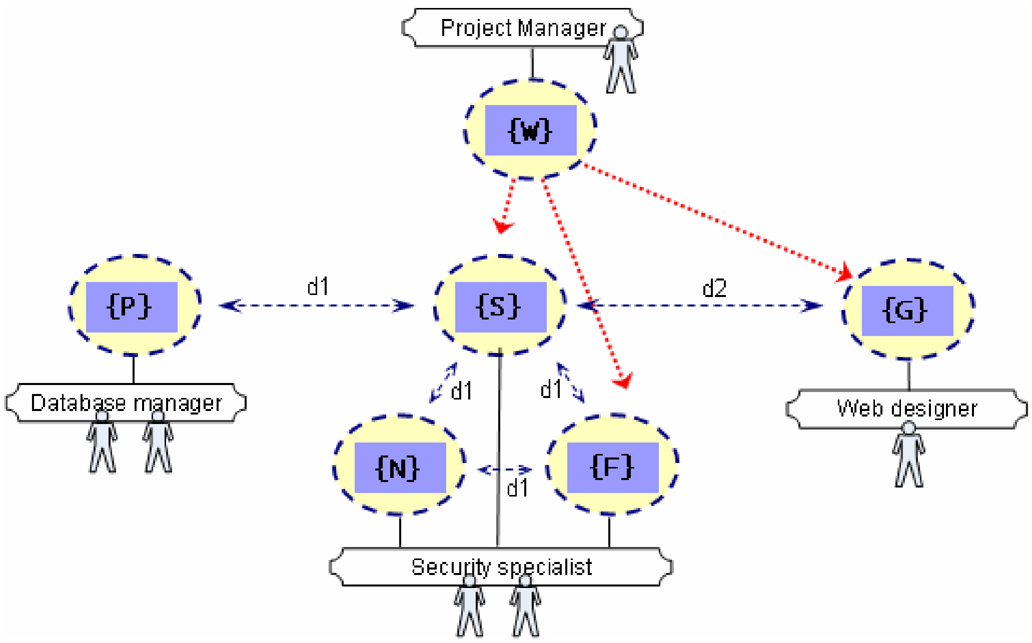


Figure 9: Web-Portal feature model - configuration spaces and role compact view

For instance, Figure 9 represents a simplification of the of the Web-Portal feature model depicted in Figure 8 that highlights specified configuration spaces and roles. From the arrangement of the elements in Figure

9 it is possible to generate constraints to validate collaborative configuration scenarios for the Web-Portal product line as follows:

Collaborative Configuration Process Constraints	**Type of Relationship**
1. {W}.pm : {G}.wd, {S}.ss, {F}.ss	Order
2. {P}.dm ↔ {S}.ss ↔ {N}.ss ↔ {F}.ss	Dependency
3. {G}.wd ↔ {S}.ss	Dependency
4. {W}.pm : {P}.dm (derived from 1 and 2)	Order
5. {W}.pm : {N}.ss (derived from 1 and 2)	Order

Table 4: Constraints to validate collaborative scenarios for the Web-Portal product line

Where:

{ds}.cr - represents the configuration space *ds* assigned to configuration role *cr*

X : Y implies that *X is handled before Y*, and

X ↔ *Y* implies that *X's* decisions *depends on Y's* decisions and vice-versa.

In constraint *1)* we say that configuration space *{W}.pm* is *handled before* configuration spaces *{G}.ss, {S}.ss,* and *{F}.ss* considering their arrangement in the feature model. It means that the *project manager* role's decisions on configuration space *{W}.pm* will define whether or not roles *web designer* and *security specialist* will handle decisions on configuration spaces *{G}.ss, {S}.ss,* and *{F}.ss*.

Furthermore, by analyzing the additional feature constraints in the feature model of Figure 8 we find out dependency paths (see elements *d1* and *d2* in Figure 9) among configuration spaces that allow us to express dependency relationships such as in *2)* and *3).* These constraints convey that decisions in configuration spaces *{P}.dm*, *{S}ss, {N}.ss and {F}.ss* as well as in *{G}.wd* and *{S}.ss* may affect each other thereby requiring a coordination strategy to be put in place.

Constraints *4)* and *5)* were derived from constraints 1) and 2) to indicate that configuration space {W}.pm is also handled before configuration spaces {P}.dm and {N}.ss. The derivation comes from the fact that whenever three configuration spaces A, B, and C are such that A is handled before B and B depends on C we expect A to also be handled before C. In other words, the order relationship is stronger than the dependency relationship when the same set of configuration spaces is considered.

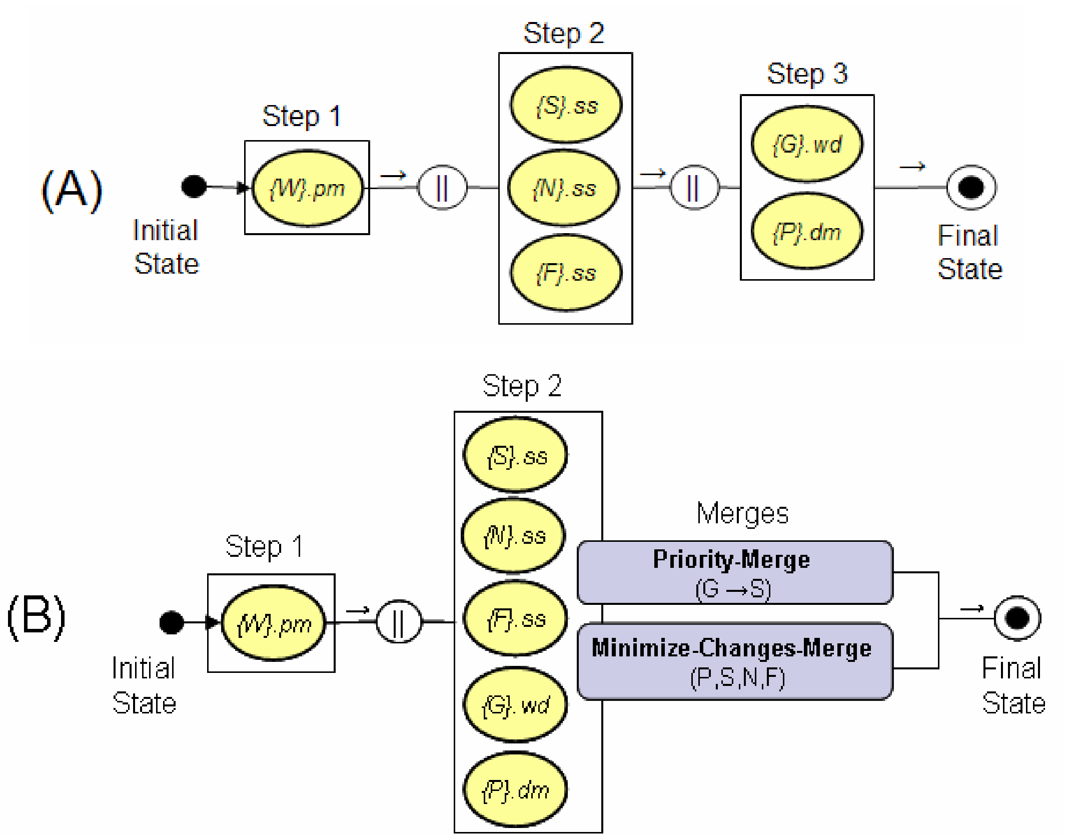


Figure 10: Two collaborative configuration scenarios (A and B) for the Web-Portal product line

Feature	Abbr.	Feature	Abbr.	Feature	Abbr.
Authentication	au	Ms	ms	Sec	se
Database	da	Network	ne	Security	sc
FTP	ft	NNTP	nn	Storage	st
GUI	gu	Performance	pe	Templates	te
Header	he	Persistence	ps	Transfer	tr
Https	ht	Protocol	pr	User Login	ul
Min	mi	Resolution	re	XML	xm

Table 5: Abbreviations of feature names for the Web-Portal product line

Figure 10 illustrates two possible collaborative configuration scenarios for the Web-Portal product line. Both scenarios respect the *order* and *dependency* constraints described in Table 6 based on the configuration space and roles arrangements described in Figure 8. It basically specifies order constraints (e.g., *{W}* is handled before *{S}*) and the need of a merge operation whenever depending configuration spaces assigned to different configuration roles are handled in parallel (e.g. *{G}* and *{S}*). Now, let us discuss how a product specification can be achieved for each of the scenarios depicted in Figure 10. To improve readability we will use abbreviated feature names for the Web-Portal feature model's features as described in Table 7.

In scenario-1 the goal was to discourage negotiation among configuration roles by sequencing the configuration spaces that could potentially lead to conflicts, i.e., spaces of different configuration roles but that expose dependencies. Constraints 2) and 3) of Table 6 illustrate these configuration spaces. In line 2 of scenario-1 the collaborative configuration scenario begins. Line 2 describes the initial state of the product specification indicating that features *ne* and *ps* are initially selected by the system as they are always *true* for all products in the Web-Portal product line. In line 3, step-1 is started where the configuration role *project manager* makes decision on configuration space *{W}.pm*. Line 4 shows the decisions of the *project manager* that selected features *gu, sc,* and *pe*. Underlined features *ps* and *ne* represent automatic decisions either made by the system or as the result of the propagation of previous decisions. The selection of feature *gu* triggered the automatic selection of feature *te* in the configuration space *{G}* of the configuration role *web designer* as represented by the expression gu*[{G}.wd:te]* in line 4.

Collaborative Configuration for the Web-Portal Product Line
Scenario 1: Figure 10.A

```
1.   BEGIN
2.     Initial state: (ne, ps)
3.     Step: 1
4.       {W}.pm = (ne, ps, gu[{G}.wd:te], sc, pe)
5.       Commit: gu, te, sc, pe
6.       Current state: (ne, ps, gu, te, sc, pe)
7.     Step: 2
8.       {S}.ss = (au[{G}.wd:ul], st[{P}.dm:da], {P}.dm:~xm], ~tr )
9.       {N}.ss = (pr, ~ht, nn, ft)
10.      {F}.ss = (ms[~se, ~mi], ~se, ~mi)
11.      Commit: au, ul, st, da, ~xm, ~tr, pr, ~ht, nn, ft, ms, ~se, ~mi
12.      Current state: (ne, ps, gu, te, sc, pe, au, ul, st, da, ~xm, ~tr, pr, ~ht, nn, ft, ms, ~se, ~mi)
13.    Step: 3
14.      {P}.dm = (da, ~xm)
15.      {G}.wd = (te, he, ul, re)
16.      Commit: he, re
17.      Final state: (ps,gu,sc,ne,pe,~xm,da,te,re,he,ul,au,st,~tr,pr,~ht,nn,ft,ms,~se,~mi)
18.  END
19.  Final Specification:  (ps,gu,sc,ne,pe,~xm,da,te,re,he,ul,au,st,~tr,pr,~ht,nn,ft,ms,~se,~mi)
20.    {W} = (ps, gu, sc, ne, pe)
21.    {P} = (~xm, da)
22.    {G} = (te, re,he, ul)
23.    {S} = (au, st, ~tr)
24.    {N} = (pr, ~ht, nn, ft)
25.    {F} = (ms, ~se, ~mi)
```

In Line 5, the *project manager* decisions are committed to the feature model after validating that all mandatory decisions were made and that none of the decisions violate the constraints of configuration space *{W}*. In addition, it is very important to check whether the decisions made by the *project manager* are leading the configuration process to a valid configuration state. In other words, the system needs to check whether the remaining constraints are still *satisfiable*, i.e., there is at least one set of feature decisions that satisfy the constraints left. Line 6 shows the partial state of the product specification after the *project manager* decisions on configuration space *{W}* have been made. Line 7 starts step-2 of the configuration process that involve configuration spaces *{S}*, *{N}*, and *{F}*. These three sets were grouped together strategically because they are under the responsibility of the same configuration role, i.e., the *security specialist*. As a consequence, there is no need of merging the decisions of these configuration spaces as the system assumes the *security specialist* will be aware of decision propagation within these spaces. In fact, we could see these three configuration spaces as a single *{S,N,F}* configuration space. Lines 8, 9 and 10 show the decisions of the *security specialist* regarding security (*{S}*), network (*{N}*), and performance (*{P}*), respectively, as well as the corresponding propagations. For instance, the selection of feature *au* triggered the selection of feature *ul* in the *web designer* configuration space *{G}*. Because the strategy enforced in this scenario was the sequencing of potentially-conflicting configuration spaces the propagated decision to select feature *ul* will hold until the end of the configuration process thereby not allowing the configuration role *web designer* to make any changes. In line 11 the *security specialist* configuration decisions are committed after proper validation. Line 12 shows the partial state of the product configuration after step-2 distinguishing between the features that were manually and automatically (underlined) decided. Step 3 starts at line 13. All previous decisions made hold at this stage. As a consequence no decision was left open in configuration space *{P}* as illustrated in line 14 where features *da* and *xm* are underlined. Hence, there is no need for configuration role *database manager* to get involved in the process according to the rules of scenario-1. In line 15, configuration role *web designer* selects features *he* and *re*. Feature *te* and *ul* represent propagations and cannot have their selection states changed. In line 16 step-3 decisions are committed. Note that only two features (*he* and *re)* are illustrated since all other features represent propagations. Line 17 shows the final state of the product specification pointing out selected and deselected features as well as human and automatic/propagated decisions. By following the propagation paths (e.g., au*[{G}.wd:ul]*) is possible to trace back the decisions that caused a particular feature to be included or excluded in the product specification. For instance, the inclusion of feature *ul* was caused by the decision of the configuration role *security specialist* to include feature *au.*

Thus, feature *ul* is found highlighted in the final specification in line 19. Lines 20-25 show feature decisions in each configuration space.

In scenario-2 the goal was to achieve maximum parallelism and postpone conflict resolution to a later stage. In addition, automatic merging was chosen indicating that conflict resolution is performed automatically based on particular merging strategies. Similar to scenario-1 in the first step the *project manager* role started making top-level decisions. Line 4 shows the decisions made in step-1 which are similar to those in scenario-1. In line 5 step-1 decisions are committed and the current state of the product decisions is shown in line 6. Step-2 parallelized five depending configuration spaces ({S}, {N}, {F}, {P} and {G}) assigned to three distinct configuration roles (*security specialist, database manager*, and *web designer*). Because {S}, {N} and {F} are assigned to the same configuration role and are part of the same step (*security specialist*) explicit propagation occurs within the boundaries of this configuration spaces. In other words, these configuration spaces can also be seen as a single {SNF} configuration space that represent the union of all features, constraints and decisions of {S}, {N}, and {F}. Lines 8-12 show the decisions made in each configuration space and the effects of their propagations without crossing the boundaries of each configuration space. In a realistic scenario we can either enable activity awareness so that decision roles are aware about others decisions and their impact on their own decisions or disable awareness and make decision roles solely concentrate on their own decisions.

Collaborative Configuration for the Web-Portal Product Line - Scenario 2: Figure 10.B

```
1.  BEGIN
2.     Initial state: (ne, ps)
3.     Step: 1
4.       {W}.pm = (ne, ps, gu[{G}.wd:te], sc, pe)
5.       Commit: gu, te, sc, ps
6.       Current state: (ne, ps, gu, te, sc, pe)
7.     Step: 2
8.       {S}.ss = (au[{G}.wd:ul], st[{P}.dm:da, {P}.dm:~xm], ~tr )
9.       {N}.ss = (pr, ~ht[{S}.ss:~tr], nn, ft)
10.      {F}.ss = (ms[~se, ~mi], ~se, ~mi)
11.      {P}.dm = (~da[xm], xm)
12.      {G}.wd = (te, he, ~ul[{S}.ss:~au], re)
13.    Merges step
14.      Merge 1 – Priority merge of {S} and {G}
15.        SGConstraints = {S}/constraints U {G}/constraints
16.        SGLinkingConstraints = LinkingConstraints({S},{G})
17.        SGSolution = {S}/decisions U {G}/decisions
18.        MERGE-DECISIONS(SGConstraints, SGLinkingConstraints, SGSolution, {S}/decisions)
```

```
19.             = Conflict: {S} U {G} = (au, ul, st, ~tr) U (te, he, ~ul, re)
20.             = (au, &ul, st, ~tr, te, he, re)           // conflict solved
21.        Merge 2 – Minimum changes merge of {P}, {S,N,F}
22.            {SNF} = {S}U {N} U {F}
23.            PSNFConstraints = {P}/constraints  U {SNF}/constraints
24.            PSNFLinConstraints = LinkingConstraints({P}, {SNF})
25.            PSNFSolution = {P}/decisions U {SNF}/decisions
26.            MERGE-DECISIONS(PSNFConstraints, PSNFLinConstraints, PSNFSolution, PSNFSolution)
27.               = Conflict: {P} U {SNF} = ( ~da, xm ) U ( … da, ~xm, …)
28.                  = Attempt #1: keep “da” ? (~xm)  // 1 change
29.                  = Attempt #2: keep “~da” ? (~st, tr, ht, ~ms, se) or (~st, tr, ht, ~ms, mi) // 5 changes
30.                  = (au, ul, st, &da, &~xm, ~tr, pr, ~ht, nn, ft, ms, ~se, ~mi)  // conflict solved
31.        Merge1 U Merge2  = (au,ul,st,&da,&~xm,~tr,pr,~ht,nn,ft,ms,~se,~mi,te,he,re)
32.     Commit: (au,ul,st,&da,&~xm,~tr,pr,~ht,nn,ft,ms,~se,~mi,he,re)
33.     Final state: (ps,gu,sc,ne,pe,&~xm,&da,te,re,he,ul,au,st,~tr,pr,~ht,nn,ft,ms,~se,~mi)
34. END
35. Final Specification: (ps,gu,sc,ne,pe,&~xm,&da,te,re,he,ul,au,st,~tr,pr,~ht,nn,ft,ms,~se,~mi)
36.    {W} = (ps, gu, sc, ne, pe)
37.    {P} = (&~xm,&da)
38.    {G} = (te, re,he, ul)
39.    {S} = (au, st, ~tr)
40.    {N} = (pr, ~ht, nn, ft)
41.    {F} = (ms, ~se, ~mi)
```

According to the model in Figure 10-B, merges are necessary to solve eventual decision conflicts. Two merges were anticipated based on different conflict resolution strategies. The first merge operation in line 14 is a priority merge where decisions in {S} will prevail over decisions in {G}. To achieve the priority merge we used our MERGE-DECISIONS operation described previously. The parameters to the merge operation are the local constraints of {S} and {G} (line 15), the constraints that link {S} and {G} and not taken into account during step-2 so far (line 16), and the decisions made in {S} and {G} represented by the *SGSolution* set in line17. The merge function call in line 18 passes these sets as parameters and the last paramater indicates that the merge operation should preserve as much as possible *security specialis* role's decisions in {S}. Line 19 shows that there is a decision conflict while trying to combine the decisions in {S} and {G}. The decision of the *security specialist* in selecting feature *au* indicates triggered the selection of feature *ul* which is incompatible with the decision made by the *web designer* to deselect *ul.* The conflict was solved through the merge operation by selecting *ul* and respecting the priority of {S}'s decisions over {G}'s decisions. In Line 20, *ul* is preceded by an *&* indicating that the feature was decided by the merging operation. The second merge in line 21 aimed at merging decisions in {P}, {S}, {N} and {F} minimizing the changes of previous decisions made. The call to the MERGE-DECISIONS operation is similar to the previous one except that this time a set with all combined decisions of {P}, {S}, {N} and {F} are passed as the last parameter indicating that the merge should look for a set of decisions that satisfies all constrains (line 23) and linking constraints (line 24) involving {P}, {S}, {N} and {F} and preserve as much as possible the decision already made. Line 27 shows a decision conflict between {P} and {S} regarding features *da* and *xm*. Attempts to combine decisions made on {P} and {S} taken propagations into account on {N} and {F} are shown in lines 28 and 29. The first attempt (line 28) focuses on preserving the decision to select *da* which leads to a change to *xm* that had to be deselected. Thus, one change is required. In the second attempt *~da* (deselection of feature *da)* is enforced and propagations change 5 other decisions made in {S}, {N} and {F} (features *st, tr, ht, ms, se*). Hence, the first attempt represents the solution to the merge operation as it minimizes the changes on decisions previously made (line 30). In line 31 the two merge solutions are unified and committed in line 32. The final specification and the decision per configuration space are shown in lines 35-41. Again, we indicate by the operator *&* the decisions that were eventually changed during the merge operation so that the final configuration (line 35) explicitly indicates which decisions were changed through the merging. For instance, features *da* and *xm* are preceded by the *&* operator to indicate that the *database manager's* original decisions on these features were changed in merge 2 (lines 21-30).

In addition, it is interesting to notice that even though resulting product specifications in scenarios 1 and 2 are identical they were achieved through different strategies. In scenario-1 conflicting configuration spaces were serialized to avoid merges as opposed to scenario-2 where parallelism was enforced and conflicts were solved through different merge policies.

3.3.2 Executing Collaborative Product Configuration Scenarios

In this section we describe the dynamics of collaborative configuration taking into account five key concepts: coordination, conflict resolution support, awareness, communication and traceability. However, let us start by discussing the elements that support the dynamics of collaborative configuration scenarios.

3.3.2.1 An execution model for collaborative configuration scenarios

We propose a model packaged as *cc_dynamics* in Figure 11 to support collaborative configuration scenarios. The *ConfigurationScenario* element represents configuration scenarios and may encompass several ordered configuration steps (*ConfigurationStep* element). Configuration steps enforce order constraints among configuration decisions as discussed in a previous session. For instance, multiples configuration steps are shown in Figure 10, scenarios A and B. As multiple configuration roles may be playing in the same step the concept of a configuration session (*ConfigurationSession*) is suggested. A configuration session grants configuration roles a safe and independent place to make decisions without the risk of damaging any other playing roles. That is, a session checks in a copy of the current state of the feature model locally and allows for temporary inconsistencies.

The configuration role in charge of the session will configure all configuration spaces associated with the session. For instance, on step-2 of scenario-B in Figure 10 a configuration session will be created containing configuration spaces *{N}*, *{S}*, and *{F}* and assigned to configuration role *security specialist* as this role is in charge of all three configuration spaces. All decisions propagated from previous steps will appear as *immutable*. Likewise, only configuration spaces that can be activated will be available for configuration, i.e., those that had their root feature selected by the *project manager*. For instance, configuration space *{S}* will only appear as *active* if the *project manager* has selected feature *Security*. Session decisions can only be committed when all mandatory decision were made and validated.

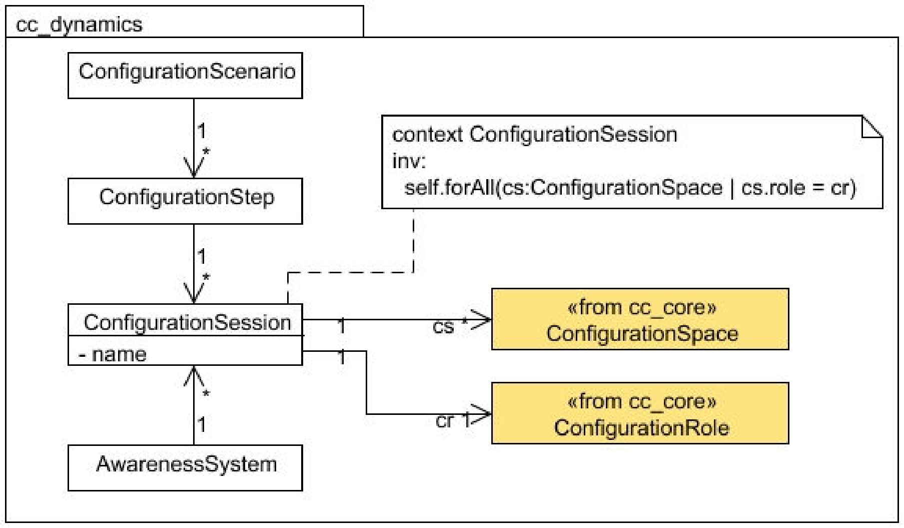


Figure 11: Model elements to support the dynamics of collaborative configuration

In addition, a session commit is only made to the master copy of the feature model after all prescribed merging operations were performed. It means that in Figure 10 Scenario-B the two merges have to be successfully performed before committing step-2 decisions to the centralized master copy of the feature model. Notice that even though sessions isolate configuration roles an awareness system (*AwarenessSystem* element) takes care of keeps configuration roles aware of each other's activities. The awareness systems works as a event propagator within the collaborative configuration system broadcasting relevant events to all active configuration sessions. Again, considering *scenario-B* in Figure 10, when the *security specialist* role decides to select feature *authentication* the awareness system sends an event to the *web designer*'s configuration session to notify that feature *user_login* is of interest to another role. That is, if feature *user_login* is not selected a decision conflict will happen. In this case, awareness may encourage the *web designer* to select feature *user_login* and avoid a conflict or even to start a conversation session with the *security specialist* role to investigate decision rationales and other alternatives.

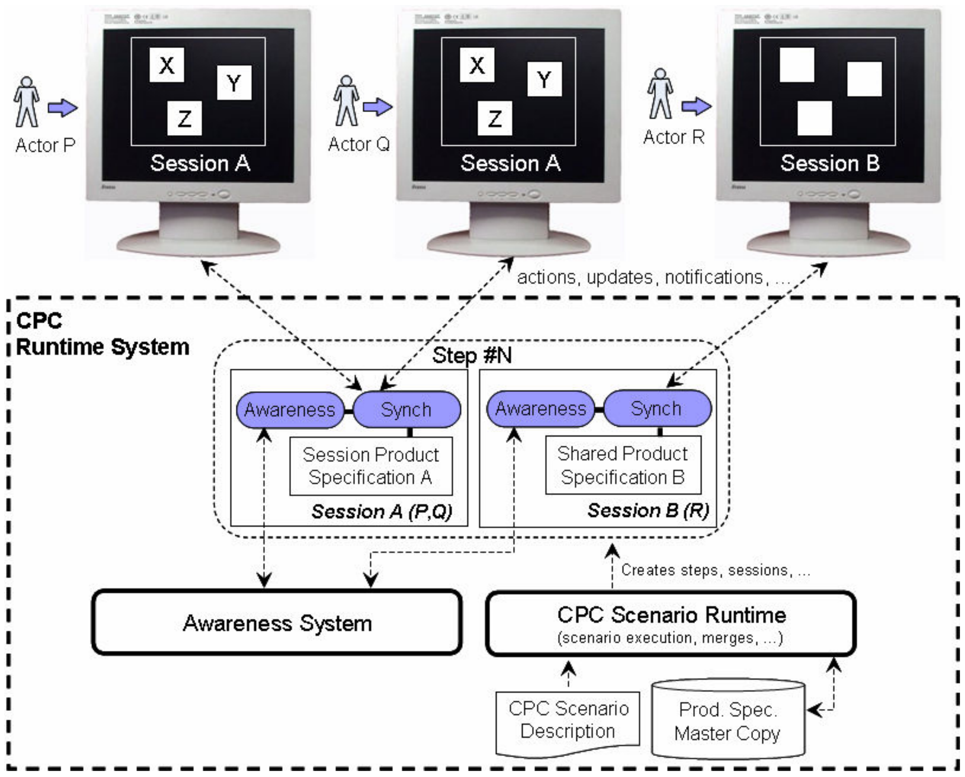


Figure 12: Dynamic view of the CPC runtime system

3.3.2.2 Arrangement of CPC runtime components

Figure 12 illustrates how dynamic CPC model elements depicted in Figure 11 can be arranged as part of a runtime distributed system for CPC scenario descriptions. Figure 12 shows one configuration step (#*N*) containing two configuration sessions (*A* and *B*), three configuration actors (*P*, *Q*, and *R*), and some configuration spaces (*X*, *Y*, and *Z*). Configuration sessions *A* and *B* run concurrently within step #*N*. The *awareness system* takes care of providing activity awareness to actors *P*, *Q*, and *R*. For instance, if actor *R* makes a decision that impact actors *P* and *Q* configuration spaces the later will be notified. Awareness is an important element of coordination and may also help to minimize decisions conflicts. Each configuration session contains a *session product specification*, i.e., a local copy of the *product specification master copy* that allows each session to run independently. However, because multiple configuration actors may be playing the same role they can be operating over the same session at the same time. Therefore, a *synchronization* unit (see *Synch* label in Figure 12) is included to avoid inconsistent concurrent updates to the shared session product specification. For instance, in Figure 12 actors *P* and *Q* are playing the same role on session A and configuring the exact same configuration spaces (*X, Y,* and *Z*). We say that session *A* is a *shared session* to resemble the concept of *shared spaces* in collaboration systems. The main goal of shared sessions is to encourage *cooperation* among playing actors.

Notice that sessions are represented in both the client applications and the central runtime server. In the client sessions are more a visual representation while in the server they enforce consistency and provide a unified view of the session product specification. Each session also contains a local *awareness* component that communicates with the *awareness system* to notify about events that may be of interest to other configuration actors. The *CPC Scenario Runtime* unit is the major component in the system. It reads a *CPC scenario description*, executes its steps, and updates the *product specification master copy*. In the end of the configuration process the *product configuration master copy* contains all actors' decisions and represents the final outcome of the process.

3.3.2.3 Collaborative product configuration and groupware/CSCW concepts

In our approach, support to *coordination* is achieved through ordering configuration steps and facilitating decision negotiations through automatic and manual merges. Collaborative configuration scenarios specified by the product manager can be either highly regulated in which steps have to be performed in a strict order or flexible allowing for maximum parallelism of configuration tasks. Flexibility comes at the price of group work awareness and negotiation among configuration roles to implement coordination. For instance, Figure 10 shows two scenarios with different coordination demands. Scenario-A imposes a strict order of steps and no merge is required. Scenario-B maximizes concurrent work but requires merge operations to combine configuration decisions. Decision merging is automatically achieved in a way that minimizes impacts on previous decisions made.

Awareness is supported by allowing configuration roles to see the impact of their decisions on others configuration spaces and vice-versa. By being aware of others decisions decision roles are able to anticipate and intentionally minimize conflicts. For instance, by being aware that the *database manager* has selected feature *XML* the *security specialist* may avoid a conflict by deselecting feature *data storage* in the Web-Portal product line. At worst, one of the decision sides could start a communication session with the other to discuss about the options to avoid later conflicts. In our approach, awareness is a very important component and thus is supported by a specific module called *awareness system.* The awareness system connects configuration sessions within the same step and is a central component for group work coordination along with communication mechanisms. The implementation of the awareness system relies on logic-based propagation and reasoning systems with the extra burden of supporting distributed configuration environments. It should be noticed that support for collaborative configuration entails the development of a distributed system with a centralized server that enforces decision-making consistency. That is, only decisions that do not violate configuration constraints relying on the configuration server will be accepted and committed.

Conflict resolution is another key component in the approach. In automatic resolution, merge operations will attempt to find solutions that preserve as much as possible decisions made. Examples of such merges include *priority* merges and *minimize changes* merges. Conflicts can also be resolved manually either by an external role such as the product manager or by the roles directly involved in the conflict. For that, we plan to offer tool support benefiting from the merge operations to allows roles to reason on conflicts and how they could change their decisions so that to resolve the conflict. We are planning to develop a *conflict resolution interface* where configuration roles involved in a particular conflict can play together and analyze possible scenarios for resolving a conflict. For instance, the system can suggest various different scenarios to resolve conflict based on the minimization of decision changes or on a given priority rule.

As for *communication* requirements we understand that configuration roles might need to communicate synchronously or asynchronously depending on the situation. In a real-time scenario when configuration roles are making decisions concurrently it may be interesting to allow them to freely interact and thus coordinate their work. We have a particular commitment not to be too formal or restrictive in the process and rather allow configuration roles to eventually decide the best way to go. A messenger-based system augmented with a conflict resolution/reasoning interface is initially what we have planned to support synchronicity. In other situations, asynchronous communication might be more appropriate especially when configuration roles are distributed across different time zones. In this case, configuration roles may still reason on conflict resolution strategies and send asynchronous messages to each other containing particular desired scenarios.

Traceability is the ability to identify the configuration roles and actors that caused a feature to be included in or excluded from a product specification. In non-collaborative configuration traceability is very hard to achieve as there is a single configuration actor in charge of all decisions based on user requirements. Tracing back to identify what requirement caused a feature to be included in a product specification entails mapping requirements to features and can be highly complex. In collaborative configuration traceability can be achieved by linking features to associated configuration decisions. In addition, we need to take into account scenarios involving automatic and/or manual merges as well as decision propagation.

We plan to test the s*calability* of our approach through simulation. For instance, we can simulate the merge of large feature-based product specifications by automatically generating feature models and valid specifications. Likewise, we want to test the scalability of other algorithms such as those to support CPC metrics. For this purpose, the same simulation strategy can be used. Previous studies [41][54] have suggested that even though product configuration can be generally seemed as a constraint satisfaction problem, i.e., NP-complete problems, it is usually a feasible problem when the complexity of feature dependencies in feature models is low. Ultimately, our goal is to better understand the limits of our approach by identifying scenarios where proposed algorithms work best and where they fall short.

Appendix A at the end of this document illustrates a possible XML representation to describe collaborative product configuration scenarios.

3.4 Limitations

Most of the current limitations of our approach are subject to further research and thus were listed in the *Next Research Steps* section.

3.5 Validation

Alternatives to validate the research include the formal verification of CPC properties, the conduction of empirical case studies, and the use of a simulation environment. The primary alternative is to formally verify desirable CPC properties. For instance, *termination* is a desirable property of the CPC runtime system that enforces that every valid CPC scenario execution will come to an end, i.e., will terminate. Likewise, we want to study how we could check for *deadlocks* on CPC scenario descriptions. *Deadlocks* occur when two or more configuration actors are blocked waiting for each other's decisions before they can continue. As expected, *deadlocks* are highly undesirable as they may block the whole configuration process. *Backtracking* is another characteristic of collaboration systems that regards the eventual need of *undoing* past actions and/or decisions in order to restore the overall consistency of a system or process. In the case of collaborative product configuration, *backtracking* can be really expensive requiring a large number of decisions to be undone together because of complex dependencies among them. In certain configuration scenarios, it may be the case that we have to enforce a *backtrack-free* process in a sense that decisions made in the past can never be undone. The point becomes: Can we express or at least prove that a given CPC scenario description is *backtrack-free*?

As for case studies, an initial plan is to run a case study using a large feature model. A possible candidate is the e-Shop feature model described in [17] that contains hundreds of features. Another alternative is to use an expanded version of the Web Portal feature model shown in previous sections of this proposal. Case studies can be very helpful to demonstrate the feasibility of the approach in practical situations.

Finally, simulation is also an alternative. In a simulation environment human configuration actors are replaced by software agents that make random decisions. Simulation can be very useful in scenarios in which it is difficult or even impractical to run a real case study or when the aspects to be assessed in the approach do not necessarily depend on mirroring a real world situation. For instance, we plan to devise an algorithm to generate random feature models with tens of thousands of features and run simulations to evaluate the boundaries of the approach's components such as its algorithms, data structures, and so forth.

4 Research to Date

4.1 Publications

Prior to the writing of this proposal we had the opportunity to discuss our research in international academic events and gather valuable feedback to improve our ideas and broaden our perspective on related works.

SPLC 2006

Mendonca M., Oliveira T., Cowan D.D., Collaborative and Coordinated Product Configuration, International Software Product Line Conference, SPLC 2006, Doctoral Symposium, August 2006, Baltimore, Maryland, USA.

At SPLC 2006 we discussed our research with a committee of experts in the field of Software Product Lines. An article [64] previously submitted and accepted guided the discussion. The committee suggested the use of large feature models to test our approach and its scalability and the development of a simulation environment to reproduce real product configuration scenarios which would allow us to validate the outcomes produced.

OOPSLA 2006

Mendonca, M., Czarnecki, K., Oliveira, T., Cowan, D.D.: Towards a Framework for Collaborative and Coordinated Product Configuration, Companion to the 21th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, Doctoral Symposium, October 2006, Portland, Oregon, USA.

In another Doctoral Symposium event, this time at OOPSLA 2006, we discussed our research through a paper entitled *Towards a Framework for Collaborative and Coordinated Product Configuration* [65]. Suggestions included the use of the available CSP infrastructure (with possible extensions) to provide support for human collaboration especially with regards to conflict resolution support, and the study of CSCW concepts and how we could potentially reuse them in our approach especially in terms of tool support.

HICCS 2007

Mendonca, M., Oliveira, T., Cowan, D.D.: A Process-Centric Approach for Coordinating Product Configuration Decisions, 40th Hawaii International Conference on Systems Science, HICSS-40 2007, Software Technology track, IEEE Computer Society, January 2007, Waikoloa, Hawaii, USA.

At HICCS 2007 we presented a paper entitled *A Process-Centric Approach for Coordinating Product Configuration Decisions* [66] in the Software Engineering track. In the paper we discussed how to describe CPC scenarios using a process notation, namely BPMN [58]. Moreover, we developed some techniques to support conflicts resolution based on priority schemes. During the discussions it was suggested the development of a specific interface for conflict resolution, what we called *conflict resolution support interface (CSRI)*, to assist playing configuration roles in manually resolving decision conflicts. Configuration roles could for example try different alternatives and ask the CSRI to check if conflicts were resolved. Additionally, the CSRI could also make suggestions to configuration roles to resolve conflicts.

4.2 Experimentation

Collaborative configuration model

We have defined a first version of our collaborative configuration meta-model as shown is Figure 10 and Figure 11. The three core abstractions to enable collaboration were specified: configuration spaces, configuration roles, and configuration actors. The dynamics of collaborative configuration are supported by concepts such as configuration scenarios, steps, and sessions. We expect to refine our meta-model as we progress in our research. The architecture of the CPC runtime system is under construction. We are concentrating primarily on the major components such as the runtime environment and the conflict resolution interface.

Tooling and algorithms

We have a strong commitment to make our research practical. Thus, we plan to develop tool support for specifying and executing collaborative configuration scenarios as discussed in this proposal. We have already investigated some existing class libraries and tools that could be potentially extended for our purposes. SAT4J [53] is a SAT solver library written in Java and developed by the *Lens Computer Science Research Centre* in France. The library contains several efficient implementations of SAT algorithms which attempt to find a solution for CSP problem described. The library can be used either as a "black-box" by first users or tailored to particular research needs. After prototyping with SAT4J we came to the conclusion that its use in our research would be too limited as satisfiability is just a small piece in our approach.

We are now prototyping with a popular open-source CSP Java tool called Choco [52]. CSP problems in Choco are represented by the *Problem* class that stores a set of variables, variable domains and constraints. Our major interest in a CSP tool like Choco is two-fold: i) identify the adherence of the library to our objectives of supporting collaborative configuration, and ii) understand its extension mechanisms so that we can adapt the tool to our needs. In fact, in order to support collaborative configuration properly we need to develop a configuration distributed system where each node is able to perform some local independent tasks and have the result of such tasks unified and committed to the central repository. So far, our experience with Choco has been positive in the sense that we were able to partially implement and validate some of our ideas. However, further research is needed to understand the limitations of the tool and its adherence to our goals in collaborative configuration. In particular, we have had some hard time to find a straightforward way to support the *awareness system* component we planned for our approach. Apparently Choco does not provide adequate support for reasoning on distributed problem-solving. Additionally, problems (the combination of variables, variable domains, and constraints) and solutions (variable assignments) in Choco can not be cloned, joined, or merged, essential operations to support negotiation and conflict resolution in our approach. Extensions to the library are needed for this purpose.

In the following, we illustrate how configuration space *{N}* of the Web-Portal product line can be represented as a Choco *Problem* object. Line 2 creates the *Problem* object. Variables (represented by the features in our feature model) are created in lines 3-7 and associated with the problem. Line 9 shows a group constraint among features *https, nntp,* and *ftp* requiring that at least one of these features be selected during product configuration. Line 10 enforces the *requires* relation between parent (*protocol)* and children (*https, nntp*, and *ftp)* features in configuration space {N}.

Configuration Space {*N*} modeled as a *Problem* in Choco

```
01. /**** {N} Configuration Space ****/
02. nConfSpace = new Problem();

03. /** {N} Decisions **/
04. IntDomainVar protocol = nConfSpace.makeEnumIntVar("Protocol", 0,1);
05. IntDomainVar https    = nConfSpace.makeEnumIntVar("HTTPS", 0,1);
06. IntDomainVar nntp     = nConfSpace.makeEnumIntVar("NNTP", 0,1);
07. IntDomainVar ftp      = nConfSpace.makeEnumIntVar("FTP", 0,1);

08. /** {N} Constraints **/
09. nConfSpace.post(
      nConfSpace.atleast(
                  new Constraint[] {
                        nConfSpace.eq(1,https),
                        nConfSpace.eq(1,nntp),
                        nConfSpace.eq(1,ftp)}, 1 ));
10. nConfSpace.post(
      nConfSpace.relationTupleAC(
            new IntVar []
               {protocol, https, nntp, ftp}, new
            ParentChildrenConstraint()));
```

On the other hand we have been able to implement in Choco the MERGE-DECISIONS algorithm proposed in section 3.3.1. The code is shown below. Please refer to section 3.3.1 for details on the algorithm implementation.

```
private int [] mergeDecisionsImpl( Problem p, Constraint [] c, int [] s, int [] a ) {

        int [] r = {};
        int maxSimilarityA = 0;
        int maxSimilarityS = 0;

        // Add the constraints in "c" to problem "p"
        for ( int i = 0 ; i < c.length ; i++ ) {
                p.post(c[i]);
        }

        // if "s" satisfies "p" returns "s"
        if ( satisfies(p,s)) {
                return s;
        }
        // search a solution for "p" that best approximates the assignments in "a"
        else {
                // for all solutions in "p"
                p.solve();
                do {
                        int SimilarityA = compareAssignments(p,a);

                        if ( r.length == 0 ) {
                                maxSimilarityA = SimilarityA;
                                r = solutionToIntArray(p);
                        }
                        // temporarily saves the solution that best approximates A
                        else {
                                // ensures the solution also minizes changes in 's'
                                if ( SimilarityA == maxSimilarityA ) {
                                        int SimilarityS = compareAssignments(p,s);
                                        if ( SimilarityS > maxSimilarityS ) {
                                                r = solutionToIntArray(p);
                                                maxSimilarityS = SimilarityS;
                                        }
                                }
                                else if ( SimilarityA > maxSimilarityA ) {
                                        r = solutionToIntArray(p);
                                        maxSimilarityS = compareAssignments(p,s);
                                        maxSimilarityA = SimilarityA;
                                }
                        }
                } while (p.nextSolution().booleanValue());
        }
        return r;
    }
```

We applied the MERGE-DECISIONS algorithm in two different merge scenarios for the configuration spaces of the Web-Portal product line. The first merge was a *priority merge* where configuration space *{P}* decisions would prevail over *{S}, {N}*, and *{F}*. Parameters *p, c, s,* and *a* of the algorithm are indicated in each scenario below. The output shows the algorithm attempting to find a solution for the merge that primarily minimizing changes in *{P}* and subsequently in the remaining configuration spaces. For instance, each line described in the *Output* session correspond to a solution found and calculations to check if the solution is appropriate for the purposed of the merge operations. Line 1 "`A:(0,0:2) S:(2,0:12)`", for example, indicates that the first solution found had no overlapping with *A* (the set of decisions made in *{P}*) and 2 out of 12 overlapping with S. Thus, the solution does not seem to be interesting as the merge aims at minimizing changes in *{P}*. The 9th solution in line 9 fully overlaps with *A* and shares 5 decisions with *S*. The best solution is coincidently found in the last solution (line 11). The *statistics* session shows that changes to *A* were fully minimized (see `Changes to Restriction`) and 41.67% of *S*'s assignments were changed. In other words, the *database manager*'s decisions on *{P}* were all preserved and prevailed over the *security specialist* decisions that had to be eventually changed.

First Run: Priority Merge
Goal: decisions in {P}.dm will prevail over decisions in {S}.ss U {N}.ss U {F}.ss
p = {P}.dm U {S}.ss U {N}.ss U {F}.ss
c = ("storage *requires* database")
s = 1,0,1,1,0,1,0,1,1,1,0,0 (xm,~da,au,st,~tr,pr,~ht,nn,ft,ms,~se,~mi)
a = 1,0 (xm,~da) - *database manager* role's decisions on configuration space *{P}*

Output

```
 1.  A:(0,0:2) S:(2,0:12)
 2.  A:(0,0:2) S:(3,2:12)
 3.  A:(0,0:2) S:(4,3:12)
 4.  A:(0,0:2) S:(6,4:12)
 5.  A:(0,0:2) S:(8,6:12)
 6.  A:(0,0:2) S:(9,8:12)
 7.  A:(0,0:2) S:(10,9:12)
 8.  A:(2,0:2) S:(4:12)
 9.  A:(2,2:2) S:(5,4:12)
 10. A:(2,2:2) S:(6,5:12)
 11. A:(2,2:2) S:(7,6:12)

****************************
STATISTICS
Restriction: 1,0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1
Original...: 1,0,1,1,0,1,0,1,1,1,0,0
```

```
Merged.....: 1,0,1,0,1,1,1,1,1,0,0,1,1,1,1,1,1,1
Changes to Restriction: 0/2 (0.00%)
Changes to Original...: 5/12 (41.67%)
*****************************
```

In the second run of our algorithm (see below) we performed a *minimize-overall-changes* merge in which no priority was assigned to any of the configuration spaces. Instead, the aim is to minimize the overall number of changes in all configuration spaces. The algorithm used is the same as in the previous case but the parameters changes. Essentially, the difference is achieved by indicating that *A* = *S*, i.e., minimize changes in all configuration spaces.

<u>Second Run</u>: Minimize Overall Changes Merge
Goal: Minimize changes on {P}.dm, {S}.ss U {N}.ss U {F}.ss (no priorities)
p = {P}.dm U {S}.ss U {N}.ss U {F}.ss
c = ("storage *requires* database")
s = 1,0,1,1,0,1,0,1,1,1,0,0 (xm,~da,au,st,~tr,pr,~ht,nn,ft,ms,~se,~mi)
a = s

<u>Output</u>

```
  1. A:(4,4:12) S:(4,0:12)
  2. A:(5,4:12) S:(5:12)
  3. A:(6,5:12) S:(6:12)
  4. A:(7,6:12) S:(7:12)

*****************************
STATISTICS
Restriction: 1,0,1,1,0,1,0,1,1,1,0,0
Original...: 1,0,1,1,0,1,0,1,1,1,0,0
Merged.....: 1,0,1,0,1,1,1,1,1,0,0,1,1,1,1,1,1,1
Changes to Restriction: 5/12 (41.67%)
Changes to Original...: 5/12 (41.67%)
*****************************
```

5 Next Research Steps

Strategies to Minimize Conflicts

One of the major goals in our approach is to promote a smooth and consistent collaborative configuration process while maximizing parallel work and minimizing decision conflicts. Currently, conflict minimization is fostered by awareness mechanisms and algorithms for automatic and manual merge of configuration decisions. However, we believe that other mechanisms can be put in place to help reducing conflicts. The first idea is to allow configuration roles to indicate the importance of the decisions they are dealing with. Currently, we have provided a categorization scheme for decisions based on their impact throughout the configuration spaces. However, the algorithms we proposed especially merge operations consider all decisions of equal importance. By allowing a decision role to indicate that she "doesn't care for" a decision could have a great impact on minimizing conflicts. In addition, allowing for constraint relaxing (e.g., weak versus strong constraints) or even their elimination would reduce the likelihood of decision conflicts. We plan to use both mechanisms in the future and study their impact on the scalability of our algorithms.

Conflict Resolution Support Interface

The Conflict Resolution Support Interface aims at assisting configuration actors in manually resolving decision conflicts. This interface supports the manual merge of configuration decisions where configuration actors are allowed to relax some of their decisions and check for the overall consistency of the partial product specification achieved. The intention is to use off-the-shelf CSP components such as those provided by Choco combined with our own algorithms for merging decisions. The user of the interface can be either the configuration actors involved in the conflict or a specific actor assigned to resolve the conflict.

Case studies

We plan to validate the applicability of the approach through case studies. There are interesting feature models in the literature that can be used in this case including an e-Shop feature model that contains hundreds of features and an augmented version of the Web Portal product line discussed in previous sections of this document. Building large feature models for an existing family of product is also a possibility.

Simulation and scalability tests

To test the approach's scalability we plan to develop a simulation environment in which large randomly-generated feature models are configured automatically by software agents, by a single human role, or both following a particular CPC scenario description. Simulation will allow us to know the boundaries of the approach and search for better means to improve its performance.

3C model instantiation for collaborative configuration group work

A possible alternative to build a more explicit connection between our work and CSCW would be to instantiate the 3C model illustrated in Figure 3 in the context of *collaborative configuration group work*.

References

[1] Software Engineering Institute (SEI). Software Product Lines website. Link: http://www.sei.cmu.edu/productlines/index.html

[2] P. Clements, L. Northrop. Software Product Lines: Practices and Patterns, Addison-Wesley, Longman Publishing Co., Inc., Boston, MA, 2001. ISBN: 0201703327.

[3] C. Krueger. Software Product Lines website. Link: http://www.softwareproductlines.com/

[4] C. Krueger. Software Product Lines web site. Link: http://www.softwareproductlines.com/introduction/ production2.html

[5] Software Engineering Institute (SEI). Software Product Lines web-site. Production Planning. Link: http://www.sei. cmu.edu/productlines/production_plan.html

[6] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Nov. 1990.

[7] J. van Gurp, J. Bosch. Managing Variability in Software Product Lines. Landelijk Architectuur Congres, Amsterdam 2000.

[8] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In Proceedings of the Third Software Product Line Conference 2004, pages 266–282. Springer, LNCS 3154, 2004.

[9] A. Goldberg. Collaborative Software Engineering. In: Journal of Object Technologies, 1 (2002) 1, S. 1-19.

[10] R. Krikhaar, and I. Crnkovic. Software Configuration Management. Sci. Comput. Program. 65, 3 (Apr. 2007), 215-221. DOI= http://dx.doi.org/10.1016/j.scico.2006.10.003

[11] W. F. Tichy. Tools for Software Configuration Management. In Proc. Of the Int. Workshop on Software Version and Configuration Control, pages 1–20, Grassau, January 1988.

[12] N. Graham, H. Stewart, A. Ryman, R. Kopaee, and R. Rasouli. A World-Wide-Web Architecture for Collaborative Software Design. In Software Technology and Engineering Practice, pages 22–32. Pittsburgh, Pennsylvania, August 1999.

[13] Carl Cook. Collaborative Software Engineering: An Annotated Bibliography. Technical Report TR-COSC 02/04, Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand, June 2004. Work in Progress.

[14] C. Krueger. BigLever GEARS tool, BigLever Software Inc., link: http://www.biglever.com/extras/ Gears_data_sheet.pdf

[15] Pure-systems GmbH. Variant Management with Pure::Consul. Technical White Paper. Link: http://web.pure-systems.com, 2003.

[16] M. Antkiewicz and K. Czarnecki, K. FeaturePlugin: Feature modeling plug-in for Eclipse. In: OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop. (2004) Link: http://www.swen.uwaterloo.ca/kczarnec/ etx04.pdf. Software available from gp.uwaterloo.ca/fmp.

[17] S. Lau. Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates. Master of Applied Science Thesis, Electrical and computer Engineering, University of Waterloo. Link: http://gp.uwaterloo.ca/files/2006-lau-masc-thesis.pdf

[18] S. Deelstra, M. Sinnema, and J. Bosch: Product Derivation in Software Product Families – A case study. Journal of Systems and Software, Volume 74, Issue 2, 15 January 2005, Pages 173-194.

[19] E.P.K. Tsang. Foundations of Constraint Satisfaction. Academic Press, London and San Diego, 1993 ISBN 0-12-701610-4.

[20] V. Kumar. Algorithms for Constraint Satisfaction Problems: a Survey. AI Msg. 13 (1) (1992) 32-44.

[21] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker. Towards Distributed Configuration. Proc. KI-2001, Joint German/Austrian Conference on AI, Vienna, Austria, Lecture Notes in AI, Springer Verlag.

[22] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. IEEE Transactions on Knowledge and Data Engineering, v.10 n.5, p.673-685, September 1998.

[23] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration through Specialization and Multi-level Configuration of Feature Models. Software Process Improvement and Practice, 10(2), 2005.

[24] K. Kang, S. Kim, L. Lee, K. Kim, E. Shin and M. Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering, 5, 1998, pp. 143-168.

[25] K. Kang, K. Lee, and J. Lee. FOPLE - Feature Oriented Product Line Software Engineering: Principles and Guidelines. Pohang University of Science and Technology, 2002

[26] M. Riebisch et al. ALEXANDRIA: Software Product Line Development Methodology.
Link: http://www.theoinf.tu-ilmenau.de/~pld/

[27] K. Czarnecki and U.W. Eisenecker. Generative Programming. Addison Wesley, 2000. ISBN: 0201309777.

[28] V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger. XML-based Feature Modelling. LNCS, Software Reuse: Methods, Techniques and Tools: 8th ICSR 2004. Proceedings, 3107:101–114, 2004.

[29] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In R. Gl¨uck and M. Lowry, editors, GPCE'05, volume 3676 of LNCS, pages 422–437. Springer, 2005.

[30] T. Bednasch, C. Endler, and M. Lang. CaptainFeature, 2002-2004. Tool available on SourceForge at https://sourceforge.net/projects/captainfeature/.

[31] P. Sochos, M. Riebisch and I. Philippow. The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines. In: 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems.

[32] P. Zave. Feature Interactions and Formal Specifications in Telecommunications. IEEE Computer 26, 8, 20-28.

[33] P. Zave. FAQ Sheet on Feature Interaction. Link: http://www.research.att.com/~pamela/faq.html

[34] M. Griss, J. Favaro, M. d'Alessandro. Integrating Feature Modeling with the RSEB. Proceedings of the Fifth International Conference on Software Reuse (ICSR), IEEE Computer Society Press, Los Alamitos, CA, pp. 76–85.

[35] J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE Computer Society Press, Washington, DC, pp. 45–55.

[36] K. Lee, K. Kang, and J. Lee. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In C. Gacek (ed.), Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR7), Austin, USA, Apr. 15-19, 2002, Vol. 2319 of Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, Germany, pp. 62–77.

[37] K. Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. PhD thesis, Technische Universit¨at Ilmenau, Ilmenau, Germany. Available from http://www.prakinf.tu-ilmenau.de/~czarn/diss.

[38] K. Czarnecki, T. Bednasch, P. Unger, U. Eisenecker. Generative Programming for Embedded Software: An Industrial Experience Report. In D. Batory, C. Consel and W. Taha (eds), Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002, Vol. 2487 of Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, Germany, pp. 156–172.

[39] T. Bednasch. Konzept und Implementierung eines konfigurierbaren Metamodells f¨ur die Merkmalmodellierung, Diplomarbeit, Fachbereich Informatik und Mikrosystemtechnik, Fachhochschule Kaiserslautern, Standort Zweibr¨ucken, Germany. Available from http://www.informatik.fh-kl.de/~eisenecker/studentwork/dt_bednasch.pdf

[40] K. Czarnecki, C. H. P. Kim, and K. T. Kalleberg. Feature Models are Views on Ontologies. In: Software Product Line Conference (2006).

[41] D. Batory. Feature Models, Grammars, and Propositional Formulas. In: Software Product Line Conference (2005).

[42] D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated reasoning on feature models. In Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05), Porto, Portugal, 2005, LNCS. Springer, 2005.

[43] M. Riebisch. Towards a More Precise Definition of Feature Models. In Riebisch, M., Coplien, J. O., and Streitferdt, D., editors, "Modelling Variability for Object-Oriented Product Lines", pp. 64–76, Norderstedt, Germany (2003). BookOnDemand Publ. Co.

[44] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Generic Semantics of Feature Diagrams. Computer Networks 51, 2 (Feb. 2007), 456-479. DOI= http://dx.doi.org/10.1016/j.comnet.2006.08.008

[45] J. Lee, K. Kang, and S. Kim. A Feature-Based Approach to Product Line Production Planning. LNCS, Springer Berlin / Heidelberg, Volume 3154/2004, Pages 183-196. ISBN: 978-3-540-22918-6.

[46] K. Schmidt and L. Bannon. Taking CSCW Seriously. In Computer Supported Cooperative Work Journal, (1992) pp. 7-40.

[47] L. Bannon, and S. Kjeld. CSCW: Four characters in search of a context. ECSCW '89. Proceedings of the First European Conference on Computer Supported Cooperative Work, Gatwick, London, 13-15 September, 1989, pp. 358-372. - Reprinted in Studies in Computer Supported Cooperative Work. Theory, Practice and Design, J. M. Bowers and S. D. Benford, Eds. North-Holland, Amsterdam etc., 1991, pp. 3-16.

[48] I. Greif. Computer-Supported Cooperative Work: A Book of Readings. San Mateo, Calif.: Morgan Kaufmann Publishers.

[49] J. Grudin. CSCW: The convergence of two development contexts. In S. P. Robertson, G. M. Olson, and J. S. Olson (eds.): CHI '91. ACM SIGCHI Conference on Human Factors in Computing Systems, New Orleans, 28 April-2 May 1991. New York, N.Y.: ACM Press, pp. 91-97.

[50] C. J. P. Lucena, H. Fuks, A. Raposo, M. A. Gerosa, and M. Pimentel, Communication, Coordination and Cooperation in Computer-Supported Learning: The AulaNet Experience. In Advances in Computer-Supported Learning, F.M.M. Neto and F. Brasileiro (orgs), ISBN 1-59904-356-4, 2006, pp. 274-297.

[51] C. A. Ellis, S. J. Gibbs, and G. L. Rein, (1991). Groupware - Some issues and experiences. Communications of the ACM, 34(1), 38-58.

[52] F. Laburthe and N. Jussien. Choco - Constraint Programming System. Link: http://choco.sourceforge.net/, 2003-2007.

[53] SAT4J - A Satisfiability Library for Java. Available at http://www.sat4j.org

[54] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using Java CSP Solvers in the Automated Analyses of Feature Models. In Post-Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE). LNCS 4143, 2006.

[55] A. Goldberg. Collaborative Software Engineering. Journal of Object Technology, Vol. 1, No. 1, May-June 2002. Published by ETH Zurich, Chair of Software Engineering. Available at: http://www.jot.fm/jot/issues/issue_2002_05/column1.pdf

[56] M. G. Pimentel. RUP-3C-Groupware: um processo de desenvolvimento de groupware baseado no Modelo 3C de Colaboração. Tese de Doutorado, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), 22 de março de 2006 (*in Portuguese*).

[57] J. C. P. Lucena. Applying the 3C Model to Groupware Development. International Journal of Cooperative Information Systems (IJCIS), v.14, n.2-3, Jun-Sep 2005, World Scientific, ISSN 0218-8430, pp. 299-328.

[58] Object Management Group (OMG). BPMN: Business Process Modeling Notation. Link: http://www.bpmn.org/index.htm

[59] K. Schmidt and T. Rodden. Putting it all Together: Requirements for a CSCW Platform. In Shapiro, D., Tauber, M., Traunmüller, R. (eds.): The Design of Computer Supported Cooperative Work and Groupware Systems. North Holland, Holland, pp. 157-176.

[60] D. C. Neale, J. M. Carroll, and M. B. Rosson. Evaluating Computer-Supported Cooperative Work: Models and Frameworks. In Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work (Chicago, Illinois, USA, November 06 - 10, 2004). CSCW '04. ACM Press, New York, NY, 112-121. DOI= http://doi.acm.org/10.1145/1031607.1031626

[61] C. Gutwin, R. Penner, and K. Schneider. Group Wwareness in Distributed Software Development. In Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work (Chicago, Illinois, USA, November 06 - 10, 2004). CSCW '04. ACM Press, New York, NY, 72-81. DOI= http://doi.acm.org/10.1145/1031607.1031621

[62] B. Singh. Invited Talk on Coordination Systems at the Organizational Computing Conference (November 13-14, 1989, Austin, Texas).

[63] P. Dourish and V. Bellotti. Awareness and Coordination in Shared Workspaces, Proc. ACM CSCW 1992, 107-114.

[64] M. Mendonca, T. Oliveira, D.D. Cowan, Collaborative and Coordinated Product Configuration, International Software Product Line Conference, SPLC 2006, Doctoral Symposium, August 2006, Baltimore, Maryland, USA.

[65] M. Mendonca, K. Czarnecki, T. Oliveira, D.D. Cowan: Towards a Framework for Collaborative and Coordinated Product Configuration, Companion to the 21th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, Doctoral Symposium, October 2006, Portland, Oregon, USA.

[66] M. Mendonca, T. Oliveira, D.D. Cowan: A Process-Centric Approach for Coordinating Product Configuration Decisions, 40th Hawaii International Conference on Systems Science, HICSS-40 2007, Software Technology track, IEEE Computer Society, January 2007, Waikoloa, Hawaii, USA.

Appendix A:
XML representation of a collaborative configuration scenario

```
<configuration_scenario configuration_roles_spaces_file=”Web-Portal_roles_spaces.xml”>
<configuration_step name="step1">
  <pre_conditions/>
  <configuration_sessions>
     <session name="pm_session" configuration_role="project manager">
        <communication_scheme/>
        <configuration_space name="{W}">
        </configuration_space>
     </session>
  </configuration_session>
   <post_conditions/>
</configuration_step>
<configuration_step name="step 2">
  <pre_conditions/>
  <configuration_sessions>
     <session name="ss_session" configuration_role="security specialist" awareness="on">
       <communication_scheme/>
       <configuration_space name="{G}">
       </configuration_space>
       <configuration_space name="{S}">
       </configuration_space>
       <configuration_space name="{N}">
       </configuration_space>
     </session>
     <session name="wd_session" configuration_role="web designer" awareness="off">
        <communication_scheme/>
        <configuration_space name="{G}">
        </configuration_space>
     </session>
     <session name="dm_session" configuration_role="database manager" awareness="on">
        <communication_scheme/>
        <configuration_space name="{P}">
        </configuration_space>
     </session>
   </configuration_session>
   <post_conditions>
       <merge type="minimize_changes" >
           <sessions>ss_session, dm_session</sessions>
       </merge_type>
      <merge type="priority" >
           <sessions>ss_session, wd_session</sessions>
       </merge_type>
       <merge type="union" >
           <sessions>ss_session, dm_session, wd_session</sessions>
        </merge_type>
   </post_conditions>
</configuration_step>
</configuration_plan>
```