Issues and Challenges with Industrial-Strength Product Composition

Lars Bendix, Jacob Gradén, Anna Ståhl
Department of Computer Science
Lund University, Sweden
email: bendix@cs.lth.se; graden@gmail.com; aannastahl@gmail.com

Andreas Göransson
Sony Ericsson Mobile Communications AB
Nya Vattentornet, S-221 88 Lund, Sweden
email: andreas.goeransson@sonyericsson.com

Abstract (English)

One way to deal with product lines and products that have to exist in many variants is to use a component-based approach. This allows a lot of flexibility in creating new products and can reduce time and costs. However, the flexibility in composing products from a base of components is not without problems. Complexity increases in the composition process when combining many components to one large system – in particular because components may also exist in several revisions or variants. Furthermore, there are many different stakeholders involved in the development of new components and products and they all work at different levels of granularity and have different needs.

From the analysis of a complex industrial context, we have identified a set of issues and challenges that need to be addressed for advanced product composition. A shared component base can work as a repository for storing facts and information about components and a rule-base will allow users to reason about configurations at a higher level such as completeness and consistency.

***Keywords**: High-level composition, rule-based configuration, variant management, configuration management, common terminology.*

Проблемы и спорные вопросы сборки продуктов промышленного уровня

Lars Bendix, Jacob Gradén, Anna Ståhl
Department of Computer Science
Lund University, Sweden
email: bendix@cs.lth.se; graden@gmail.com; aannastahl@gmail.com

Andreas Göransson
Sony Ericsson Mobile Communications AB
Nya Vattentornet, S-221 88 Lund, Sweden
email: andreas.goeransson@sonyericsson.com

Abstract (Russian)

Одним из инструментов создания продуктовых линеек, а также продуктов, выпускаемых в различных конфигурациях, является компонентно-ориентированный подход. Данный подход, во-первых, допускает немалую степень гибкости процесса создания новых продуктов, а во-вторых, способен снизить временные и финансовые издержки. Однако у подобной гибкости, достигаемой путем сборки продуктов из набора базовых компонентов, существует и обратная сторона. Прежде всего, чрезвычайно сложным оказывается сам процесс построения крупной системы из множества мелких частей – в том числе из-за того, что каждый отдельный компонент может существовать в различных вариациях и исправлениях. Кроме того, в процесс разработки новых компонентов нередко вовлечен целый ряд различных заинтересованных лиц, каждое из которых работает в соответствии со своими потребностями и на своем уровне детализации.

*Анализируя ситуацию в промышленности, мы определили набор проблем и задач, требующих разрешения для эффективной сборки продукта. Фактологическим и информационным хранилищем может служить *разделяемая база компонентов*, а *база* *правил (**rule **base) *позволит пользователям принимать решения по конфигурации на более высоком уровне, оперируя понятиями завершенности и связности.*

Ключевые слова: *высокоуровневая сборка, конфигурирование по базе правил, управление вариантами, управление конфигурациями, общая технология.*

1. Introduction

In today's market many products have a short lifecycle and companies have to come up with "new" products with a high frequency. Such a number of products cannot be made from scratch and a high degree of reuse is needed. One way to obtain this reuse is to develop components and have a base of such components that can be composed in different ways to create different products. Much the same way as it is done when playing with LEGO® bricks.

However, it is no easy task to create products from a component base. It could easily contain tens of millions of lines of code in thousands of components and each component might exist in dozens or even hundreds of revisions and/or variants. Blindly combining components produces an exponential explosion in the potential number of products because of the multiple ways in which components and their versions can be combined. Furthermore, it is difficult to tell whether these combinations are complete, let alone consistent.

In a company there are many different groups of people involved in the production of a new product, ranging from developers over testers to sales and even the customers that in the end buy the product. The present situation is that almost each group has its own representation of a product and each time they have to communicate, this representation has to be "translated" with the risk of loss of information and misunderstandings. Furthermore, there are problems in keeping all those different representations consistent at all times.

The configuration process investigated in this paper does not focus on connecting components at the micro-level of ensuring that type and number of parameters of methods correspond. Rather, we are interested in what issues have to be dealt with at the macro-level in creating a common composition model that can be used by all involved stakeholders.

In this paper, we first describe the specific context of our analysis and identify the different roles and tasks that are involved in the production of products. Next, we analyze the problem domain to expose the issues to address in an industrial-strength composition system. Then we discuss our findings before finally drawing our conclusions.

2. Background

The specific context that we base our work on is the mobile phone industry. This is a field where there is a high pressure for new products and where products often have to be customized. However, we believe that our findings are general and can be used in any context that has to create a high number of different products. Furthermore, we have a background in software configuration management (SCM) and want to explore established SCM techniques for creating workspace configurations from a source code repository [5] in the context of creating products from a component base.

Verification and validation of the composed product is usually a very slow, cumbersome, error-prone and labour intensive process if we have to use the traditional compose-compile-link-load-test process. There is a need for a process that is easier, more flexible, and can provide more immediate feedback. Furthermore, composition is not just a purely technical matter – also business and legal aspects of the products and their components will have to be considered. The user would need to be able to work and reason at a higher level than the components and their detailed source code.

In an organization there are many different groups of people who are involved in creating configurations or using and working with configurations created by others. The present state means that each stakeholder works at a different level of granularity. Some, like the developers, are looking at details like lines of code, while others, like customers, look at the product as one single entity with some desired properties. This creates many challenges when different representations have to be kept synchronized or converted between each other. To add to the confusion different stakeholders often use different terminology.

An analogy to the software composition process is that of meals in a restaurant: There are distinct ingredients which need each other to work properly, and from the same set of available ingredients, many different dishes can be made. There are also relationships and constraints between components (fish should be served with one wine and red meat with another; fish and red meat should not be served together). Dishes and ingredients are also the common terminology shared by the cook, the waiter, the client, and the person buying ingredients.

We would be looking for a composition model that can be used by all stakeholders at the macro-level as shown in Figure 1 below. Components are entered into a component base from which the stakeholder (user interaction) picks a number of components and with the help of a configurator tool, that uses a set of rules, a product (or configuration) is created and worked on. Can such a configurator tool be built and what are the issues and challenges that need to be addressed?

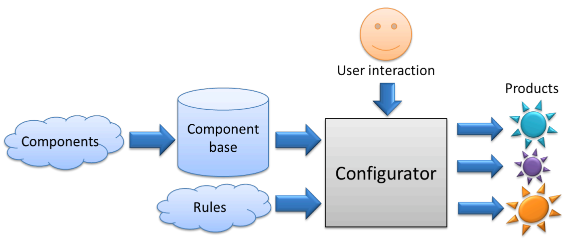

Figure 1: Composition model.

A first step is to identify who are the stakeholders and what are their needs. We interviewed different people in the organization to get an idea of who might – directly or indirectly – be involved in the creation of new products. From that information we established a list of user roles. We then talked to representatives for these user roles to get an idea of what their tasks were, how the composition process looked (or should look) from their perspective, and what they would want a configurator tool to be able to do to help them.

Sales: Sales create new configurations to explore possible future products – and they investigate how "expensive" new products are going to be. For this role we have the following use cases: Configuration creation, Configuration verification, Configuration quotation.

Configuration management: One of the primary tasks of Configuration Management is to provide different kinds of people with proper workspaces where they can carry out their work. For this role we have the following use case (see also *Component manager* below): Create configuration.

Designer: The designer works with the overall design and architecture of the products (and is sometimes called architect). For this role we have the following use cases: Set up configuration, Problem resolution, Create placeholder components, Manage rules.

Developer: Developers work with source code, however, their work is in the context of configurations. For this role we have the following use cases: Set up configuration, Build product (or configuration), Record dependencies.

Tester: A tester gets a configuration and is supposed to test whether it passes the designated test cases or not. For this role we have the following use cases: Set up configuration, Build product (or configuration), Create configuration.

Component manager: This is a new role that materialized during our work. In the current setup most of these tasks are taken care of by Configuration Management, but it might be useful to keep it a separate role though it might be taken care of by the same people as Configuration Management. For this role we have the following use cases: Component administration, Manage rules.

Customer: The customers who buy products from the company are primarily mobile phone operators (like Telia and Vodaphone). In turn they re-sell the products to the real end users. For this role we have the following use cases: Create configuration.

We also identified a number of use cases that it was not possible to place completely on any single user role: Notification, Statistics, Quality aid, and Requirements fulfilment.

3. Analysis

In this section, we will use the roles and use cases from section 2 to analyse what problems need to be solved and what kind of functionality a composition system (a configurator tool or a set of manual processes) should provide.

There are many technical aspects to consider regarding product composition, but since the system is intended for several disparate groups of users, usability concerns are also very important. The composition system must be simple enough to be used by people with little or no technical understanding of composition, but at the same time powerful enough to allow savvy users complete freedom. Different modes of operation could be a possible solution to this, depending on the role the user has.

It is also important to note that due to the complexity of the problem, the system should be able to give quick and exact feedback. In other words, as soon as a component is selected or deselected, the entire configuration should be re-evaluated. If new components are made available or unavailable as a result of an action, this should also be updated at once. It would be detrimental to user understanding if this were not the case.

The technical aspects of a composition system are detailed below. They fall under the categories of Component base, Configurations, and Rules.

3.1. Component base

The component base is the complete set of all components and all their versions. This also includes the properties of each component version, the relationships between components, and even the created configurations, since configurations can also be used as components.

The question of exactly which form the component base and its storage should take we leave open, but there are several very important high-level aspects of the component base, which must be addressed regardless of the actual implementation. Specifically, the capabilities, evolution, clean-up and how to handle consistency when new components are added (commit checks) require attention.

Capabilities. Regardless of the form the component base takes, there is one basic capability it must have, namely support for proper versioning of items. There are also a great many capabilities, which users may wish that it had, such as allowing for variants and concurrent work [3]. The distinction between capabilities of the component base itself, and of tools working with the component base – such as a configurator – is somewhat blurry, since tools can often simulate many aspects even if the component base does not support them directly.

Without versioning capability, however, development on many different components over time cannot be accommodated, and the component-based aspect collapses. Depending on how complex configurations become and how many people work on them, there might even be an interest in providing features like branches of components (variations on the same basic component, existing simultaneously).

In addition to versioning, the component base could provide support for answering questions such as "Who added component A", "What was changed between revisions X and Y", and "What does the entire history of changes look like for configuration Z". These capabilities are not strictly required, because the same questions can be answered by performing manual work, but would be valuable to improve efficiency.

One important thing to note about the component base, which is very relevant for a configurator, is whether components are stored as source code or as binaries. From a composition perspective, it is much easier to manage binaries. Source code must be compiled, which means that every such component requires the compiler, and that the compiler version, all compiler flags, and so on, must be stored for future reference. This is not necessary for binaries, and the focus can therefore be exclusively on composition.

It should be noted that irrespective of whether source code or binary components are added to the component base, this is not a place where active source code development will take place. Rather, all day-to-day development will take place in a dedicated source code repository, and once a component reaches a certain stage it will be added to the component base from the source code repository.

Evolution. When a component base is created it is normally small and it is possible – even easy – to get a complete overview. Over time, however, as new components are added, and revisions and variants complicate the picture, the component base becomes larger and more complex. The overview is lost and achieving a full understanding of even a single component may require considerable effort. Mechanisms must be in place from the very beginning to help counter this problem and keep the component base usable.

A typical example of this is a method to trace all configurations where a given component is used, to quickly see if it is used at all, and whether or not those configurations are still active. Sorting and filtering capabilities are also a bonus, to facilitate administrative tasks. Whether this is technically carried out by the component base itself or a configurator tool is less interesting.

Clean-up. When more and more components and versions thereof are added, together with relationships, large webs of component interdependencies risk being created. This limits the configurations, which can be created, and endangers the basic premise of component-based systems: that components are mainly independent.

A simple solution to this is to regularly remove old components, which should no longer be available to ordinary users. Since it is still desirable to be able to recreate old configurations, components should not actually be deleted from the component base, but instead marked as not being available when new configurations are created. This will keep the amount of components available for daily use limited, thereby helping the user to retain an overview.

A more demanding clean-up is also possible, in the form of architectural decisions. It should be possible to identify newly developed or updated components which rely on old components, or components which have relationships with many others, and consider whether their relationship webs can be pruned. While this costs in terms of development effort, the benefit is a smaller and healthier system, which can be combined more freely. At some point, entire component webs may even be marked as deprecated, and replaced with newer solutions. Crnkovic and Larsson [1] present some interesting thoughts about this, namely that development on a component is most active in the beginning and end of its useful lifespan, eventually reaching a point where it is better to start on a new design than continue working on the old.

Commit checks. Whereas clean-up is performed on the contents already in a component base, commit checks are run before any changes are allowed to be made to the component base. The basic idea is to apply the same checks that are run when a configuration is created, as well as checks performed during clean-up, and make sure everything looks good. Commit checks allow for catching some structural problems, such as incorrect dependencies. In its most basic form, component A requires component B, but at the same time, A and B are in conflict with each other. Component B is usable, but component A can never be part of any complete and consistent configuration. In practice, the problems caught by using commit checks are both subtler and more complex.

3.2. Configurations

In component-based systems, configurations are top-level objects – they are products. A configuration is in essence a collection of components, and by varying the components, different product variants can be created. However, since components have relationships, versions and properties, collecting them poses challenges. Configurations themselves can also exist in different versions – namely when the collection is changed as components are added or removed – and have properties of their own, in addition to properties that come with the components.

Configurations do not need to consist of many components, although they normally do. An empty configuration (containing zero components) is normally useless, but a configuration with only one component may be useful: This is a way of transforming just that component into a product. The same goes for small configurations, containing only a few components (partial configurations), and for configurations, which represent entire systems with all the necessary components (full configurations).

It is worth noting that component suites are technically equivalent to configurations. The only difference is that suites are used internally, when configurations are created, whereas configurations (products) are used externally, when configurations have been created and are being distributed as products. There is no technical difference between them, and they could be handled in the same way.

Creating and working with configurations is not trivial. There are issues of completeness, consistency and identification to consider, and also the optimality of component selection.

Completeness. A configuration is complete if it contains all needed components. There are several reasons a component may be needed, corresponding to different kinds of completeness. Relationship completeness is a technical problem and the most fundamental completeness. There is also rules completeness, which can be broken down in several sub-categories, depending on the types of rules.

Relationship completeness is the condition that all components are present which are required by other components. If component A is selected, and A requires B and C, then a configuration containing only A would not be relationship complete; nor would a configuration containing A and B, or A and C. Only if all three are included will relationship completeness be achieved.

Rules completeness is the condition that all rules are obeyed which call for the inclusion of components. The situation is basically the same as for relationship completeness, but the mechanism is different. The reasons for rules to require components may vary widely – layer requirements may be one reason, business logic or legal issues could be other reasons.

Layer completeness is the requirement that components from certain layers must be present – typically, the operating system is required to be able to use components from any other layer. Required layers differ between configurations: partial configuration may have no required layers at all, whereas full configurations do. Certain layers simply must be present for the system to work. The reason is that if there is no operating system, all the applications of the world can be part of the configuration and it still will not be usable.

Consistency. Consistency is similar to completeness in that it is a sort of sanity check for configurations. If a configuration is inconsistent, some components will be unable to function – applications may for example not start. As for completeness, there is relationship consistency and rules consistency.

Relationship consistency is based solely on the relationships between the components in a configuration, and a configuration is said to be consistent if all anti-dependencies of all components are respected – that is, no two components in the configuration are in conflict with each other. Depending on the specific configuration, there may also be other relationships, which must be considered, for example if components break or replace each other.

Rules consistency is based on rules between components and their properties. A configuration can be relationship consistent and still make no sense from a business perspective – for example if two components are included which technically work together, but which are targeted for two different customers.

There are additional consistency problems too, such as how to handle shared libraries. If component A requires version 1 of library L, and B requires version 2 of L, the configurator must either include both versions or inform the user that different versions of the same library are disallowed, depending on the rules in effect. A similar situation is if two selected component suites reference the same fundamental component, perhaps in two different versions – that component should not be included twice just because it is referenced by two suites, but it may have to be included in both its versions to satisfy the requirements of both suites.

Optimality. In several cases, more than one version of a component may be available, for example if component A requires B version 2.0 or greater. B may exist in many revisions greater than 2.0, but only one of them must be chosen. A common tiebreaker is to pick the latest component, but there are other issues to consider first. Do all versions of B have the same relationships, and are they all satisfied? Have all versions of B been tested and accepted, separately and together with all other components in the configuration? Are there more open bugs in any of them than in others? Are they all roughly the same size or are some larger than others, thus costing more in time and transfer fees than others? All else being equal, choosing the latest version is still a good tiebreaker, but "all else" must first be verified as being equal.

An important guideline is to select specific versions as late as possible. The rationale is that when a specific version is selected, the set of possible component combinations is limited, and that set should be kept as large as possible to ensure flexibility for the user. Figure 2 shows a typical example of late binding. Component A requires B, versions 2 or 3. Component C also requires B, versions 1 or 2. If A is selected and B v. 3 is automatically added, then C is unavailable for selection. Using late binding, selecting A would instead mean also selecting B, "any version". Only when C is also selected would the version of B become unambiguously known, since only B v. 2 works with both A and C.

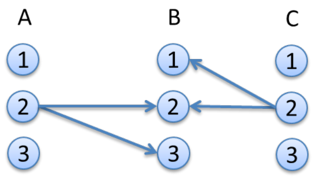

Figure 2: Late binding.

In some cases, there may be not only several versions of a component, but more than one component as well. For example, component A may depend on a feature and there may be many components available which supply that feature. In such a case, not only is the set of candidates greater (many components in many versions, instead of one component in many versions), but different components may differ more from each other than do different versions of the same component.

3.3. Rules

Rules are a way of explicitly stating what configurations are allowed to look like, in terms of components, component versions, component properties, configuration properties, and so on. They form a general framework to enforce completeness and consistency on configurations – not just from a technical point of view, but from every conceivable other outlook as well.

There can be many different types of rules, corresponding to these outlooks. The archetypal example is business rules, which state that regardless of the technical possibilities, certain component selections are mandatory or prohibited for reasons of strategy, competitiveness, or any other business-oriented concern. The types of rules can differ depending on the situation – sometimes, business rules is too vague a concept and needs to be split into marketing rules, strategy rules, and so on.

Functionally, rules are a mechanism – a form of logic, or expressions. Rules are used to handle facts – components, properties, relationships, and many more. For instance, if component A requires B, that is a fact. A rule could then be "All components which are required must be present for a configuration to be complete".

This allows a separation of concerns, and hides a lot of complexity inside the rules, which means that ordinary users of the configurator have no need to understand those intricacies. The complexity is contained in an isolated area, but it is not gone – somebody has to maintain the rules.

The task of rules maintenance would fall on different user roles, depending on the type of rules. Rules affecting the overall structure would likely require input from architects and designers. Marketing rules would require input from the Sales department, even if some other user role might be more likely to actually formalize the rules and add them to the system. Rules that influence the selection of specific component versions (see Late binding in section 3.2) would require input from configuration managers.

Rules can perform many different functions. Placing constraints on the technical structure of a valid configuration, as in the example above, is one. Another function could be to ensure marketing strategy – for example by requiring that a certain component is always included in products created during 2010. Regulating which user roles are allowed to perform which duties could be yet another function.

The practical design and actual implementation of rules and the system to actually apply the rules, is an open area, which requires future work. Several questions need to be investigated, such as:

- Should all rules be general for the entire configurator and all items in the component base?
- Should there be implicit rules in the configurator, which must always be adhered to – for instance rules regarding how completeness is calculated?
- How should different versions of rules be handled?
    - What should happen when an attempt is made to introduce a new rule, which would

invalidate existing configurations, or even components?

- What should the result be if different rules are in conflict – for instance a marketing rule and a legal rule?
    - Is there a need for prioritization of rules?
    - Should all (active) rules always be followed, except possibly for certain user roles?

Rules are an extraordinarily powerful way of simplifying the process of creating configurations and ensuring verification and validation for very different user roles.

4. Discussion

In this section, we will discuss some related and future work. The perspective we have in this work is an industrial "this is what we need from a composition framework", whereas the mostly related work we have been able to find has an academic perspective of "this is what is possible in component composition". The main difference is that even though it is also important for industry that micro-level details of composition are dealt with, it is at least equally important that it is possible to use a composition framework at a high level and that the representation is shareable between very diverse groups of users.

Very early work by DeRemer and Kron [4] focused on providing components with a "module interconnection language". That was in the days before modern programming languages that since have integrated the work of DeRemer and Kron. Now that components again tend to be "stand alone" new research has been done in "connecting components". The work of Lau et al. [6] also emphasizes that a component model should cover more phases. However, they use different tools in the different phases and do not obtain the same tight integration that we get from our common configurator. In Oberleitner et al. [7], they also add metadata to components with the purpose of validating compositions. However, they "bake" their metadata into the binaries of the components. We cannot allow such a thing since it changes the binary when metadata is changed and because we under strict space restrictions for memory. Scheben [8] works with a model that allows dynamic selection and late binding, which works in a way that is very similar to our "feature dependencies" described in section 3.1. However, whereas she uses it to connect components in a flexible way at runtime, we are more interested in using it to reason about flexible compositions at a higher level.

Work with focus on picking out components to form configurations and products is more scarce. The work of Vanhatupa et al. [10] focus on how to specify configurations in an XML-like language allowing the description at a fairly high level to avoid too many details. They do consider both implicit and explicit dependencies between components to be taken into consideration during the composition. However, there is no notion of high-level rules nor do they take the versioning aspects into consideration. In Thao et al. [9] the handle the versioning aspects since they provide a full-fledged version repository. They are able to propagate changes in a component to the products where it is used and vice versa. The system can also detect conflicting changes to a component used in different configurations. However, they also have no or only little notion of rules – in particular for business and legal purposes. Likewise their version repository lacks descriptive power for representing facts and information about components and configurations.

During our work, the role of a third party supplier surfaced. It was indicated that it might become important in the future, but for now we consider this role to be covered by other roles. However, an important difference from these roles is that a third party supplier is an "outsider" and as such might not be granted the same access rights.

We have also identified an **End User** role. The **End User** is the person who actually uses the product (a mobile phone). We have not included this role in our analysis as it has a completely different nature from the others roles. However, it could have the following use case: *Update configuration.* Just like we get new updates to our computer's applications or services. We will get new versions of existing components in the configuration, but most likely no new components or applications. *Create new configuration.* The new configuration is the old one plus one or more new components (applications) that should be added – we should get the right version and be warned if the new configuration is not possible.

In the present work, we only considered software components. However, in our context and in general, products also consist of hardware parts. A natural extension to our work would be to include hardware components and the way they are handled by Product Data Management [2]. So much more as there are many commonalities between Product Data Management and Software Configuration Management.

5. Conclusions

In this paper, we have addressed the issues and challenges in providing a common, high-level composition model that can be used by everyone involved in working with products. Such a common model will allow people to work with, share and exchange configurations in a safe and controlled way.

We analyzed product composition from the perspective of different stakeholders and identified a number of important issues to address and challenges to overcome. Our analysis was primarily based on information from interviews, but also drew on our expertise in software configuration management. The issues were grouped into three main categories: component base, configurations, and rules.

We also show that it should indeed be possible to construct a configurator tool – or design a set of manual processes – that can use intelligently data regarding the components to allow users to reason at higher levels about things like completeness and consistency of the configurations they work with. Most of the necessary techniques are already used in software configuration management. In such a setup the user will be able to manage the complexity of all possible combinations of components and versions.

In a context with simple configuration needs all this could in fact be in the head of one person knowing all the facts and working in a disciplined way. However, if that person should leave the company or the problem should grow more complex, we would like data to be explicit and persistent through the component base and the knowledge to be represented through explicit and persistent rules that can be used in a (tool supported) composition process.

We used one specific company as the object for our study. However, we did not use anything that is special to this company, so our findings should generalize to the whole sector of this company. Furthermore, the mobile phone industry probably has to deal with more complex and exotic composition situations than “the average company”. This makes us confident that we did not overlook any important issues due to a context that was too simple.

6. References

[1] Crnkovic, I., Larsson, M., *A case study: Demands on component-based development*, in proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, June 4-11, 2000.

[2] Crnkovic, I., Asklund, U., Persson-Dahlqvist, A., *Implementing and Integrating Product Data Management and Software Configuration Management*, Artech House, 2003.

[3] Dart, S., *Concepts in configuration management systems*, in proceedings of the 3rd international workshop on Software Configuration Management, Trondheim, Norway, June 12-14, 1991.

[4] DeRemer, F., Kron, H. H., *Programming-in-the-Large Versus Programming-in-the-Small*, IEEE Transactions on Software Engineering, Vol. SE-2, No. 2, June 1976.

[5] Feiler, P. H., *Configuration management models in commercial environments*, Technical report, Software Engineering Institute, 1991.

[6] Lau, K., Ling, L., Elizondo, P. V., *Towards Composing Software Components in Both Design and Deployment Phases*, in proceedings of the 10th International Symposium on Component-Based Software Engineering, Medford, Massachusetts, July 9-11, 2007.

[7] Oberleitner, J., Fischer, M., *Improving Composition Support with Lightweight Metadata-Based Extension of Component Models*, in proceedings of Software Composition, Edinburgh, Scotland, April 9, 2005.

[8] Scheben, U., *Hierarchical composition of industrial components*, Science of Computer Programming, Vol. 56, Issue 1-2, Elsevier, 2005.

[9] Thao, C., Munson, E. V., Nguyen, T. N., *Software Configuration Management for Product Derivation in Software Product Families*, in proceedings of the 15th International Conference on the Engineering of Component-Based Systems, Belfast, Northern Ireland, March 30 – April 4, 2008.

[10] Vanhatupa, J.-M., Hammouda, I., Korhonen, M., Kaskimies, K., *Model-driven Approach for Interactive Configuration Description in Component-based Software Product-lines*, in proceedings of the 5th Nordic Workshop on Model-Driven Engineering, Ronneby, Sweden, August 27-29, 2007.