A configuration management perspective on composing software product lines

Lars Bendix
Department of Computer Science
Lund University
S-221 00 Lund, Sweden
bendix@cs.lth.se

Jacob Gradén
Department of Computer Science
Lund University
S-221 00 Lund, Sweden
graden@gmail.com

Anna Ståhl
Department of Computer Science
Lund University
S-221 00 Lund, Sweden
aannastahl@gmail.com

ABSTRACT
The high demand for more products has led industry in the direction of using software product lines. However, there are still many issues to be solved before software product lines can be handled and supported just as well as "normal" software development. We have obtained some results by applying a software configuration management perspective to a restricted set of problems from product lines and identified a number of interesting open issues in the intersection between product lines, composition and software configuration management.

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management – Software configuration management, productivity, software quality assurance.

General Terms
Design, management, performance, standardization.

Keywords
Configuration management, component-based, product lines, composition, variants.

1. INTRODUCTION
The requirements of new products from the modern consumer market put many companies in a difficult situation. The lifetime of a new product becomes shorter and shorter, meaning that you have to create more products to remain in the market. On the other hand, products become bigger and more complicated and expensive to develop. One answer to these challenges has been to move from single products to product lines in order to produce more products cheaper and faster.

Moving from monolithic systems – which are often the result when products grow into different variants – to component-based ones has advantages, such as code reuse and flexibility in creating product lines. In monoliths, variants are created by conditional compilation or, in the worst case, a new copy is created for each product. In component-based systems, this is done by selecting different sets of components to create different products (composition). There are however also drawbacks, one of the more noticeable ones being that component-based systems require more design and attention to system structure.

One typical example of a more advanced use of product lines could be the mobile phone industry, where there are both product lines and customer (operator) variations. The component-based approach is not the only way to produce product lines, but it is promising. In this paper, we examine component-based product lines from a software configuration management perspective.

In the next section, we give a more detailed description of the background and specific context for our work into how SCM can provide support for software product lines. Then we present the parts of our results that are most relevant for this workshop and end the paper by giving an outline of a number of open questions that could provide interesting discussions for the participants.

2. BACKGROUND
A product line can be defined as a common core plus a part that varies between instantiations [10]. The same product line can give rise to many different instantiations; all of which share certain characteristics, but differ in others. Instantiations are often called *variants*, or simply products. Generally, variants are created by assigning specific values to so-called *variation points* – places in the product line where choices can be made on what the particular product should be like and that are different from the common core.

There are many ways to manage product lines: aspects, services, components, variation points, and more. Our premise in the following is that of component-based product lines, meaning that we restrict ourselves to looking only at that approach to product lines.

Software configuration management (SCM) is a general support discipline for software development [1]. SCM covers a wide range of topics from versioning and traceability over reproducibility to more formal activities as configuration audits. SCM has had a look at support for variability management in general before [2, 5] but without much luck or progress. However, this time we only look at supporting a limited part of variability management (composition of product lines) and with focus on one specific technique (component based systems).

One very central activity in SCM is that of composing configurations [3]. This is mostly done for creating developer workspaces, but there is little or no difference between creating a configuration for a workspace, and creating a configuration for use as a product to be shipped.

There are different ways offered by SCM techniques to represent variants, notably *single source variation* and *variant segregation* [9]. Single source variation keeps everything in one large container and extracts the particular parts which are needed to instan-

tiate a certain product – a method commonly used in monolithic systems. Variant segregation maintains different versions of each part, and simply combines the appropriate parts when the product needs to be instantiated – which is a suitable method for component-based systems. Both methods, however, have advantages and drawbacks.

In the context where we have carried out our investigation (the mobile phone industry), we deal not only with lines of products, but each single product (a specific model of a mobile phone) may also have some parts or components that are tailor-made customizations. This means that a product can in itself also be a product line. In fact, this can happen in many steps: A very general platform can be considered a product line, which is instantiated to create both high-end and low-end products. The high-end instantiation can then be considered a product line, which instantiates to different phone models, which are in turn instantiated, until an actual, physical phone is completed.

We have a working context – and industry case – where software development creates a number of components in a number of versions (variants and revisions). They are compiled into binaries that are combined with a description (meta data) and put into the component repository, as depicted in Figure 1.

For monoliths, there is normally just one repository where all source code is stored, and products are created directly from it by compiling the relevant source code – which, for large systems, may take a lot of time. In this context, there are two repositories: one used for the development of source code (corresponding to the items managed by Software developers in Figure 1); and one repository where components are stored as pre-compiled items (the Repository to the right, managed by Configuration managers).

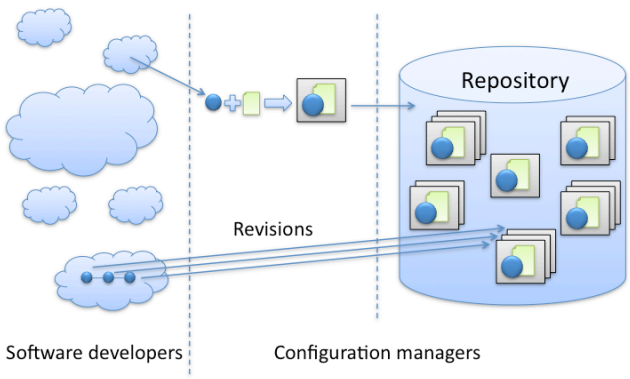

Figure 1. Component creation

The component repository holds all components in all their versions, and can therefore be used to create every possible variant of the system. In essence, the component repository corresponds to the product line. When a particular variant needs to be created, the corresponding components are then extracted and combined. There is no need for compilation at this stage, which keeps the time required to instantiate products down – even creating many different products at the same time can be done very quickly. The process of instantiation is show in Figure 2 below.

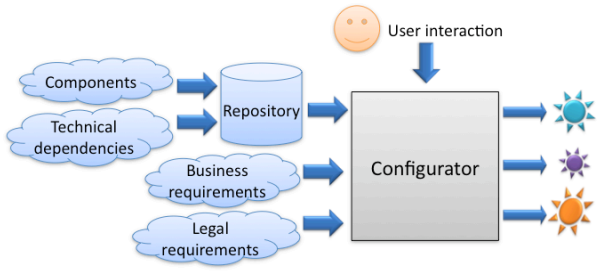

Figure 2. The composition process

There is much complexity inherent in configuration composition. This complexity stems from three sources.

- The components to choose from – different applications, for example.
- The revisions of these components – such as new versions of a specific application.
- Variants of components – different alternatives of the same component, which serve slightly different purposes. This can be seen as "internal" product lines, inside the components.

In fact, there is even more complexity, since there are *relationships* between components. Relationships restrict which components can be combined, and in what way – and relationships may even change between different versions of the same components.

The goal is to compose *complete* and *consistent* configurations – essentially, to only instantiate useful variants of the product line. If components were truly independent there would be no problems with consistency, but completeness (having all functionality) would still be an issue. However, even if components are technically independent there may be legal requirements on how they may be combined, or simply business considerations. A configuration can therefore be complete and consistent from different perspectives, depending of the context.

Once we have obtained this goal, we want to be able to actually work with the configurations. It would be useful to be able to ask questions about the configurations, to quickly create mock-ups of possible configurations, and to perform sanity checks on configurations – for example to make sure that all components can in fact be used, that there are no duplicate components, etcetera.

3. RESULTS

We studied the context described in the previous section to analyze the potential and problems in composing products from component-based product lines and sketch how that could be done. The results most relevant to the workshop themes are the following:

- The composition of components from component-based systems into product variants can be assisted by introducing a support tool to manage the considerable complexity.
- Such a tool can also reduce the lead times, increase efficiency and lessen the risk of human error, by automating tasks which would otherwise have to be performed manually.
- As many decisions as possible should be taken automatically by the tool. Ambiguous or undefined decisions must be handled by humans.

Additionally, several mechanisms can be identified, which should be part of this kind of tool.

Relationships. Components should remain mainly independent of each other, but sometimes there are good reasons for allowing *dependencies*[1] between them. A component can *require* or *conflict* another; or may *recommend* or *suggest* one. Components can also *replace* or *break* each other; and for the dynamic composition problem, *pre-requires* is necessary. The abstract concept of relationships is in practice handled by rules.

Metadata. Components can have properties. So too can configurations, services, layers and rules. Properties can be either *static* or *dynamic*. The former are set by a user and are absolute; the latter are calculated by the system when required. For instance, the property "last modified" on a configuration may dynamically depend on the corresponding properties of its constituent components. Calculating dynamic properties requires knowledge of relationships and can reduce unnecessary work – for instance by predicting that a configuration will not work, based on the test status of its components, thereby precluding the need for testing the configuration.

Layers. Components can have very different roles in a component-based system, and a suitable way of structuring the components is a layered system. Components on one layer are allowed to require that other components on the same layer, or lower layers, are present, but must not require that higher-layer components are present. This facilitates a good system structure and helps to determine if a configuration is complete. Examples of layers could be *operating system*, *service layer*, *applications* and *customizations*; but many more can be envisaged.

The exact number of layers may vary for different configurations, as may the requirements on each layer – for instance that exactly one component from the lowest layer is included in a configuration, and that more than zero components from a higher layer are included.

Rules. Rules are used to express relationships between components, to evaluate which components must or must not be included in a configuration, and to state what traits a configuration must exhibit. Generally, a rule consists of one or more *conditions* and one or more *consequences*.

A *condition* can refer to a configuration – or one or more specific versions (variants and revisions) thereof; a component (in any given version); a service (in any given version); a layer; a rule (in any given version); current date and time; a property of a configuration, component, service or rule; and more.

A *consequence* can be to require that a component or service must or must not be included in the current configuration; that a property must or must not have a specific value, or that a property is assigned a value; that a rule is or is not to be evaluated; and more.

[1] Similar to Debian's notion of dependencies.

Rules belong to different *categories*, being for example technical, legal or business-related. They can also have *priorities*, to decide the order in which they are evaluated. A considerable amount of work is required to produce good rules, but the benefit is that implicit knowledge is codified and can be used by the support tool to automatically perform work that would otherwise have to be done manually – for example checking consistency or automatically completing a partial configuration.

Completeness. A configuration is complete if it *contains all components* which are needed, in correct versions. A component can be needed because of a relationship from other components (transitive closure); from fulfilling layer requirements; as a consequence of a rule; or directly because the user of the support tool asked for it.

Consistency. A configuration is consistent if *all rules are adhered to*. Specifically, no components should be in conflict with each other; and only certain cases of replace and break are allowed. Also, all property consequences must be followed.

Sanity. A configuration is sane if all included components fulfill a *useful purpose*. A configuration containing only an operating system and nothing else may be both complete and consistent, but can be used for no sensible task; or a service may be used by no component, in which case the service should probably not be included.

Repositories. We find a need for two different repositories – one for source code, in the ordinary fashion, and one for components and configurations, where greater control is exerted before commits are allowed. Each repository has its own purpose and different people using it in different ways. Making the distinction explicit will simplify many things and make it easier to implement a special repository that suits the needs for composing configurations from a set of components.

Commit checks. The latter repository can be expected to grow over time, in both space and time (variants and revisions). To maintain some level of cleanliness, changes to the repository need to be screened – or the contents risk ending up with unnecessary relationships or even mutually incompatible rules. At best, this will lead to a bad architecture, and at worst, composition will be rendered impossible.

Commit checks can typically involve making sure that the committed component version can be used, at all, in a configuration[2]; that all existing rules are still valid after the commit; that new rules are consistent with existing ones and with current configuration and component data; and otherwise maintain a sane state of the repository. Not all problems can be caught using this mechanism, but the structural problems can be, leaving only the dynamic problems of actually creating a configuration.

More results and more details about the results above can be found in [4].

4. FUTURE WORK

Our odyssey into the world of constructing configurations and products from a base of components has uncovered just about as many new questions as it has provided answers – if not more. In the following, we outline a number of interesting new questions

[2] It is possible to set up rules in such a way that a component is in conflict with itself, for example using cyclic anti-dependencies.

that are still to be explored more closely – some related to SCM, some to composition, some to software product lines – but most of them have aspects from all three.

The discussion questions have been grouped into two themes: Once a configuration has been constructed by completion, what kind of information can help us with how we verify that it is also a valid composition – and how can we capture that information? Could we use composition information in the construction phase so it becomes more than just a simple completion – or are there other ways to compose configurations from software product lines?

The two proposed discussion themes are not necessarily completely disjoint.

4.1 What can we do with configurations?

Constructing configurations is a complex thing. People will mess it up, so they need tools to help them. Once a configuration has been created we need to help them verify a number of things about the configuration: completeness (whatever is the definition of that), consistency (we can think of several variants of a definition for that), metrics, and more.

Rules. The foundation for rules is presented in section 3 "Results", but many questions remain. What, for instance, is the best way to represent rules – separation of facts and logics; Turing-complete languages; simpler, more pre-defined mechanisms; or entirely different approaches? In addition to these questions, many of the problems outlined below can probably be handled by using rules. Should the rules be set up "globally" for the verification system or should it be possible to have "personalized" rules for each user/product? What new advantages and challenges does this bring? How can products be reproduced if rules change? Should it be possible to dynamically add/modify rules during the selection and/or verification process?

Status. Can a configuration's status be automatically determined from constituent component statuses, their relationships and rules? How should this be done? What advantages would this bring? One use we have in mind is for determining the test status of a configuration, but are there other areas where this idea can be applied to assist product composition?

Relationship automation. Can relationships between components be found automatically – and if so, how, when, under which circumstances, and by whom? Which relationships can be found, and which cannot? Which tools are available for finding them (code analysis, automated smoke testing with different components, statistical analysis, reviews of component histories), and how reliable are they?

Repositories. The commit checks, mentioned in section 3 "Results", are merely one aspect of repository management. The number of components, versions and variants of components will grow by time. Do repositories need to be cleaned out from time to time? If that is the case – when, how, and by whom? Can repository contents ever be allowed to be truly deleted, or should they be archived somewhere (for example for traceability reasons)? What are the advantages and drawbacks? How large can repositories be allowed to become, and how should this be measured – which metrics are most pertinent? Do the same rules apply for different kinds of repositories – requirements, design, source code, binary components and configurations? And what kinds of repositories are there, anyway?

Layers. Is a layered structure, as the one we propose for our specific context, also a support in general when working with component-based system to produce products in a product line? Can a layered structure also work as a support for not letting the repository of components evolve towards a monolithic system? Or can a layered structure harm the evolution of new features by forcing components to be in a fixed structure? Is the layered model even a good structure? Would a two-dimensional (or more) structure be more functional? Are layers more than just having "classifications" of components and a rule stating that we must have exactly one component of classification OS?

4.2 How to build configurations?

Can composition information in the construction phase be used so it becomes more than just a simple completion? What is really a product line? How should it be represented – and what about the products in the product line? How do we get from the product line to the product(s)?

Configurations in repositories. Assuming that configurations are stored in a repository, what information must be saved? Should each and every version of every configuration be stored in full detail and with full information about all properties, or are subsets of this data sufficient? What should the change process for configurations look like? Can several people modify the same configuration at the same time, and how should this be handled? What models and policies should be in effect? Are these the same as for source code?

Static vs. dynamic component selection. We can have loosely-coupled components that do not need a specific component, but rather a specific service that can be provided by a number of different components – in this case, is dynamic or static composition the best alternative? What are the impacts of statically requiring a component to use a certain component, or enabling it to choose among components dynamically at runtime? Should the system maintain the list of services of each component, or should there be a common interface which components can use to query each other? What are the advantages and drawbacks?

Variation points. Variation points were not the primary concern in our work so far, component-based development was – but variation points can be used even "inside" components, so they become interesting also in a component-based context. The growing used of variation points for product lines could bring back the old – and little understood – fragment system [8] that was built on the concept of "syntax-directed" modularization [7] and could be used for aspect-oriented programming [6]. Could these concepts be taken to new levels of use, now that people have caught on to variation points?

End-user composition. Though we have worked in a "producer" context so far, we should also more generally look at the "customer" context, where new services/applications are downloaded to an "environment" that we have little or no control over. What components will work in a specific environment? What other things does it require (that may or may not be there)? What version of the component should we get – and does that require that we get newer versions of other (depending and dependent) components? What other new challenges does this brings?

Components of components. What advantages are there by being able to display a set of components as one component and have different views of the components in different situations (as seen in Figure 3)? What happens to the relationships and properties of

components inside the component? Is it useful for the compound component to inherit relationships and properties from its included components? What advantages and drawbacks are there from this? Are there other ways to do this? Should the relationships and properties be classified as internal and external properties?

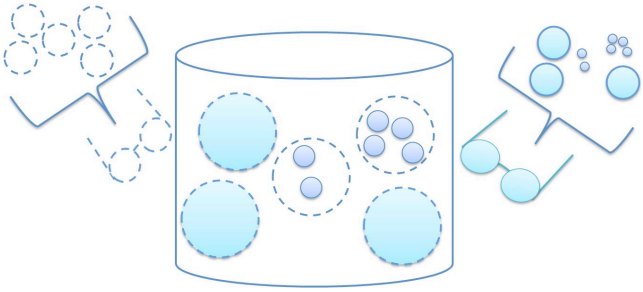

Figure 3. Perspectives on components

Configurations as "first class objects". In many situations we could be interested in treating configurations as first class objects. To be able to treat them in the same way as we do with components. What are the advantages of being able to treat a group of components in the same way as a component? Should it be possible to work with bound configurations only or should we also allow partially bound configurations? What relation do the unbound components in such a "configuration" have to the concept of variation points?

Variation points in components. Since a component can be a set of components, can such a component have un-bound variation points? How does this affect completeness and consistency checks? Is it still possible to specify a set of rules to assist in the verification of completeness and consistency?

5. CONCLUSIONS

We approach this workshop with a hammer (SCM) in our hands and are curious to explore how much of what the workshop deals with is actually nails (issues that can be solved of alleviated by SCM techniques). Furthermore, we would be interested in finding out which nail-like things (issues) there are around and how they – or our hammer – could be restricted or modified to create workable solutions. Where and how is it that nails (and screws) are used in product lines and when is it that glue would work better – and when and how to use the hammer appropriately?

We believe that our SCM perspective on component-based composition and the results we have obtained so far will generalize to aspects, services and other ways of dealing with software product lines – at least on the level of the SCM problems faced.

However, *the authors* also expect to learn something from the "composition" and the "software product line" communities that we can feed back into our continued development of SCM support for software development in general and of product lines in particular.

One final addition – we have started to look deeper into adding the version and/or variant dimension that has not been very prominent in our work so far [4]. How to handle that evolution may change the structure of the "base product" – or how to handle that the system model changes in Feiler's "composition model" [3].

6. REFERENCES

[1] Wayne A. Babich: *Software Configuration Management – Coordination for Team Productivity*, Addison-Wesley, 1986.

[2] Lars Bendix: *Software Configuration Management Problems and Solutions to Software Variability Management*, Proceedings of the ICSE Workshop of Software Variability Management, 2003.

[3] Peter H. Feiler: *Configuration management models in commercial environments*, Technical report, Software Engineering Institute, 1991.

[4] Jacob Gradén, Anna Ståhl: *Managing product variants in a component based system*, Master's thesis, Department of Computer Science, Lund University, November 2009.

[5] Andreas Hein, John MacGregor: *Managing Variability with Configuration Techniques*, Proceedings of the ICSE Workshop of Software Variability Management, 2003.

[6] Jørgen Lindskov Knudsen: *Aspect-Oriented Programming in BETA using the Fragment System*, Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99, 1999.

[7] Kristensen, B. B., Madsen, O. L., Møller-Pedersen, B., and Nygaard, K.: *Syntax Directed Program Modularization*, in: Interactive Computing Systems (ed. P. Degano, E. Sandewall), North-Holland, 1983.

[8] Ole Lehrmann Madsen: *The Mjølner BETA Fragment System*, in J. Lindskov Knudsen, M. Löfgren, O. Lehrmann Madsen, B. Magnusson (eds.): *Object-Oriented Environments: The Mjølner Approach*, Prentice Hall, 1994.

[9] Axel Mahler: *Variants: Keeping things together and telling them apart*, in Walther F. Tichy, editor, Configuration Management, John Wiley & Sons, Inc., 1999.

[10] Klaus Pohl, Günter Böckle, Frank J. van der Linden: *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer Verlag, 2005.