Analysis of Issues with Industrial-Strength Composition for Component-Based Systems

Lars Bendix[1], Jacob Gradén[1], Anna Ståhl[1], Andreas Göransson[2]

[1] Department of Computer Science, Lund University, S-221 00 Lund, Sweden
[2] Sony Ericsson Mobile Communications AB, Nya Vattentornet, S-221 88 Lund, Sweden

bendix@cs.lth.se, graden@gmail.com, aannastahl@gmail.com,
andreas.goeransson@sonyericsson.com

Abstract. A component-based approach can reduce time and costs for creating new products. 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 versions or variants. Furthermore, there are many different stakeholders involved in the creation of new products and they all work at different levels of granularity of the product.

We identify use cases for each involved stakeholder. From those we derive a set of issues to a high-level common model (and tool) that can be used by the many different people involved in creating, handling and using products. In this way it will be possible to unify their different needs in a common framework.

Keywords: High-level composition, configurations, common model, use cases, configuration management.

1 Introduction

In today's market many products have a short lifetime 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 obtain different products.

However, it is no easy task to *create* products from such 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 versions 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. A configurator tool that intelligently uses data regarding the components and is based on a general composition model would be able to help a user *manage the complexity* of that explosion.

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 problems when different representations have to be kept synchronized or converted between each other. We should therefore look for a *common representation and model* that all involved stakeholders can use.

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

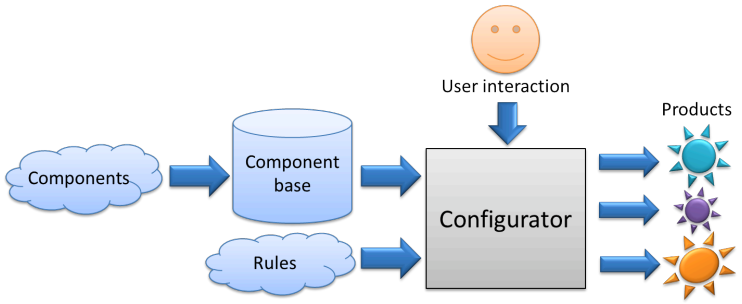

Figure 1. The configuration process.

The configuration process investigated in this paper is not about 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 in creating a common configurator that can be used by all stakeholders at the macro-level as shown in figure 1. Components are entered into a component base from which the stakeholder (user interaction) picks a number of components and with the help of the configurator, that uses a set of rules, a product (or configuration) is created and worked on. Can such a configurator be built and what are the issues that need to be addressed?

The specific context that we have based our work on is the mobile phone industry. This is a field where there is a high pressure for new product 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 and want to explore "our established models" [4] for creating workspace configurations from a repository in the context of creating products from a component base.

In this paper, we first identify the different roles that are involved in the production of products and for each role we present the tasks they have to perform as use cases. Next, we analyze these use cases to arrive at the issues of an industrial-strength composition system. Then we discuss our results before finally drawing our conclusions.

2 Use Cases

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. In order to avoid that, we want to investigate the possibility of using a common representation for all the involved stakeholders.

The first step towards that 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 and what they would want a configurator tool to be able to do to help them.

2.1 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. Sales would be interested in creating configurations of new products that can be put on the market. They want to be able to work in different ways: bottom-up, where they select component from the bottom up to create a given product; top-down, where they specify some functionality and explore which components are needed to create the product; and mixed modes. It is implicitly included that they can "save" a created configuration and work on it at a later time.

Configuration verification. Sales want to verify that a given configuration of a product is complete and consistent. Sales use that to "explore" the potential consequences of new hypothetical products. What functionality is missing to complete the desired product, are there components that are in conflict with each other, are there any violations of business or legal rules?

Configuration inspection. The inspection of properties of a configuration (or a component) is another sought after possibility. To be able ask questions about the state or other properties of components and configurations – there are a number of standard queries, but they also want to be able to handle ad-hoc queries.

Configuration quotation. Sales wish to create and save an invalid or incomplete configuration, and then ask development and/or design what would be the cost to have it produced.

2.2 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 role 2.6 below):

Create configuration. Configuration Management has to set up a working environment (workspace) for many different groups of people and in different ways for the same group of people. Part of such a working environment may be specified with components that come from the component base and part of it may be specified with files that come from the source code repository. Some people (developers) would like to work with *partially* bound configurations where the version selection is left open for some parts. Other people (testers) would like to work with *completely* bound configurations where all version selection has been resolved. Configuration Management will need flexibility to be able to cater for working environment needs of all groups.

2.3 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. Apply and use as working environment a configuration that has been defined by others.

Problem resolution. Designers would get a (hypothetical) configuration that they need to analyze for what is needed to make it work – it might be incomplete or inconsistent. There might need to be created new components or existing components could be modified. In some cases it could be that additional testing of (parts of) the configuration was needed.

Create placeholder components. When new components are designed they are first created as a placeholder component that has no actual contents. The developers will then eventually create the contents for the component. However, it should still be possible to use placeholder components (maybe in restricted ways in some situations) in the creation of configurations.

Manage rules. Designers sometimes introduce new business rules or register legal rules. Maintain and create rules to ensure that, for example, only one component is selected from a specific layer – or other architectural or design rules.

2.4 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. Apply and use as working environment a configuration that has been defined by others.

Build product (or configuration). Developers will need to build an actual product from a configuration so they can see that their code runs and performs like it should.

Record dependencies. Technical dependencies between components are set by the developers who are the people that have this information.

2.5 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. Apply and use as working environment a configuration that has been defined by others.

Build product (or configuration). Testers would be interested in getting a completely bound configuration and build the product they have to test from that. In alternative they could have gotten an already built executable to test.

Create configuration. Testers sometimes want to modify their given configuration in case something does not test and they want to see if it tests in a slightly (newer/older/alternative version of a given component) different configuration.

2.6 Component Management

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. There are many administrative tasks around the component base and its contents. New (or versions of existing) components need to be added. Deprecated or faulty components should be cleaned out or at least marked accordingly. The (meta-)data for components has to be changed or updated. Furthermore, it should be possible to run checks on the component base to see that it is still in "good health" (that components are still fairly independent so they can be combined in flexible ways). As it works today with source code, Configuration Management adds new versions to the repository at the request from developers.

Manage rules. Besides the architectural rules that are managed by the designer, there are also business and legal rules that have to be managed. Because these rules can come from many different places, it is Component Management's responsibility to add and/or change such rules. Component Management should also be in charge of making sure that rules are consistent.

2.7 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. The customer wants to be able to create new configurations. In some cases they are interested in seeing what they can get based on existing components and to "order" a new product. In other cases the customer in interested in getting the price for a "hypothetical" product that is not yet ready/possible – that could be for special customization of some component(s). Often the customer will do this work on his own, but sometimes it is done together with Sales.

We also identified some use cases that it was not possible to place on any user role: **Notification**, **Statistics**, **Quality aid**, and **Requirements fulfilment**.

In the next section, we will carefully analyze the use cases to extract and identify what problems need to be solved and what kind of functionality the configurator system (the whole setup) should provide to be able to support all user roles and all user tasks.

3 Analysis

This section presents an analysis of the problems found in the use cases above and suggests possible solutions. For more details, please refer to [5].

From the use cases in section 2 and many discussions with people we identified a number of issues that have to be dealt with in order to develop a common model that can be used for a configurator tool.

There are many technical aspects to consider regarding the configurator, but because the tool is intended for several disparate groups of users, usability concerns are also very important. The configurator must be simple enough to be used by persons 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 configurator must 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 – the configurator should operate in *real-time*. It would be detrimental to user understanding if this were not the case.

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

3.1 Components

Component-based systems are built from individual, stand-alone components, which interact using well-known interfaces. A typical component could be an application or the operating system.

Components are developed over time, and changes cause components to come in new *revisions* or new *variants*. Revisions and variants are collectively called *versions*. They create possibilities, but also complexity. Components may also need other components to work properly (*relationships*), can be combined into *suites* of several components, can provide *features*, and can additionally have *properties* (section 3.4).

Relationships. In their most general form, relationships are connections between two components, which say something about how they affect each other technically. When components affect each other in non-technical ways, *rules* are used instead of relationships (rules are explained in section 3.5).

Relationships always relate to components or versions thereof, whereas rules can be more general and relate to components, properties of components, other rules or relationships, etcetera. It also makes sense to manage the technical relationships in close connection to the components they relate to, whereas rules are handled on a higher level of abstraction and are bound less tightly to components.

Relationships are specified with one of two basic premises: Either two components are allowed to be combined unless there is a relationship between them stating the opposite; or two components may be combined only if it is explicitly permitted. The second approach is better at guarding against poor combinations, but the cost is that if most components can be freely combined, which is often the case, this leads to a staggering amount of relationships. The premise that all combinations are allowed does not suffer from this drawback, and also encourages separation of components, which leads to a greater number of possible configurations.

The most fundamental relationships are *requires* and *conflicts*. If component A requires component B, then A will not work without B – B, however, will work without A. If A conflicts B, then A will not work with B; nor B with A.

Less fundamental but still rather common are *replaces* and *breaks*. Replaces is commonly found when one component is deprecated in favour of another, and even if one component replaces another it does not necessarily mean that they cannot work together. If they cannot, breaks is the better relationship. In many situations, the two complement each other.

Some relationships can be found automatically, while others need to be added manually. Requires is among the easier to find, since it is often made explicit, for example in the form of include statements. It is however not trivial, since knowledge of the source language is required, and not always possible to find automatically. Most of the excluding relationships must be found manually, for example through testing – conflicts and breaks are prime examples of this.

Component suites. Sometimes, components are strongly connected and commonly used together as entire component suites. This is a powerful way of simplifying administration: instead of having to select several different components, one single suite can be selected. A clever way of providing suites is to create a new component – the suite itself – which has no code and provides no functionality. Instead, the suite has relationships with its constituent components. Selecting the suite means the configurator will automatically select all required components referred to by the suite. While not as problem-free as one might imagine, this is a powerful tool and a great help to the persons creating configurations.

Feature dependencies. Sometimes, it is less interesting exactly which component is used, and more interesting to make sure that a certain function is available. For example, if component A requires the feature Foo, and B and C both supply that feature, then A will require either B or C, but just one of them, to function properly.

Two important characteristics of feature dependencies are that they allow for choosing one of several supplying components, and that a direct relationship is transformed into an indirect one. This means that A's dependency on Foo needs not be updated if a new supplier is added; all that has to be done is to specify that the new component, say D, provides Foo, and A will be able to rely on B, C or D.

A great advantage with feature dependencies is that there is no need for A to have relationships with specific versions of the other components – so long as they supply the Foo feature, they are usable. For the same reason, components can be freely updated with new revisions or variants, or deprecated, and no relationships need to be updated. This means that feature dependencies are very useful in limiting complexity.

A complication is that even features may be developed over time – Foo may for example be upgraded to Foo version 2, which must be tracked. Sometimes, but not always, a component providing Foo 2 also provides Foo 1. Another complication is that in some cases, only one component providing a feature must be chosen, whereas in other situations, it can be perfectly acceptable to select two or more components. This depends on the specific feature, and is best handled by using the conflicts relationship when the situation calls for it.

A final note regarding components is that they can belong to different groups, such as Operating system or Application. Such groups are not set in stone, and there could realistically be situations where one component belongs to several groups. In the simplest cases, groups are built on top of each other (applications clearly run on top of the operating system, for example) and are then called *layers*, but it is also conceivable that groups have more exotic relationships.

An important aspect of groups and layers is that they can help with the structure of components; and also that some layers are absolutely necessary and at least one component from such a layer must be present. Without an operating system, for example, no other components will work properly.

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[1], 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

[1] There are cases where even an empty configuration is useful, for example in planning configurations. An empty configuration can then be created and given properties and rules, and only afterwards be filled with components. See [5] for more details.

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 another.

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 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 compo-

nent, 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.

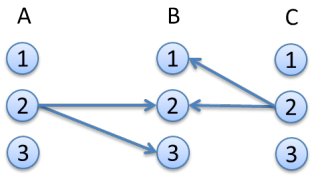

Figure 2: Late binding.

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 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 be used as components – remember: technically, component suites are configurations.

The question of exactly which form the component base and its storage should take is left 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 the 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 persons 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*, *What does the entire history of changes look like for configuration Z*, etcetera. 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 the 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, etcetera, 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 the configurator 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, the 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 anti-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.

3.4 Properties

Depending on the actual situation, a component (or configuration) may have any number of properties; for example its name or the date it was created. There are two classes of properties: Properties that are specific to the component or configuration itself (*static properties*), and properties that depend on the components a configuration contains (*dynamic properties*).

Static properties, such as name, are set directly and manually by a user, and should generally never be changed. Dynamic properties on the other hand are calculated from the configuration's components and their properties, according to pre-existing rules. Typical examples of dynamic properties are completeness and consistency. The effect of dynamic properties is that when the set of components in a configuration changes, that configuration may also have its properties changed, automatically.

Sometimes, static properties *can* be changed for a component. A typical example is the property Test status. A component may start as *untested*, and after scrutiny by a tester become *accepted*. This goes against the configuration management principle of immutable components, but is a necessary evil for developers and testers to work with the same component base.

It should also be noted that properties might affect the consistency[2] of configurations. For instance, if one component has the property Customer with a value of ACME, and another component has Ajax as Customer, then the two components cannot be used together in a configuration. This is because it makes no sense to target a specific product at two different customers. Certain properties are of such a type that all components in a configuration must have the same value for them, or must have no value at all (be generic).

3.5 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, etcetera.

Functionally, rules are a *mechanism* – a form of logic, or expressions. Rules are used to handle *facts* – components, properties, relationships, etcetera. 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"[3].

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.

---

[2] Completeness is not affected by properties, but by relationships and rules. Rules, however, may be affected by properties – so properties can indirectly influence completeness.

[3] As a matter of fact, this is the preferable way to handle completeness and consistency.

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 – etcetera.

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[4]?
- 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 work and sketch a number of possible future extensions to the work we have presented above.

4.1 Related work

To the best of our knowledge, there is no previous work that relates exactly to ours. 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. However, we have found three contributions that in part relate to our work.

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

[4] If not, each configuration, or each user role, or each user, could specify the rules, which should be active. There could even be rules for specifying which rules should be active, depending on which user and which configuration were involved.

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.

4.2 Future work

We are aware that the results reported in this paper are not final. Though it is a solid first step more work needs to be done before we can have "the perfect configurator". We see several ways in which the work in this paper could be extended.

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 also identified an End User role. The End User is the person who uses the product (a mobile phone). It is left out for the time being as it has a completely different nature from the others roles. However, it could have the following use cases: **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 a new component (application) 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.

Finally, the results in this paper are based on interviews with people about their work and visions for a future work context. Now that we have identified the important issues to address, a demo or prototype configurator can be built so these people can get more real hands-on experience than was possible from just looking at the design sketched in [5]. This could be used to validate the use cases and would most probably bring up some new issues now that the users see what they can actually do.

5 Conclusions

In this paper, we have addressed the problem of providing a common, high-level composition model that can be shared by everyone involved in working with products. We have shown that it should indeed be possible to construct such a configura-

tor model and tool that can be useful for this large and varied group of users who need to work with, share and exchange configurations.

Through interviews with various stakeholders in the company, we identified no less than seven different user roles (plus two more indicated in future work) that are more or less involved in creating and using configurations of products both inside and outside the company. For each user role we also distinguished its relevant tasks and sketched them as a number of concrete use cases.

We analyzed these use cases and identified a number of important composition issues that have to be dealt with to build a common, high-level configurator model and tool. Our analysis was primarily based on the use cases and interviews, but also drew on our expertise in software configuration management. The issues were grouped into five main categories: components, configurations, component base, properties, and rules. For each category a number of more specific issues were identified and discussed in more detail.

Some of the identified issues seem to have straightforward solutions, while others will probably need more research. Since there seems to be no prior work that directly matches ours, an obvious future work would be to use a demo or prototype of the configurator tool to conduct experiments with the users to validate our use cases.

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.

References

1. Crnkovic, I., Larsson, M.: A case study: Demands on component-based development. In: 22nd International Conference on Software Engineering, pp. 23-31, ACM (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: 3rd Workshop on Software Configuration Management, pp. 1-18, ACM (1991)
4. Feiler, P.H.: Configuration management models in commercial environments. Technical report, Software Engineering Institute (1991)
5. Gradén, J., Ståhl, A.: Managing product variants in a component-based system. Master's thesis, Lund University (2009)
6. Lau, K., Ling, L., Elizondo, P.V.: Towards Composing Software Components in Both Design and Deployment Phases. In: CBSE 2007. LNCS, vol. 4608, pp. 274-282. Springer, Heidelberg (2007)
7. Oberleitner, J., Fischer, M.: Improving Composition Support with Lightweight Metadata-Based Extensions of Component Models. In: Software Composition 2005. LNCS, vol. 3628, pp. 47-56. Springer, Heidelberg (2005)
8. Scheben, U.: Hierarchical composition of industrial components. Science of Computer Programming 56, pp. 117-139. Elsevier (2005)