Modelling Deployment using Feature Descriptions and State Models for Component-Based Software Product Families*

Slinger Jansen, Sjaak Brinkkemper

Institute of Information and Computing Sciences
Utrecht University
{slinger.jansen,s.brinkkemper}@cs.uu.nl

Abstract. Products within a product family are composed of different component configurations where components have different variable features and a large amount of dependency relationships with each other. The deployment of such products can be error prone and highly complex if the dependencies between components and the possible features a component can supply are not managed explicitly. This paper presents a method that uses the knowledge available about components to ensure correct, complete, and consistent deployment of configurations of interrelated components. The method provided allows the user to perform analysis on the deployment before the deployment is performed, thus allowing error prevention before making any changes to the system. The method and model are discussed and presented to provide an alternative to current component deployment techniques.

1 Component Deployment Issues

The deployment of enterprise application software is a complex task. This complexity is caused by the enormous scale of the undertaking. An application will consist of many (software) components that depend on each other to function correctly. On top of that, these components will evolve over time to answer the changing needs and configurations of customers. As a consequence, deployment of these applications takes a significant amount of effort and is a time consuming and error-prone process.

Software components are units of independent production, acquisition, and deployment [1]. Software deployment can be seen as the process of copying, installing, adapting, and activating a software component [2]. Usually the only way to find out whether a deployment has been successful is by running the software component. This leads to frustrating and complex deployment processes for both the software vendor and the system manager. There are many reasons why components that have been deployed onto a system cannot be activated and run. The factors that increase complexity during the steps of building, copying, installing, adapting, and activating a component are numerous.

To begin with, there are relationships amongst components. Components can explicitly require or exclude a specific revision of a component. Some components allow for

* This research was supported by NWO/Jacquard grant 638.001.202.

only one version of the component to be deployed onto one system, placing a restriction on the components that are to be deployed onto that system. If such relationships are not respected the deployments will result in missing components and inconsistent component sets. Secondly, deployments are also complex due to the fact that components can be instantiated in different shapes and forms, due to variability [3]. A component that supports variability can have different features that are offered to the user, which are bound and finalized at different times during the deployment of that component. The binding time of a component can be at different stages of the component deployment, such as build-time or run-time. Thirdly, the order in which components are deployed can determine whether the deployment process of a set of components is successful. Components require other components during the deployment process and these can be removed when the system has a limited set of resources. Also, when components exclude each other and different deployment orderings are possible, the possibility arises that one of these orders does not ensure correct deployment. The above holds especially for component based product families [4], where many different variants of one system are derived by combining components in different ways.

A components' lifecycle consists of different states, such as *source*, *built*, *deployed*, and *running*. Many parts of the process of a component going through these phases have been automated to do such things as COTS (Components Off The Shelf) evaluation, automated builds, automatic distribution, automatic deployment, and automated testing. Current component lifecycle management systems, however, do not support different component (lifecycle) types, variability, component evolution, and are not feature driven. One of the main reasons for initiating this research is that the current tools for component deployment [5] do not take into account both variability, different types of distribution (source, binary, packaged), and different binding times.

There are tools that can manage the lifecycle of components, such as Nix [6], the Software Dock [2], and Sofa [7]. These systems have downsides however. To begin with, Nix is a technology based on an open source environment that can guarantee consistency between components and allows for concurrent installations of components. The biggest downside of Nix is that it requires a system manager to "stop the world", i.e., to adjust all the components the system uses to include a component description and reinstall the system. The Software Dock can be used to deploy software using the XML based Deployable Software Description [8] for describing the software. The Software Dock does not support the complete lifecycle of a software component. It does, however, focus on the complete deployment process of software, including such states as *activate*. Finally, SOFA is a corba based component model that uses the OMG Deployment and Configuration specification [9], and is also focussed on the deployment of generic components. Sofa, as well as Software Dock, only assumes a very simple lifecycle with four states being *source*, *built*, *deployed*, and *running*. Of these three component tools, only Nix focusses on variable features provided by different instantiations of components, and only Nix discusses the opportunities for a transparent configuration environment [10]. Software component developers often use their own specific deployment tools or custom build checks to see whether the system on which the component is deployed satisfies all requirements for consistent and correct deployment [11]. The

developer therefore must develop its own models and formalisations to ensure a correct component deployment.

The situation described above calls for a generic modelling technique that can handle the complex issues that are introduced by the use of variable components that can be instantiated in different versions and forms on one system. Such functionality, of which none of previously evaluated systems above provide it [5], requires a central knowledge base that stores the variables that initialize the different varieties of component instances and a categorization of such knowledge. This paper presents a modelling technique that can support the deployment of a component in different versions and variants, and still guarantee consistency and correctness. The modelling technique is based on a central storing of the restrictions and knowledge about component features and the system, thus allowing all components to use such information for correct build, release, testing, deployment, and activation of software.

The rest of this paper is structured as follows. Section 2 proposes two types of describing the properties of a software component and its relationships to other software components. Section 3 describes how the knowledge can be used to create an instantiation tree of component instantiations by using the provided algorithm, thus enabling the user to reason about deployment of software components. The algorithm is clarified with an example. Finally, we discuss the proposed methods and models in Section 4 and provide some insight into our future work and the conclusions reached throughout this research in Sections 5 and 5.

2 Component Descriptions

Currently used component models generally do not support consistent and complete deployment. Most models used by conventional technology [5] such as InstallShield and RPM-Update, focus on the artefacts that make up a component. Next to that these technologies perform some very general dependency resolution and only support the ”Requires Always” relationship. These tools often have some scripting capabilities that can be used to check whether the right resources are available, if required components for the deployment processes are available, and to perform some pre- and post-installation checking of the artefacts. These qualities, however, are underemphasized.

The model proposed here is based on three viewpoints. To begin with, a component is not merely a set of artefacts. A component has a context that describes the relationships to the components, hardware, and configuration information that affect the component upon and after deployment. A component also has internal variability, influencing that context, which is bound at different times. Secondly, if a component model supports variability, component features must be communicated to the user. This allows for the user to select these features at different stages of the deployment, changing the context as the component is built, activated, copied, and run. Thirdly, components are available in different revisions. When relationships amongst components can be specified with a specific version number, many deployment problems can be averted. Most deployment tools, including Nix [12] and RPM-Update, already have advanced versioning and dependency resolution mechanisms.

To summarize there are four factors that make a the deployment of a software component with internal variability complex. Each of these factors is handled by the presented model using specific modeling techniques and model extensions. These are as follows:

- **States -** Components can exist on a system in different incarnations simultanaously. These incarnations, such as a *source* incarnation or *installed* incarnation, have relationships to eachother. States enable the modeling of the complete build and deployment process, by describing such relationships as “to build this component the source is required first”. The introduction of states leads from Figure 1(a) to 1(b).
- **Revisions -** Components are generally available in different revisions. Different revisions have different states, thus leading to a seperate set of states for each component revision. Such revisions are modelled in Figure 1(c).
- **Features -** A component can have variable internal functionality, depending on parameters that have been bound at several times during the deployment process. Such points in time are known as binding times. These features can be modelled using a feature description language. In the model provided, each revision of the component source leads to a new feature description, since the code and thus the variability options might have changed. In Figure 1(d) this is displayed by the addition of feature trees per revision.
- **Relationships -** Component states have explicit and implicit relationships to eachother, such as the *built* state of an e-mail client requires both an instance of a *source* state that an e-mail client (explicit) and an instance of a running build tool state with a c++ compiler feature (implicit). These relationships can be further classified into “requires always” and “requires once” dependencies. An example of a “requires always” is that a *running* state instance requires a library at all time during the existence of the instance. An example of a “requires once” relationship is when a build instance requires the compiler only during its instantiation.

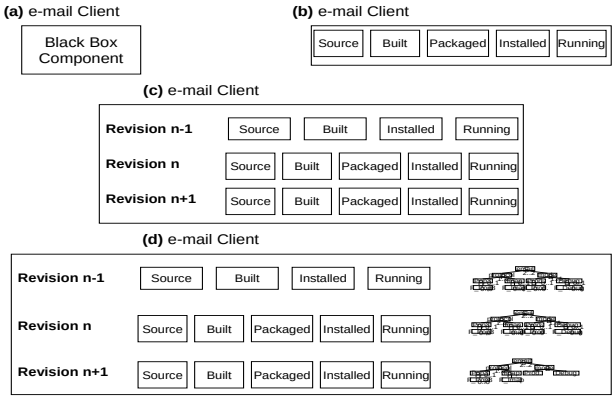

Fig. 1. Expanding Model for Software Components

To support the presented techniques a component description describes the component name, revisons, the revision’s states, and the revision’s feature diagram. This

definition shows that a component has one or more revisions. Each revision consists of a set of component states, a feature diagram, and a number of feature restrictions expressed by feature logic. The component states describe the shape or form in which a component can be present on a system. Examples include *built*, *activated*, and *running*. Component states can have relationships, such as *e-Mail client running always requires e-Mail client installed* or *e-Mail client built requires once a running build tool*. In the following sections these component states are described further. The feature diagram is used to describe the features a revision of a component supplies to the user and is defined using a feature description language (FDL) [13]. The feature restrictions describe whether features exclude or require each other. Figure 1 does not show feature constraints. These constraints are, however, an important part of our model. Features and FDL will also be explained further in the following sections.

2.1 Component States and Instantiations

The introduction of component states has many reasons. To begin with, component states force a developer to manage component relationships, restrictions, and deployment environment from the moment the component is created. Component states allow for a more detailed specification of component requirements. Some tools, such as Nix and SOFA, already include state models with the states *source*, *built*, *installed*, and finally *running*. Also, component states enable the component developer to specify and manage the process of how to create a component state instance.

A component state can generally be seen as a portable encapsulation format for a collection of artefacts, relationships to other component states, and a number of state instantiations. A state instantiation is a list consisting of actions and requirements that upon fulfilment of all the requirements performs the list of actions to reach the requested component state. In the presented model component states belong to one revision. A revision of the component can thus have a set of component states in which it can reside. The component state definition describes the component state's name, it's instantiation list, and it's relationships to other component states. The component state has relationships to other component states *Requires Always* and *Excludes*. These requirements are actually expressed as a combination of a component state and provided feature(s). This allows for a component state to have a relationship with a component state with a specific feature, such as *excludes(eMailClientRunning, Pop3)* which can be interpreted as "this component state cannot exist on a system concurrently with the eMailClientRunning state instance that provides the Pop3 feature".

Once the developer is forced to consider component states many possibilities arise. To begin with, the processes of automated building, testing, and deployment can all be performed using the same component state model. Secondly, since the developer can describe any type of component [14] the component model described here can be used to manage different component types, such as Corba or Java components, using the same model. Thirdly, since component state instances are portable, component state instances can be distributed amongst different systems. The model allows for derivation of component dependencies, and can therefore be used to create complete packages of component state instances to be delivered to customers. Finally, a component state model allows developers to model and reason about component updates. Component

states can have relationships with component states from other revisions, thus enabling modelling of complex patch or update processes. Such processes are, after all, nodes in the instantiation tree, which represents the full update process.

Previously some criticism was expressed toward a four component state model. The main reason for this is that evidence was found, during case studies at a number of software vendors, that there is a need for more states. The software vendor Planon [15], for instance, has a component state model with seven states, being *source*, *built*, *packaged*, *packaged with license*, *installed*, *activated* and *running*. Another software vendor we encountered applies six states, being the same ones as Planon but without the *packaged with license*. This software vendor builds plug-ins for Autocad for which the software vendor actually adjusts the component state model. The software vendor first unpacks Autocad from its installation package, then binds some variabilities, and then packages Autocad with their plug-in. Clearly, a component state is added to the Autocad state model as well.

A component state description is only a description and does not have any effect on a system. To create a state instance on a system, a component instantiation is required. A component state instantiation consists of a list of actions and a list of component state instances that are required to execute the instantiation and create an instance. The component instantiation consists of a *Requires Once* list and an action list. The *requires once* list shows what component states and features are required once the instantiation is activated. The *built* instantiation will generally require a compiler and a component state instance of *source*. The actions are specified as a tuple of *(precondition, action, postcondition)*. These actions usually are operations on artefacts, such as copy or edit actions.

To clarify the concepts of component state description, component state instantiation, and component state instance the following example is used. Figure 2 displays a compilation component, its feature tree, and the binding time of the feature. The feature tree can be interpreted as follows. The compilation tool has one main feature, that is bound as soon as the component is instantiated, called “build”. The compilation tool also has two features that mutually exclude each other (one-of). The next section provides more information on the feature descriptions at hand. When executed, the compilation tool can build either C++ and Java code. The user binds this feature at run-time, i.e., when a developer wishes to compile his Java code, he will state at start-up time that the code to be compiled is written in Java. The R stands for *running* and corresponds to the *running* component state. It is necessary to remind the reader that the figure does not show anything about the state of the system. The system can contain just the knowledge about this component, but also multiple instances of this components’ states, such as two installed versions and one running.

For this example a system containing an installed version of this compilation tool is used. That implies that the component state *installed* has been instantiated on the system once. This component state instance can be used to create a *running* instance. The relationship for the build tool is “Compilation Tool Revision 1: State Running Always Requires Compilation Tool Revision 1: State Installed”. The fact that it is a *Requires Always* relationship can be derived from the fact that it is the state that requires another state, and not an instantiation that requires another state. The *Requires Always*

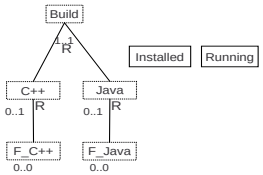

Fig. 2. Compilation Component Example

relationship describes that as long as a component state instance is present on a system, the required component must be present too.

Once the presence of the *installed* state instance has been confirmed, we must check for feature bindings. In this case that means a choice must be made between Java or C++. Once the right language has been chosen the instantiation of the component state can be performed. As mentioned before, it is well possible to instantiate a state multiple times, to do a parallel compilation of different source files, for instance. The aim of the algorithm described in Section 3 is to create an instantiation tree of component state instances, instantiations, and features. An instantiation tree for this component revision is quite simple, since no instances from other components are required and the component only has one revision. It will consist of two nodes, with the node "Compilation Tool Revision 1: Running" depending on "Compilation Tool Revision 1: Installed".

2.2 Feature Diagrams

To express variability we use the varied feature description language (VFDL). VFDL is a succinct, natural, and non-redundant language [16] that can be used to express features of components or products within a product family that contain any number of other components. The VFDL describes *and*, *or*, *mutex*, *xor*, and *requires* feature relationships. The *and* relationship is described by using a variation point that states that each of the features must be selected, by stating "S..S", where S equals the number of available features. An *xor* relationship can be described by introducing a variation point with two children stating "1..1", which means that one and only one feature can be selected. In case an *or* relationship must be represented a variation point is introduced stating "1..S", where S is the number of nodes and 1 means that at least one must be chosen. An *optional* relationship is described by adding a variation point is added stating "0..1" and using F_node is added that can either be chosen or be ignored. If two features exclude each other, they share a top variation point (using "1..1"), and each feature is optional.

The advantages of using a feature description language to express variability are numerous. FDL allows us to describe complex composition relationships, such as *one-of*, *optional*, and *more-of*, for features. If we then annotate these features with component state requirements it enables the creation of large component compositions. This is best clarified with an example of an e-Mail client that can both support the IMAP and Pop3 protocols (see Figure 3 for its feature tree). The binding time of these features is at install time. This means that one or both of these protocols can be installed. If these features have requires relationships with an IMAP and Pop3 component, it becomes

possible to deploy (and build) only the minimal required set, which is useful for space restricted systems such as mobile phones. If a user chooses the IMAP protocol, only the IMAP component needs to be deployed onto his system.

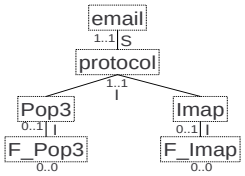

Fig. 3. e-Mail Client Feature Tree

Another advantage of using FDL to describe our feature model is the fact that there are many tools available to perform calculations and operations on the feature descriptions. More specifically, the techniques developed by van der Storm [17] allow for automatic composition of components using feature trees. Many of his techniques are reused here.

To satisfy the research goal of also incorporating binding times, each relationship between two features, such as *one-of* and *more-of* is annotated with a binding time. Binding times are directly related to component states in our model so each of these relationships is annotated with a pointer to a component state. Next to that, features have two lists of requirements attached to them, being *requires once* and *Requires Always*. Features can thus require component state instances and other features.

3 Instantiation Trees

One application of using feature descriptions and component state models is the creation of instantiation trees. An instantiation tree can model a number of instantiation sequences to reach a certain state or feature. This section describes algorithm 1, which creates an instantiation tree from a number of component descriptions and component state instances.

The algorithm uses some functions that are explained here. The function *returnFeatureBindings(State)* returns all features than can possibly be bound by instantiating the component state *State*. In the example shown in Figure 3 the function will return Pop3 and IMAP if called *returnFeatureBindings(Installed)*, which means that these features must be selected at install time. The function *returnFeatureList(RequestedFeatures, State)* returns a subset of the *RequestedFeatures* with a binding time that lies before the component state *State* or is bound during the instantiation of the component state *State*. The *alreadyInstantiated(State, Features)* function checks whether a component state instance already exists on the system with the features supplied in *Features*. Not all features need to be specified, since quite often the requirement relationship is with one feature only. An example is when someone requests an e-mail client with the Pop3 protocol. That person at that point does not care whether the IMAP protocol is also included or not.

The *featureSetConsistent(Features, N, State)* function applies the techniques of van der Storm [17]. *featureSetConsistent* checks whether the feature collection *Features*

Algorithm 1 createTree(State, RequestedFeatures)

```
new List RequiresOnce = {}, RequiresOnceCurrentInstantiation = {}
new List RequiresAlways = State.RequiresAlways, new Node N
new FeatureList StateBoundFeatures = returnFeatureBindings(State)
new FeatureList FeaturesUpToNow = returnFeatureList(RequestedFeatures, State)
if alreadyInstantiated(State, StateBoundFeatures ∩ RequestedFeatures) then
    Return N
end if
if StateBoundFeatures ≠ {} then
    if N ≠ FeatureSetConsistent(FeaturesUpToNow, N, State) then
        return N
    end if
    for all CurrentFeature ∈ StateBoundFeatures do
        RequiresOnce = RequiresOnce ∪ CurrentFeature.RequiresOnce
        RequiresAlways = RequiresAlways ∪ CurrentFeature.RequiresAlways
    end for
end if
i = 0, j = 0
for all (RAState, RAFeatureList) ∈ RequiresAlways, i++ do
    N.RAChild[i] = createTree(RAState, RAFeatureList ∪ RequestedFeatures)
end for
for all Instantiation ∈ State.InstantiationList, j++ do
    RequiresOnceCurrentInstantiation = Instantiation.RequiresOnce ∪ RequiresOnce
    for all (ROState, ROFeatureList) ∈ RequiresOnceCurrentInstantiation do
        N.Instantiation[j].ROChild = createTree(ROState, ROFeatureList ∪ RequestedFeatures)
    end for
end for
```

and the feature tree of the component the state *State* can return a consistent feature binding. van der Storms technique returns an empty set if it's impossible to bind these features, an empty list if this feature binding is correct, and a list of unbound features is returned if there are still features left unbound. In the case of an empty set the function *featureSetConsistent(Features, N, State)* returns a dead node, meaning that this branch can never be reached. In the case of an empty list the features can be bound correctly, implying that the component can be instantiated with these feature bindings. Finally, when a set of unbound features is returned, the function checks whether these feature bindings are relevant yet. If they should be bound first, this is returned to the tree. If not, these are simply discarded. The function thus returns a changed node in two cases, and an unchanged node in one case. The first parameter is a list that contains the features that have been required up to now. The reason for that is that otherwise it would be possible to get a conflict, even though this conflict is one that will occur in the future. The algorithm leaves room for improvement here, but the current interface excludes conflicts because all features that can be bound must be bound by the user anyway. The full algorithm is shown in algorithm 1. The main idea is that a new node is created each time this function is called. This node will have two types of children being *require always* children and the more elaborate "Instantiation" children. Each

instantiation will then again have a number of children, which are nodes that are created once the algorithm is called for the state instances that are *required (once)* for that instantiation.

The tree will expand until an instantiation tree is created that shows for each node what component states must be instantiated first before that node can be created. There are some prerequisites, however. Some branches will end because the right features have not been bound. If there is no sequence available due to the fact that insufficient feature bindings have been specified the user will need to add more features. Also, if the current system contains no first component instances a problem is encountered, simply because the tree building cannot end. Another problem is when a component state diagram includes a circular dependency, this will lead to an endless tree. Thus, there cannot be circular dependencies.

3.1 An Example: Instantiating the e-Mail Client

The aim of the following example is to clarify the workings of algorithm 1. In Figure 4 a number of components are shown. The components are an e-Mail client, a Pop3 protocol implementation component, an IMAP protocol implementation component, a binary patch component for the e-Mail client, and a compilation tool that can compile Java and C++ source files. The e-Mail component is the focal point of our example and to instantiate the e-Mail client with certain features, all these other components are required.

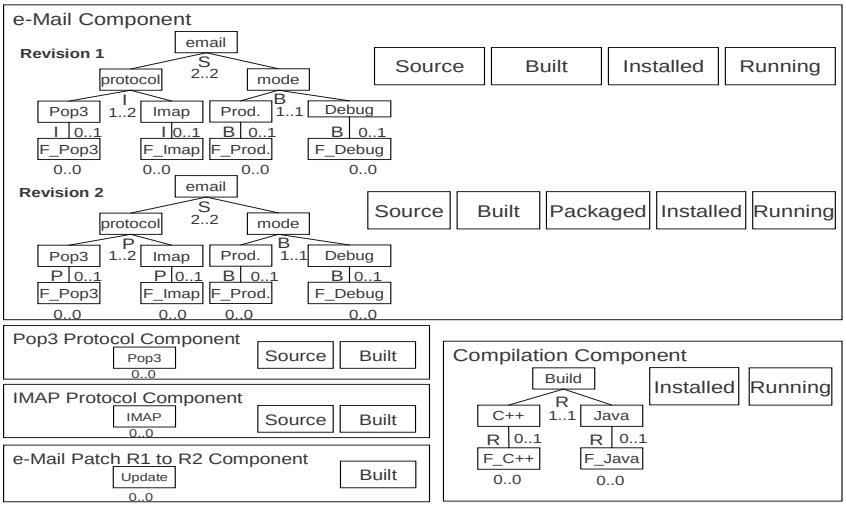

Fig. 4. Component Definition Examples

The following example is based on a system that contains two source state instances (revisions 1 and 2) of the e-Mail client, a source instance of the Pop3 and IMAP protocol implementation, an *installed* instance of the compilation tool, and a *built* state instance

of the update component. The component knowledge in Figure 4 will now be used to create an instantiation tree for the state *running* of the second revision of the e-Mail client component with the feature IMAP.

The example begins with the top node, being “ECR2: Running with IMAP”. In the table for state dependencies is found that the *running* instance cannot exist without the *installed* instance of the second revision of the e-Mail client. The second node, thus becomes the *installed* node. This node requires the *packaged* instance of the e-mail client. At this point the tree building has been straightforward. However, the IMAP feature inclusion now causes there to be two requirements at instantiation time, being the *built* instance of the e-Mail client and the *built* state instance of the IMAP protocol implementation. The *built* state of the second revision of the e-Mail client can be reached in two ways, being through the source of the second revision (ECR2: Source) or through the *built* state of the first revision in combination with the patch. The first instantiation thus depends on the compilation tool with the Java feature and the source code of the second revision. The second instantiation depends on the patch and *built* state of the first revision of the e-Mail client component. The final instantiation tree can be found in figure 5.

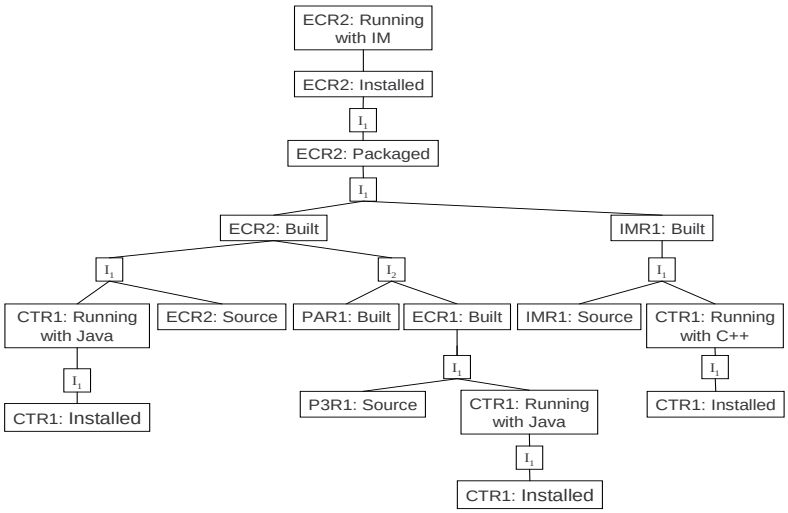

Fig. 5. Instantiation Tree for Component State ECR2: Running With Pop3

To illustrate the following section, some practical uses of the tree are explained here. To begin with, different instantiation sequences can be derived using the instantiation tree. It is possible, for instance, to first satisfy the right subtree of the instantiation of “ECR2: Packaged” and then decide which of the two instantiations must be used for the left side. An example instantiation sequence for “ECR2: Running” thus consists of “CTR1: Running with Java (to build ECR2: built)”, “ECR2: Built”, “CTR1: Running with C++ (to build IMR1: built)”, “ECR2: Packaged”, “ECR2: Installed”, and finally

Ins No.	Instantiation	Component State with Feature(s)
	Requires Once	
1	ecR1: Instantiation Build	ecR1: State Source
1	ecR1: Instantiation Install	ecR1: State Built
1	ecR1: Instantiation Build	btR1: State Running with Java
1	ecR1: Feature Pop3	p3R1: State Built
1	ecR1: Feature IMAP	imR1: State Built
1	ecR2: Instantiation Build	ecR2: State Source
1	ecR2: Instantiation Package	ecR2: State Built
1	ecR2: Instantiation Build	btR1: State Running with Java
1	ecR2: Instantiation Install	ecR2: State Package
1	ecR2: Feature Pop3	p3R1: State Built
1	ecR2: Feature IMAP	imR1: State Built
2	ecR2: Feature Pop3	p3R1: State Built
2	ecR2: Feature IMAP	imR1: State Built
2	ecR2: Instantiation Build	ecR1: State Built
2	ecR2: Instantiation Build	upR1: State Built
1	p3R1: Instantiation Install	p3R1: State Source
1	p3R1: Instantiation Install	btR1: State Running with Java
1	imR1: Instantiation Install	imR1: State Source
1	imR1: Instantiation Install	btR1: State Running with C++

Table 1. Requires Relationships Between Instantiations and Instances

"ECR2: Running w IM". In the following section is demonstrated that it's possible to perform some calculations using the properties and prerequisites for state instantiations.

4 Discussion

The main advantage of the presented models, besides correct and consistent deployment, is the possibility of "what-if" questions. The presented models enable analysis on the deployment of a component *before* the deployment of a component state instance, its dependent features, and state instances. The what-if questions are answered using a number of properties of the instantiation trees. **Excludes relationships** are specific to one state instance instead of components, allowing for components that normally exclude eachother to still reside on a system simultaneauosly. The **tree depth** and instantiation descriptions can be used to evaluate deployment effort. Finally, during the building of the tree, **users can be queried** about what options they have left open, to reduce both the number of possible configurations and to give the user insight into the instantiation order building process.

There is a large advantage of dividing excludes amongst states instead of full components, since it imposes a minor restriction compared to full component exclusion. The example presented in section 3, displays that many components are only required

once during the deployment process of others. This allows for removal (of the patch, for instance) after a certain state has been reached. To support this, the *requiredConcurrently* sets have been introduced as a property of the instantiation tree. These are sets of component state instances that must be present on a system concurrently during the deployment of a component state instance. When two component state instances are in a *requiredConcurrently* set they cannot exclude each other.

Another advantage of the instantiation trees is that the depth of the tree can be used to estimate the effort a deployment costs. The example instantiation tree in figure 5 has two instantiations below the "ECR2: Built" state instance. The branch on the left has less children thus indicating less steps to a final deployment. This tool must be used with care, however. When an instantiation sequence is shorter, that does not necessarily mean it takes less deployment effort. Another indirect advantage of composing these instantiation trees is that during the composition of such a tree, when unbound features are encountered these can be communicated back to the user. The user can then bind the feature to see what the results are of that action. The user can then again remove the feature if it is not to his liking, before actually executing the instantiation sequence.

There are clear links between the methods applied here and the practices of product lifecycle engineering. This research can be seen as a first step in creating a software product lifecycle management system that can facilitate and support the processes of development, release, delivery, and deployment. The following steps in this process are a distribution architecture and a knowledge management framework. The closeness between software product management and product data management is further confirmed by Crnkovic et al. [18].

The main downside of the presented models and methods is that the data entered by the component developers is crucial for the correct functioning of the deployment algorithms, since "garbage in results into garbage out". As discussed in the previous section, however, there are many possibilities for adding information to the software knowledge base. To begin with, automatic feedback can be used to report back to a supplier of a component after the deployment of that component [19]. This feedback can then be used to test for excludes on external products that a software vendor can never discover independently.

Feature descriptions are rather misused here, since they are generally used to describe high-level application requirements and features [13]. The framework, however, uses feature descriptions to model the binding times of features, the requirements of components, and the relationships between the features. We firmly believe that feature descriptions form the solution to many of the complexities related to component configuration and deployment. Feature descriptions can be used to model binding times and show the relationships between features and other required instances. Feature logic and restrictions allow for complex relations to be modelled and simplified, thus enabling algorithms such as shown in algorithm 1. The final question that needs to be answered is whether a software knowledge base really improves the processes of release, delivery, and deployment. There are four facts that point to that direction.

– **Product data management** improves the release and delivery of other products [20]. Since software production processes share many similarities with other pro-

duction processes [18], software release, delivery, and deployment can also be improved.
- Since the current trend in the software market is **mass customisation**, much of the information gathered in the development stages of the product can be reused at later stages during implementation at the customer and customisation phases.
- **Case studies** [19] [15] show that centrally storing knowledge leads to reduced delivery effort.
- The ability to present **"what-if" questions** to a local software knowledge base that is connected to multiple component sources can increase the reliability of the component deployment process. These questions enable a system manager to more explicitly predict what changes can be made to a system and what features can be provided within a certain configuration of components.

These facts show that managing knowledge about software explicitly and making it available to all involved parties improves release, delivery, and deployment processes.

5 Future Work and Conclusions

Currently the models have been implemented in Prolog, however, to fully apply the models in an industrial setting, a new implementation technology must be chosen with the support of cross platform compilers. We are hoping to apply the tools in a practical situation in the context of a case study. To avoid reinventing the wheel and to standardize the models, the applicability and feasibility of the OMG specifications for reusable assets [21] and IT portfolio management [22] must be evaluated for the current models. The current algorithm blindly builds trees that can explode in complexity quite quickly. There are many opportunities for reuse and further research is required in that area to reduce the complexity of these instantiation trees. Also, the representation of the software component knowledge must be compared to other methods [8] to store and share software component knowledge.

This paper establishes a relationship between component state models and feature descriptions enabling reasoning about the deployment of a component or component set without actually deploying the software. An algorithm is provided that can build instantiation instantiation trees to determine the deployment order of components. These trees can be used to answer "what-if" questions about the deployment of a component or set of components. The research has shown that both feature descriptions and a component state model can be used to create a software knowledge base that stores information about components and their context. The knowledge used to achieve this, however, relies on information provided by developers and users of the components.

References

1. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc. (2002)
2. Hall, R.S., Heimbigner, D., Wolf, A.L.: A cooperative approach to support software deployment using the software dock. In: ICSE. (1999)

3. Jaring, M., Bosch, J.: Representing variability in software product lines: A case study. In: Second Product Line Conference (SPLC-2),San Diego CA, August 19-22. (2002)
4. Bosch, J., Högström, M.: Product instantiation in software product lines: A case study. In: LNCS. Volume 2177. (2001) 147
5. Jansen, S., Brinkkemper, S., Ballintijn, G.: A process framework and typology for software product updaters. In: Ninth European Conference on Software Maintenance and Reengineering, IEEE (2005) 265–274
6. Dolstra, E., Visser, E., de Jonge, M.: Imposing a memory management discipline on software deployment. In: IEEE Workshop on Software Engineering (ICSE'04), IEEE (2004)
7. Hnetynka, P.: Component model for unified deployment of distributed component-based software. In: Tech. Report No. 2004/4, Charles University, Prague. (2004)
8. Hall, R., Heimbigner, D., Wolf, A.: Specifying the deployable software description format in xml. In: Technical Report CU-SERL-207-99, University of Colorado SERL. (1999)
9. Object Management Group: Deployment and Configuration of Component-based Distributed Applications Specification. In: OMG document ptc03-07-08. (2003)
10. Dolstra, E., Florijn, G., de Jonge, M., Visser, E.: Capturing timeline variability with transparent configuration environments. In Bosch, J., Knauber, P., eds.: IEEE Workshop on Software Variability Management (SVM'03), Portland, Oregon, IEEE (2003)
11. Carzaniga, A., Fuggetta, A., Hall, R., van der Hoek, A., Heimbigner, D., Wolf, A.: A characterization framework for software deployment technologies. In: Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado. (1998)
12. Dolstra, E., de Jonge, M., Visser, E.: Nix: A safe and policy-free system for software deployment. In Damon, L., ed.: 18th Large Installation System Administration Conference (LISA '04), Atlanta, Georgia, USA, USENIX (2004) 79–92
13. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis feasibility study. Technical Report CMU/SEI-90-TR-21, Pittsburgh, PA (1990)
14. Clegg, S.: Evolution in extensible component-based systems. In: Master Thesis. (2003)
15. Jansen, S.: Software Release and Deployment at Planon: a case study report. In: Technical Report CWI, SEN-E0504. (2005)
16. Bontemps, Y., Heymans, P., Schobbens, P.Y., Trigaux, J.C.: Semantics of feature diagrams. In Männistö, T., Bosch, J., eds.: Proc. of Workshop on Software Variability Management for Product Derivation (Towards Tool Support), Boston (2004)
17. van der Storm, T.: Variability and component composition. In: Software Reuse: Methods, Techniques and Tools: 8th International Conference (ICSR-8). LNCS, Springer (2004)
18. Ivica Crnkovic, U.A., Dahlqvist, A.P.: Implementing and integrating product data management and software configuration management, Artech House Publishers (2003)
19. Jansen, S., Brinkkemper, S., Ballintijn, G., van Nieuwland, A.: Integrated development and maintenance of software products to support efficient updating of customer configurations: A case study in mass market erp software. In: Proceedings of the 21st International Conference on Software Maintenance, IEEE (2005)
20. Helms, R.W.: Product data management as enabler for concurent engineering, ph.d. dissertation. In: Eindhoven University of Technology press. (2002)
21. Object Management Group: Reusable Asset Specification. (2004)
22. Object Management Group: IT Portfolio Management Specification. (2004)

Acknowledgements

The authors thank Vedran Bilanovic and the Trace team for their many inspiring ideas that contributed to this paper. Also, the anonymous reviewers did a great job in reviewing this paper and giving highly constructive criticism. Finally, The authors thank Tijs van der Storm for providing the Prolog implementation of FDL to BDD conversion and partial evaluation functions that made the prototype implementation so much quicker.