ALLEVIATING THE RELEASE AND DEPLOYMENT EFFORT OF PRODUCT SOFTWARE BY EXPLICITLY MANAGING COMPONENT KNOWLEDGE*

Slinger Jansen
Utrecht University
slinger.jansen@cs.uu.nl

Abstract

Product software release, delivery, and deployment are complex tasks for software vendors, system managers, and end-users of a software system. This position paper proposes a framework that manages software knowledge during the lifecycle of a software product, with special focus on the release, delivery, and deployment processes of software components. The first pillar of the framework manages components and their associated information on a local system. The second pillar of the framework enables knowledge sharing amongst system nodes in a network. Central to the framework is the underlying knowledge base, a distributed knowledge base that stores knowledge about software systems and its components.

Keywords: *Product software, software product lifecycle management, release, delivery, deployment*

1 The Complexities of Release and Deployment

A software product is a packaged configuration of components or services with auxiliary materials, which is released for and traded in a specific market. Product software encompasses information systems, out-of-the-box applications, B2B software, and mass customisation software [1]. The development, production, release, (re)sale, implementation, deployment, maintenance, and support of these products generally take more time and cost more resources than planned due to the complexities of these processes.

Some of these complexities are directly related to the software vendor and the complex processes of software development. For the vendor there are many customers to serve, all of whom might require their own revision or variant of the application. Furthermore, the application itself 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 meet the changing needs of the customers. As a consequence, the development, release, delivery, and deployment of these applications takes a significant amount of effort and is frequently error prone. The tasks of adding new features, adding support for new hardware devices and platforms, system tuning, and defect fixing all become exceedingly difficult as a system grows with age.

Next to development the release, distribution, maintenance, and deployment processes of a component can be elaborate. First, the software vendor can have different sales channels, such as (re)distribution partners, who wish to make their own adjustments and customisations to the software product. Secondly, the context in which the software is deployed and used can be a complex network architecture, with different privileges and licensing scenarios. Thirdly, customers might first want to test updates and software components before installing the software onto their network nodes. Finally, the maintenance of a (running) system can be complex due to consistency relationships between components, datamodels that need to be updated, or users that rely on continuous service.

1.1 Current Solutions

Many different isolated solutions that attempt to handle the afore mentioned complexities exist, yet not one provides full coverage of the software release, delivery, update [2], and deployment processes. Also, the processes of release and deployment are complex fields for which no integrated solutions exist that both manage the product and artifacts throughout these states [3]. Thirdly, there are many different component architectures that work well for specific technologies [4] but do not allow for integration with other component technologies without taking tedious manual steps.

There are academic frameworks that manage the

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

component lifecycle of generic components, such as Nix [5], the Software Dock [6], and Sofa [7]. These systems have downsides however. For instance, Nix is a technology based on an environment where the source code always must be available and that can guarantee consistency between components and allows for concurrent installations of components. Nix, probably the most advanced of the three, requires a system manager to rebuild a component X if a component on which X depends is altered. Also, as of yet Nix only minimally manages variation due to changed configuration settings. The Software Dock can be used to deploy software using the XML-based deployable software description [8]. The Software Dock does not support the lifecycle of a software component, since it does not take into account the different variation points in a software component. It focusses only on the deployment process of software, including such states as *activate*. Finally, SOFA is a component framework that uses the OMG Deployment and Configuration specification [9], and is focused on the deployment of generic components. Sofa, just like Nix, assumes a very simple lifecycle with four states being *source*, *built*, *deployed*, and *running*. Of these three advanced [10] component frameworks, only Nix focusses on variable features provided by different instantiations of components, and only Nix discusses the opportunities for a transparent configuration environment [11], where all configuration options, binding information, and configuration settings are externalised and managed separate from the component artifacts.

The fact that these tools only partially alleviate the release, delivery, and deployment processes of product software vendors shows that more research is required in the area of product software lifecycle management. Earlier research shows that many aspects of the processes of development, production, release, re(sale), implementation, deployment, maintenance, and support of a software product can be integrated [12]. Each of these processes encompasses managing or changing the software artifacts that are part of the software product or of the peripheral tools and utilities used to create the software product.

1.2 Four Complexities

The four complexities on which this paper focuses are all related to the fact that components are evolving variable entities. These four complexities all affect the processes of development, release, delivery, and deployment:

1. A component is constantly evolving and therefor **multiple revisions** will be available at any time to customers.
2. A component can be instantiated in **different variations**, depending on external knowledge such as the configuration settings or environment settings of a system [13] and the restrictions imposed by the environment in which the component is deployed.
3. Component variabilities can be **bound at different times**, making them hard to manage, since components can be distributed at different binding times, such as a source or binary distribution.
4. There are **different component types** (J2EE, .Net, databases), that need to communicate with eachother to guarantee a consistent and complete system.

A software component will commonly be released periodically, thus creating different available versions of that component. A well-known example is Firefox, currently available as version 1.0.4. On the Mozilla website it is possible to download earlier versions as well. Firefox is available in different variations, such as the Dutch Windows version or the Albanian Mac Os X version. Variability can be bound at different times, such as the location for installation of Firefox is bound at installation time, whereas the e-mail account settings can be modified at run-time. Finally, the firefox component can communicate with different types of plug-ins, such as a html2ps converters or an ML interpreter. Clearly the interfaces between Firefox and these components are going to be different.

This paper proposes a framework that handles these four complexities in the area of product software lifecycle management by integrating knowledge and sharing this knowledge amongst different processes. First, this paper presents a component system management model that supports the deployment of a component in different revisions and variants, and still guarantee consistency and completeness of component configurations. The model is based on the central storage of restrictions and knowledge about component features and the system, thus allowing all components to share such information for correct release, testing, deployment, and activation of software. Secondly, a communication architecture is introduced that abstracts from the customer vendor model and can be applied in a wide range of scenarios to share and distribute components independent of their environment, in their different appearances, between two nodes in a network. Both the component lifecycle and the communications concepts are centered around the notion of a software knowledge base (SKB)

that stores and shares knowledge about the software components on a node in a network.

Beneficiars of this research are those who intend to build tools that manage the development, release, and deployment of software components. Indirectly this affects not only software vendors, but also system managers and end-users of product software. If tools become available based on the presented framework software vendors do not need build their own product installers and updaters. Also, such tools can increase reliability of the software vendor's products and updates, since feedback is shared automatically. System managers profit because such tools can cause a reduction in testing, deployment, support, and network node management time. Finally, tools that are based on the research provide end-users with the ability to pose "what-if" questions to the software knowledge base resulting in less problems caused by inconsistency and incompleteness due to sharing of information amongst a large group of end-users.

The rest of this paper is structured as follows. Section 2 describes how the four complexities can be modeled on a system management level, by a framework that manages meta-information about any type of component. Section 3 describes how knowledge can be shared amongst different nodes in a network, to provide users with the ability to transfer a software component in any state to a different system. Section 4 describes how knowledge can be stored to support both the component framework for system management and the component framework for component transfer between network nodes. This section also focusses on the operations that are necessary to populate and use such a distributed software knowledge base. We end this paper with a discussion of the proposed framework and some conclusions.

2 Deployment and Systems Management

Software components are units of independent production, acquisition, and deployment [14]. There are many different component technologies, such as Corba, J2EE, and Koala [15], all with their own lifecycle and associated management tools. For each of these technologies different management tools have been implemented to build, test, evaluate, and deploy these components. Such tools and techniques usually implement only parts of the phases of the component lifecycle, however, and allow only for very specific integration of different component technologies. On the other hand, large systems often consist of different types of components that all need to be managed, and thus require a combination of complex component technologies and interfaces. Also, depending on the functionality a component provides, components require other software components, hardware components, and configuration knowledge. Such knowledge is quite often stored in different places and not apparent from the deployment instructions, leading to frustrating and time consuming deployment processes.

The situation described above calls for a generic framework that can handle the four complexities that are introduced by the use of variable evolving components that can be instantiated in different revisions and appearances on different systems. Also, none of the systems mentioned in Section 1 include a central knowledge base that stores the variables that initialize the different varieties of component instances, nor a categorization of such knowledge. One other reason for initiating this research is that the area of software component updaters [2] deserves more attention, since many component vendors struggle with the automation of their release and deployment processes [16].

2.1 Proposed Framework

When released, delivered, and deployed on a customers hardware system, a component goes through different states, such as *source*, *built*, *deployed*, and *running*. These examples are not exhaustive since multiple states might be required to describe different binding times. Contrary to other component models such as Nix and Sofa, which assume a four state model, the framework presented allows the lifecycle model to be extended. Such extensions function as an update to the component lifecycle model and allow for more complex states such as *updatedSource*, *packaged*, and *activated*.

For the framework the concepts state, state instance, and state instantiation are used. A component state describes a shape or form in which a component can reside on a system. A state instance is an actual appearance of such a state. An instantiation of a state refers to the actual creation of an instance. For example, a source distribution of an e-mail client Godzilla Mail can be available in a source instance. The source instance is a requirement for the built instantiation of Godzilla Mail, since the source code is required only during the building process and not thereafter. In this example both source and built are (hypothetical) states. Once referring to a directory on a filesystem that stores the built artifacts, we speak of an instance.

The proposed component lifecycle framework [17] enables a software vendor to express the component lifecycle. The component lifecycle consists of states as described above. A simple component could have three states, being *source*, *built*, and *running*. Once a component resides on a system within a certain state, the occupant is referred to as a component state instance, such as "the source files installed in directory X". The instantiation of an instance can only be performed once the conditions for that instantiation are met. Instantiations are transfer steps between states, such as building, running, or installing a component. These conditions enable the software vendor to check the requirements for a component instantiation and to perform steps to satisfy such requirements.

The component state description defines the properties of a component state instance, i.e., the features the component state can supply (using a feature description language [18]), the other component states it requires, and the revision number of the state. It also describes the requirements that need to be met to instantiate the component state, the features and resources a component state instance can provide to other components, and the restrictions that hold for all component state instances of the state. To create a component state instance, both the requirements of the component state and of the instantiation need to be met. Once a state has been instantiated, other instances can use the features that are provided by this instance. For example, a database tool can provide connections once it is running.

To instantiate a state, a previous instance is required, i.e., to go from the *source* state to the *built* state an instance of the source files is required. Instances are unique since each instance binds variability and thus the instances might provide different features. There may be limits to the number of instances depending on a previous instance. An example is when an executable can only be executed twice simultaneaously on one hardware system. Instances can have complex relationships with other instances. An example of such a relationship is when an instance alters a previous revision through a patch, therby removing the previous instance.

Figure 1 shows a UML diagram of the meta model of the framework, where components have a lifecycle that consists of (possibly instantiated) states. The picture can be illustrated with an example of the imaginary component Godzilla mail. Godzilla mail has a lifecycle consisting of a *source*, *built*, *installed*, and *running* state. A user can have different state instances deployed on her computer, she could have an *installed* instance and two *source* instances. Once she wishes to run Godzilla, she

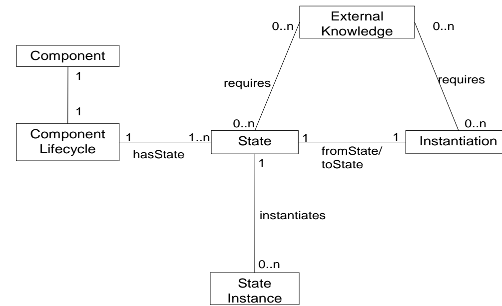

Figure 1: UML diagram of States, Instances, Instantiations, and Knowledge

can instantiate the running state. If successful, she will have four instances on her system, an *installed* instance, two *source* instances, and a *running* instance.

2.2 Complexities

The four complexities presented earlier are all reduced by the proposed framework. Issues with **binding times** can be solved using the proposed framework. Since the relationships between feature bindings in the lifecycle model are known, it is possible to build a dependency tree of states. As such, problems can be avoided by using such knowledge when early instances are created. An example of using this information is when a user builds the source of a component and can choose between a production or test build. The framework can then be used to inform the user that, for example, the test build cannot provide any data import functionalities during runtime.

The problem of **different (possibly incompatible) revisions** of components is handled by feature logic in the framework. At present no decision has been made whether feature relationships will be intensional or extensional. Early prototypes implementing the conceptual framework, however, assume feature relationships have been defined extensional and the aim is to include both in future experimentational tools. The framework introduces two advantages to the use of differently revisioned instances. First, due to the fact that there are often different ways to reach a certain instance, all ways can be evaluated to obtain more information about the instantiation. Secondly, due to the fact that dependencies are feature-based instead of component-based, the feature descriptions can be used as an interface mecha-

nism. A precondition to enable feature descriptions as an interface, which isn't the case for the prototype, is that features no longer need to have unique names.

Customers have **different configurations** creating inhospitable environments for components. This implies that configuration settings need to be managed explicitly. Since the presented component framework attempts to encompass all kinds of components, including license pools, configuration settings knowledge, hardware resources, and many other kinds of knowledge, configuration settings can also be managed. Explicit management brings us a step closer to a transparent configuration environment [11]. The management of such knowledge allows complete modelling of the lifecycle of a component system, and therefor what-if questions can be posed to the component knowledge base.

Finally, the issue of dealing with **different types of components** that need to communicate is approached by creating a model that manages the meta-information of each component. When all components are managed as equal entities, it becomes easier to specifically address a component independent of its type.

2.3 Running Example

A developer is working on the next release for Godzilla Mail with support for POP3 and IMAP. The developer is using the Palm e-mail reader at present, which only supports POP3. The developer might wish to test Godzilla on her platform, however, there is an exclude relationship between the two readers. The proposed framework has many advantages in this example. To begin with, exclude relationships are established per lifecycle state, so it could be that the two only exclude eachother while running, and Palm does not need to be removed to test Godzilla. The planned interface will automatically attempt to build, install, and run Godzilla, based on the availability of the source instance of Godzilla Mail. Also, since Godzilla and Palm exclude eachother, the framework-based tool can propose to remove Palm (or stop it, if there is only the concurrently running exclusion). Since all component knowledge, including configuration settings such as the e-mail account settings, is externalized, Godzilla Mail can simply reuse the account settings originally used by Palm. If other components are required for the instantiation of Godzilla, such as build tools and make scripts, these will first be downloaded, installed, or run. One feature the framework can support is the user requesting a generic feature instead of an application, such as the developer invoking Godzilla by stating that she needs an IMAP e-mail reader, instead of stating Godzilla explicitly.

3 Delivery and Sharing Knowledge

The delivery and distribution of software products and updates to customer systems is highly complex. The software can be deployed in many different network architectures and settings, each imposing different constraints on the possible configurations of the software product. Secondly, there are different deployment scenarios, such as a network manager who wants to update all its clients at once or a system manager who gradually wants to update all nodes in a network over a longer period of time. Also, there are different users with different privileges and roles in a network, where each individual can introduce new feature bindings and constraints onto the software product. Finally, there are different media to distribute the software through, such as a CD-ROM, an internet site, or a network connection through which the software is pushed or pulled to the customer.

To illustrate some of these complexities we take the distribution chain of a client-server product from a software vendor to a customer. It is not unusual that a component first goes through several third-parties and resellers before a component actually reaches the customer. On the developer site the product is developed without any bound variability. As soon as the developer distributes to a reseller some variabilities, such as the language, can be bound. The reseller will again bind some other variabilities, by adding a logo to the product, a customised manual, or a database layer. Once the reseller sells the product to a customer, the product will first be deployed on a componany server where a multitude of variabilities will be bound. Once the client product is distributed to the customers systems in the company, only a mere subset of the early variabilities need to still be bound. The key to this example is the sharing and hiding of different kinds of information to different levels of distribution. This information can, for example, encompass software artifacts, feature diagrams, and configuration information. Nodes in a network should share, hide, and manage this information each time they communicate with another node.

The interaction between nodes in a network about components consists of the same steps at each transfer step. Each step requires a receiving network node and a sending network node (see also Figure 2). The interaction starts with the sending node (optionally) binding some of the variabilities of the product. Once the product or update is ready to be sent (released), the sending node notifies the receiving node of the availibility of a product or update. Depending on whether the policy is push or pull, the receiving node can decide to down-

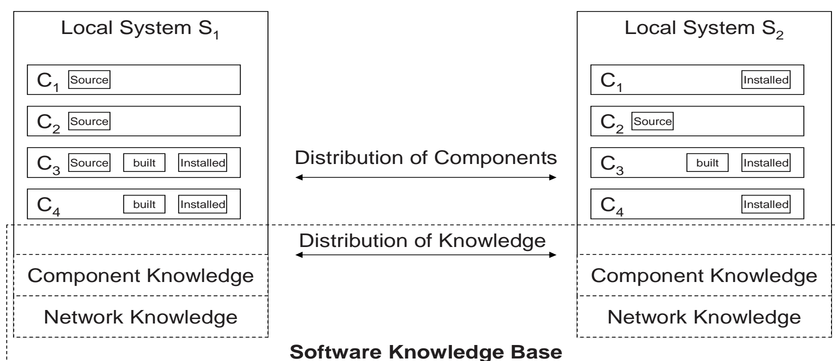

Figure 2: Knowledge Sharing between two Network Nodes

load the product, or the sending node can decide to upload the product or update. Once the product has been downloaded it still needs to be installed by the receiving node. During the installation, activation, and running processes feedback can be sent back to the sending node. The process described here is closely related to the software update process [2] that has been created by comparing and categorizing different product update techniques. The process is displayed in Figure 3. If we replace software vendor by sending node and customer by receiving node in the figure, the model becomes applicable to the transfer of any component between any two "equal" parties.

3.1 Communication Framework

The model displayed in Figure 3 abstracts from the type of media used to interact about the software products, leaving different ways to send back and forth software products, components, and updates. An early prototype uses RSS[1] feeds to distribute news, and the BitTorrent[2] protocol to distribute components across a network. Feedback reports from installs and runs are sent to the sending party by e-mail.

[1]Really Simple Syndication
[2]http://www.bittorrent.com/

The advantage of RSS feeds is that the reader of the feed can decide what to do depending on the policy of the receiving party. The updates can be installed straight away, simulating push behavior, or after an amount of time. Also, the receiver can decide whether to prompt the user about the component installation or update. A downside of RSS feeds is that news items cannot be sent to nodes in the network specifically. The BitTorrent protocol assists in distributing the load on a distribution server, since nodes can download parts from other nodes, that downloaded the component at an earlier stage. This leaves a minimum pressure on the distribution server that would otherwise send the component or update to all nodes.

3.2 Running Example

Looking back on our previous example, the developer might wish to publish Godzilla Mail onto a release server, where releases of the Godzilla Mail client are stored. The server, which is also running the distribution tools based on our framework, can easily be accessed (by the developer only) to deploy components for future distribution. The developer can specify what knowledge can be stored on and retrieved from the release server. The components are thus uploaded to the distribution server, including knowledge about (in)compatibilities.

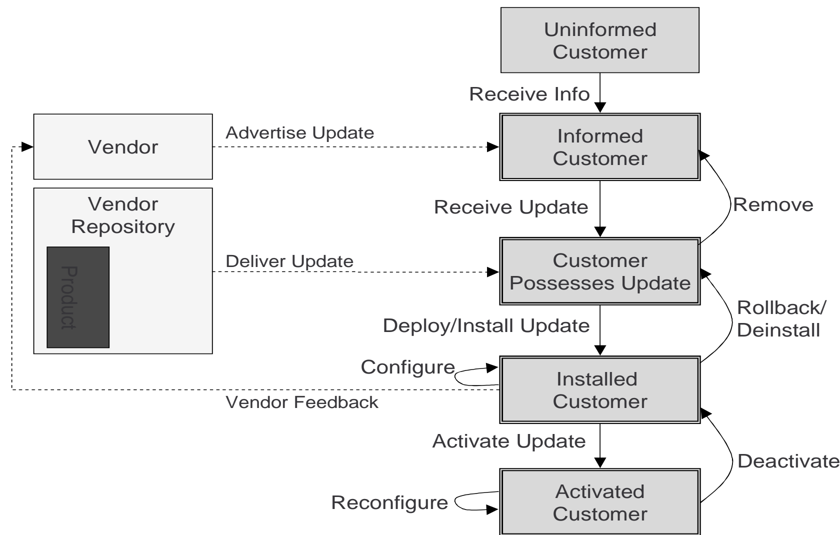

Figure 3: Product Software Update Process

The developer can decide whether she wants a source, built, or combined distribution on the server so the framework can be used by developers of open source and commercial software alike. Once deployed on a customer system, the framework can be used to communicate incompatibilities and bugs back to the software developer. Such incompatibilities can be found automatically by integrating the debug functionalities of the software product with the framework. Bugs could be turned into bug reports automatically, enabling an automatic feedback cycle from customers.

4 SKB Research Challenges

The software knowledge base that allows for management of software artefacts and meta-information can be compared to a product data management system for a conventional product. Research shows that software configuration management systems are closely related to product data management systems [19]. The software knowledge base proposed here can be seen as an extension to a software configuration management system, in that it manages meta-information about software artifacts and performs operations (in cooperation with tools that support the development, release, deliver, and deployment processes) on the software configuration management system. Many of the concepts introduced here are based on the software knowledge base designed by Meyer [20].

The SKB stores knowledge that might be required during the delivery of a component state instance and during the state instantiation of a state. With respect to components the SKB at a site will store what instances of component states are available at present, the dependencies between states, whether instances can be shared or not, and the dependencies between state instances. With respect to the system the SKB stores information on hardware resources, operating system, user profiles, network location and its network capabilities. With respect to the users, the SKB stores licensing information, user settings, user preferences, and permissions.

4.1 Extraction

One of the main questions remains where the information should come from. Some of the dependency information can be automatically extracted [21], however, most of the information, such as lifecycle information for a component, simply needs to be input by a developer, a development environment (such as Eclipse[3]), or a party that is well accustomed to a product. One appealing function of the framework is that it enables knowl-

[3] http://www.eclipse.org

edge sharing about components. Incompatibilities can be shared, since they can be sent back to the knowledge (and component) provider. Such compatibility knowledge, which is normally gathered by the software vendor through testing, is usually incomplete, due to the scale and complexity of possible component configurations. The sharing of knowledge, however, allows for a much larger testbed since feedback is generated at the customer, much like Microsoft does when sending an error report after a problem has occured. Information about software components can also be extracted from source code, by performing dependency analysis for instance, and is a relatively cheap method to derive simple dependencies. However, many dependencies are implicit, and can often only be extracted at runtime.

4.2 Representation

The UML diagram in figure 1 attempts to categorize and further define a structure for a software knowledge base in which to store the knowledge. This attempt is based on a long history of different manners to identify and categorize knowledge. The Object Management Group[4] attempts to store software knowledge through its Reusable Asset Specification [22] and the Software Portfolio Management specification [23]. Microsoft has, by introducing the open source deployment toolset and format WiX[5], also proposed a format to manage component deployment. Other languages used for component description and deployment are evaluated in [3]. The major downsides of most of the component description and deployment languages are:

- **Extensibility -** The languages and description formats are usually not extensible enough to incorporate information such as feature descriptions.
- **Scalability -** More often than not the languages only apply to very specific types of components, thus not allowing establishment of relationships amongst different components.
- **Problem specific -** The languages have often been designed to store only necessary information for one specific process, such as deployment or automated testing. The description languages and schema should rather allow for a generic description of a (software) system to enable relationships amongst different levels of complexity.

One aim of the research is to identify and categorize the knowledge items that are relevant to all stages of software development, release, and deployment. For knowledge representation we are looking at the formats available and some state-of-the-practice technologies that can be reused for the central knowledge base we propose. However, this is left for future work.

4.3 Inference

Through inference many of the applications of the software knowledge base can be enhanced. To begin with, feature descriptions can be used to supply a user of the software system with requested features or a report on why a certain feature combination is not possible. The same holds for component states and instances that are linked to artifact knowledge that can be used to establish the validity of a configuration. Such information can also be used to remove the unused artifacts once an instantiation is removed from a system.

The most important feature of the software knowledge base remains the ability to answer "what-if" questions. "What-if" questions allow the user of the knowledge base to see the impact of a decision with respect to components or configuration settings. Such knowledge can be used to inform or warn the system manager of incompatibilities, resource problems (not enough diskspace), and many other delivery and deployment issues.

4.4 Application

Reusing component knowledge allows for a myriad of applications of which most are part of our future work. The framework can be used to update a software configuration and with the right communication protocols and procedures can even become a generic release, deployment, and update tool, that supports all features (instead of a subset) described in [2]. The envisioned deployment tool should also be able to perform automatic deployment within a multitude of scenarios, such as a software product that consists of client and server components and must be deployed automatically on different nodes in a network.

There are two other areas of interest that belong to our future research. First, there are security and licensing issues that come into play when dealing with component and configuration settings knowledge distribution. The distribution framework presented allows network nodes to share component instances more easily, thus leaving the door to piracy wide open. Also, when sharing configuration settings, such as e-mail account information, amongst different nodes in a network the information

[4] http://www.omg.org

[5] Windows Installer XML, http://sourceforge.net/projects/wix

can be misused. In the future, the framework presented will focus on these areas as well.

The other area of interest is the management of distribution and deployment policies. Some networks and components, for instance, will allow for automatic updates of virus definitions, whereas the operating system update may not be installed automatically before the update is tested. Such policies, greatly affecting usage of network resources, must be explicitly managed and adhered to by the framework.

5 Discussion

The raison d'être for this research consists of two parts. First, at present there do not seem to be any deployment tools out there that cover all states of the lifecycle of a software product. Also, it seems as though many developers try, but do not succeed, in delivering a software component management system that can fully and correctly support the delivery and deployment of different types of software components.

The first question that is raised by this unavailability of complete software deployment tools is whether there is a need for such tools and frameworks. The fact that each software vendor is required to produce their own specific update tool to deliver updated functionality and content to their own products suggests a positive answer, since much of their work consists of reinventing the wheel. The large amount of commercial and non commercial delivery and deployment tools suggests that there is no ideal software deployment tool yet, where there is a need for a generic software deployment tool.

The second question raises concerns what properties are lacking in current component delivery and deployment tools. Even though many of the available tools cover large parts of the software release, delivery, and deployment processes, they are far from perfect. Not only do they lack coverage of certain processes (often configuration and interaction with external components are neglected), but usually they are specific to one product, technology, operating system, or deployment format. The research performed here abstracts attempts to create a framework that can be used as a basis for all tools and applications that have relationships with other components, hardware, and user settings. One important feature that lacks from current technologies and does not receive enough focus is the representation of variability to the user. Our use of feature diagrams can solve this problem [24].

Previous research and case studies show that companies can manage their release, delivery, and deployment processes quite well with their own specific tools or simply by manually sending their software by mail or consultant [16] [25]. Some simplifications, such as when a consultant is sent to the customer to deploy the update there, may even generate more business. The question thus is whether software vendors have any specific need for frameworks that can improve the quality of their software product. However, the research performed here aims to simplify the processes, thereby alleviating the resources currently going into delivery and deployment, such as the burning of CD-ROMs and mailing updates to customers.

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. First, product data management improves the release and delivery of conventional products [26]. Since software production processes share many similarities with conventional production processes [19], software release, delivery, and deployment can also be improved. The widespread use of configuration management systems for software development already proves that some techniques from conventional production processes can easily be applied for software development. Secondly, 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. Thirdly, a number of case studies [12] [16] show that centrally storing knowledge leads to reduced delivery effort. Finally, the ability to ask "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.

6 Contribution and Conclusions

The contribution of this paper is threefold. To begin with a framework is introduced that manages components on an evolving component-based system, guaranteeing consistency relationships amongst, and enabling "what-if" queries about component configurations on a system. Secondly, a model for distribution of such components in any state to any node in a network is proposed, that can improve release and delivery of software components. Thirdly, a central software knowledge base

is discussed, and we elaborate on the issues that make the software knowledge base a vital element in software development, release, and delivery.

References

[1] L. Xu and S. Brinkkemper, "Concepts of product software: Paving the road for urgently needed research," 2005.

[2] S. Jansen, S. Brinkkemper, and G. Ballintijn, "A process framework and typology for software product updaters," in *European Conference on Software Maintenance and Reuse (CSMR '05)*. IEEE, 2005.

[3] R. S. Hall, D. Heimbigner, and A. Wolf, "Evaluating software deployment languages and schema," in *ICSM*, 1998, pp. 177–187.

[4] S. Clegg, "Evolution in extensible component-based systems," 2003.

[5] E. Dolstra, E. Visser, and M. de Jonge, "Imposing a memory management discipline on software deployment," in *IEEE Workshop on Software Engineering (ICSE'04)*. IEEE, 2004.

[6] R. S. Hall, D. Heimbigner, and A. L. Wolf, "A cooperative approach to support software deployment using the software dock," in *International Conference on Software Engineering*, 1999, pp. 174–183.

[7] P. Hnetynka, "Component model for unified deployment of distributed component-based software," *Tech. Report No. 2004/4, Dep. of SW Engineering, Charles University, Prague*, June 2004.

[8] R. Hall, D. Heimbigner, and A. Wolf, "Specifying the deployable software description format in xml," 1999.

[9] Object Management Group, "Deployment and Configuration of Component-based Distributed Applications Specification," *OMG document ptc03-07-08*, June 2003.

[10] A. Carzaniga, A. Fuggetta, R. Hall, A. van der Hoek, D. Heimbigner, and A. Wolf, "A characterization framework for software deployment technologies," 1998.

[11] E. Dolstra, G. Florijn, M. de Jonge, and E. Visser, "Capturing timeline variability with transparent configuration environments," in *IEEE Workshop on Software Variability Management (SVM'03)*, J. Bosch and P. Knauber, Eds. Portland, Oregon: IEEE, May 2003.

[12] S. Jansen, G. Ballintijn, S. Brinkkemper, and A. van Nieuwland, "Integrated Development and Maintenance of Software Products to Support Efficient Updating of Customer Configurations: A Case Study in Mass Market ERP Software," in *Technical Report UU, sumbmitted for publication*, 2005.

[13] J. V. Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*. Washington, DC, USA: IEEE Computer Society, 2001.

[14] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[15] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, Mar. 2000.

[16] S. Jansen, "Software Release and Deployment at Planon: a case study report," in *Technical Report CWI*, 2005.

[17] ——, "Feature Driven Component Lifecycle Management," in *Technical Report UU (to be)*, 2005.

[18] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," SEI, CMU, Pittsburgh, PA, Tech. Rep. CMU/SEI-90-TR-21, Nov. 1990.

[19] U. A. Ivica Crnkovic and A. P. Dahlqvist, "Implementing and integrating product data management and software configuration management." Artech House Publishers, 2003.

[20] B. Meyer, "The software knowledge base," in *Proceedings of the 8th international conference on Software engineering*. IEEE Computer Society Press, 1985, pp. 158–165.

[21] J. Dunagan, R. Roussev, B. Daniels, A. Johnson, C. Verbowski, and Y.-M. Wang, "Towards a self-managing software patching process using black-box persistent-state manifests," in *International Conference on Autonomic Computing (ICAC'04)*. IEEE, 2004.

[22] Object Management Group, "Reusable Asset Specification," *OMG Final Adopted Specification*, June 2004.

[23] ——, "IT Portfolio Management Specification," *OMG Final Adopted Specification*, November 2004.

[24] Y. Bontemps, P. Heymans, P.-Y. Schobbens, and J.-C. Trigaux, "Semantics of feature diagrams," in *Proc. of Workshop on Software Variability Management for Product Derivation (Towards Tool Support)*, T. Männistö and J. Bosch, Eds., Boston, August 2004.

[25] G. Ballintijn, "Software Release and Deployment at Chipsoft: a case study report," in *Technical Report CWI*, 2005.

[26] R. W. Helms, "Product data management as enabler for concurent engineering, ph.d. dissertation," in *Eindhoven University of Technology press*, 2002.

Acknowledgements

I would like to thank Remko Helms, Tijs van der Storm, Gerco Ballintijn, and Sjaak Brinkkemper for carefully reviewing this paper. Furthermore I would like to acknowledge Vedran Bilanovic for our inspirational discussions.