A Process Model and Typology for Software Product Updaters

Slinger Jansen
Centre for Mathematics and Computer Science
Amsterdam, The Netherlands
Email: r.l.jansen@cwi.nl

Sjaak Brinkkemper
Institute of Information and Computing Sciences
Utrecht University
Utrecht, The Netherlands
Email: s.brinkkemper@cs.uu.nl

Gerco Ballintijn
Centre for Mathematics and Computer Science
Amsterdam, The Netherlands
Email: g.ballintijn@cwi.nl

***Abstract*— Product software is constantly evolving through extensions, maintenance, changing requirements, changes in configuration settings, and changing licensing information. Managing evolution of released and deployed product software is a complex and often underestimated problem that has been the cause of many difficulties for both software vendors and customers. This paper presents a process model and typology to characterize techniques that support product software update methods. Also, this paper assesses and surveys a variety of existing techniques against the characterisation framework and lists unsolved problems related to software product updaters.**

1. Product Updating

Managing evolving software is a complex task for software distributors and vendors. Moreover, maintaining a large software system, such as a business ERP application, can be particularly difficult and time consuming. 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 ages and grows.

One particular area of software evolution that requires more research, is the evolution of released and installed applications. To deal with the evolution of released software, distributors and vendors currently have the choice of either buying an (expensive) general product updating tool or building propietary tools. After a thorough analysis, to be presented in this paper, we conclude that both approaches unfortunately have significant problems. On the one hand, existing software update tools usually do not provide all the required functionalities. On the other, the effort and risk of building product update tools "in house" is often underestimated.

The contribution of this article is threefold. Firstly, a process model is provided that embodies the software update process and the uncovered areas of deployed software evolution. Secondly, a typology is provided to classify software product updaters. Finally, the process model is used to compare current techniques and technology, and to indicate what areas still need to be covered.

Updating software can be seen as moving from one configuration to another by addition, removal, replacement, or reconfiguration of software functionality. A physical software update contains the applicable functionality and configuration alterations. By this definition, changing a license or some configuration setting can also be seen as part of the software update process. To discuss the concepts and technologies of this paper, we introduce the notion of software product updaters. A software updater automates the process steps involved with software updates. The main aim of a product updater is to continuously support user needs within changing circumstances. A product updater must communicate updated configurations of components to users but also communicate back to the vendor what parts of the environment have changed, such as necessary components and changed user requirements.

The remainder of this paper is organized as follows. Section 2 describes what the software update process looks like. The steps that make up the update process are modelled and explained. We also provide a typology for updaters and finally evaluate current software updaters in relationship to the process model. Section 3 further defines the steps of delivery and deployment and uses the

detailed descriptions to evaluate the same updaters against the detailed descriptions. Finally, we discuss the presented process model and our future work in Section 4.

2. The Product Software Updating Process

2.1. Update Process Model

This Section describes the software product update process model and a detailed description is given of the steps that make up the process. The update process model has two participants: the customer and the vendor. The process model, shown in Figure 1, is based on customer states and vendor-customer interaction, and has been derived from other update model descriptions and the evaluated tools. The customer blocks are states in this diagram, with the bold lined states being final states. The Uninformed Customer state is the start point for the process. Solid arrows are state transitions, which can be activated by both the vendor and by the customer. The dotted arrows show interaction between the vendor and the customer. Once the vendor offers the customer the ability to update a product of that vendor the update process is initiated. The following list describes all the process steps in the product update model in detail:

Advertise Update - An update will first be made available in some release repository. When a vendor wishes to provide updates to its customers, the customers first need to be informed through the available communication channels.

Receive Information - Customers inform themselves about updates from a vendor through commercial channels, such as web sites, mail, e-mail, and portals. Other channels are memory resident notifiers, such as the Windows Update Notifier, and memory resident processes that automatically start downloading an update once a customer accepts the update that is sent.

Receive Update - A customer can receive an update automatically and manually. Issues for receiving the update are security, authenticity of the update, and integrity checks. Another issue is the checking of pre-download dependency checks such as available disk space and the presence of dependent components.

Remove Update - The presence of the update data that has been downloaded during the Receive Update step, enables switching between configurations and redistribution of updates. For this reason the remove update is an explicit step in the update process model.

Deliver Update - Once a customer has been informed of an update, the vendor wishes to transfer the update to the customer site by mail, e-mail, a website from which the customer can download (pull) the update, or a memory resident process that automatically receives and installs an update. Several issues, which partly are discussed in this paper, arise when the transfer of an update occurs, such as security problems and the format in which the update is sent to the customer.

Install/Deploy Update - A customer installs an update wishing to gain functionality, improve performance, and fix problems. The deployment of updates is the most complex software update process step, and is explained in further detail in Section 3.

Rollback/Deinstall Update - When a customer wishes to go back to a previous configuration, an update must be rolled back or deinstalled. Deinstallation introduces requirements on the software architecture and its extensibility, such as state transformations to the older configurations, and incremental updates instead of destructive updates.

(Re)configure Update - An update can be (re)configured before activation and after activation. These settings can often be changed at runtime or by editing some configuration file, such as the httpd.conf file for the Apache webserver.

Vendor Feedback - An opportunity that is often missed by software producers, but widely used by for instance Microsoft and Exact Software [1], is the use of vendor feedback after the deployment of an update or component. Feedback generated by the deployer of the update can be sent back to the vendor to be used for future testing and feedback on the deployment process.

Activate Update - After deployment the update must be activated so that the update can be used by the customer. The activation process step is threefold and consists of configuration, a license approval, and running the update. The configuration binds all unbound variabilities that have been introduced by the update. Licensing, if necessary, makes sure that the software update is used according to the vendor-customer contract.

Deactivate Update - Deactivation is required when a user does not want or is not allowed to use the update anymore. The most important part of deactivation is the return of a license key to the licensing system or deployment and distribution system. If a deployment and distribution system is present, the deactivation process could also signal the server so that future updates for the deactivated software are no longer sent to this

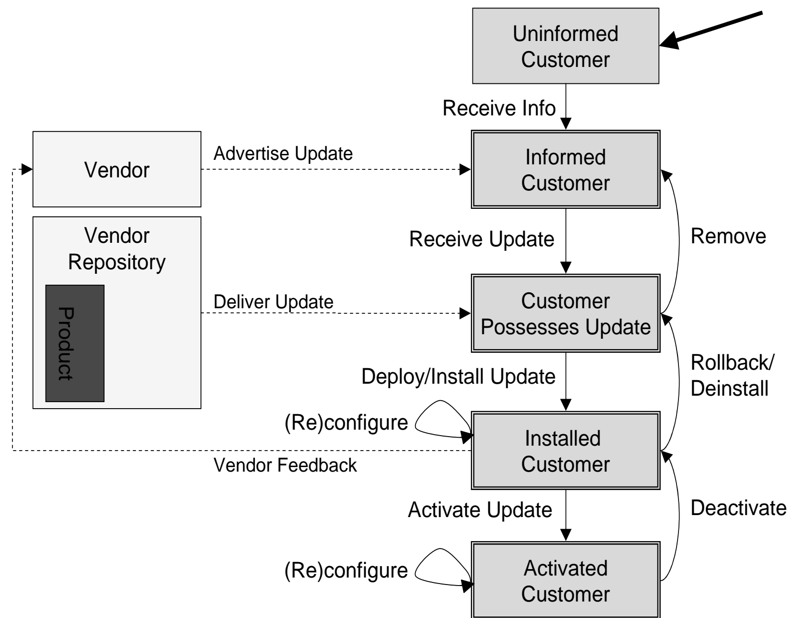

Fig. 1. Update Process Model

workstation or workspace.

For our research we have evaluated the coverage of these process steps for a number of techniques currently used in the field or implemented by academia. The evaluation shows what parts of the process model are still uncovered, how covered process steps have been implemented by the techniques, and what requirements are imposed by these implementations.

Each of the process steps has specific requirements and problem areas. Two process steps that are crucial for the process model, being delivery and deployment, are further explained in detail in Section 3. The release and derelease steps on the vendor side have not been included in this model. The reason for this is that in this paper we do not focus on the processes that take place on the side of the software vendor. At present our focus lies on the implementation and process model of product update software and we are less interested in the development process of the software that is actually distributed. The same holds for the channels through which the software and requirements are communicated. In our model we assume the presence of one vendor and one customer. A more complex scenario is imaginable where a vendor distributes through a software distributer or where a vendor uses COTS (commercial-off-the-shelf components) in its product, however, we abstract from such scenarios because these do not add to our contribution.

2.2. *A Typology for Product Updaters*

In order to obtain more insight in the available product update technology we distinguish three types of product updaters. The typology is created because it creates more insight into the specific available technology and draws out the process model for evaluation of product software update techniques. The three types are distinguishable by looking at delivery and deployment methods and policies, and by looking at process coverage.

- **Package Deployment Tools -** During the evaluation of update tools many package deployment tools (PDTs) were encountered. These deployment technologies are based on the concept of a package, and on a site repository that stores information representing the state of each installed package. A package is an archive that contains the files that constitute a system together with some meta-data describing the system. Examples of these package tools are Red Carpet, APT, Loki-

Update[1], RPM-update[2], Nix [2], SWUP[3], and Portage[4]. RPM, Portage, and Nix are the most advanced.

- **Generic Product Updaters -** Generic product updaters (GPUs) are updaters that completely abstract from a product and attempt to be usable for any product. Two generic product updaters that are available commercially are InstallShield[5] and PowerUpdate[6].
- **Vendor Product Updaters -** Vendor product updaters (VPUs) specifically facilitate the update process of one product, such as Microsofts Windows XP update, Exact Software's Product Updater [3], and Symantec's LiveUpdate.

The typology described above is largely inspired by Carzanigas grouping [4] of deployment techniques and Ajmanis listing of update techniques [5]. One specific technology has not yet been included in the typology, being runtime updaters, which are further discussed in Section 4. This technology, however, can still be described using the updater typology.

2.3. Evaluation of Update Process Coverage

In Table I is displayed how the evaluated update techniques cover the process steps that make up the update process model. The process coverage for update techniques shows different classes of updaters and enables identification of updaters. The process coverage also displays what areas certain techniques focus on and what process steps need more research from both academia and the industry. • Means that a process is completely covered. ○ Means that the process is only partially covered. Coverage has been evaluated based on a number of characteristics of each process step, but for the sake of brevity we cannot go into more detail. For instance, partial support for "send update" means that there are means to get the update to the customer, such as a release repository and communication channels. Full support for "send update" means that push technology is also available.

2.4. Discussion

When looking at the process coverage of the various techniques, there are clear distinctions between the types. One of those distinctions is that

ProductName	Type	Customer interaction			Transferral		Deployment				Licensing	
		Receive Info	Advertise Update	Vendor Feedback	Receive Update	Send Update	Install Update	Rollback	Remove	Re-configure	Activate Update	Deactivate
PowerUpdate	GPU	○	○	○	•	○	○		•			
InstallShield	GPU	○	○	○	•	○	○		•		•	•
Red Carpet	GPU	•	•	•	•	•	•	○	•	○		
Software Dock	GPU	•	•		•	•	•	•	•		•	•
FileWave	GPU			○	•	○	○	○	○		•	•
APT	PDT				•	○	•		•			
RPM-update	PDT				•	○	•		•			
Nix	PDT				•	○	•	•	•	•		
SWUP	PDT				•	○	○		•			
Portage	PDT	○	○		•	○	•	○	•	•		
Loki Update	VPU	○	○		•	○	○					
Exact PU	VPU	○	○	○	•	○	○				○	○
MS SUS	VPU	•	•	○	•	○	○					
LiveUpdate	VPU	•	•		•	○	○				•	•

Legend: • Full support; ○ Partial Support
GPU: General product updater; VPU: Vendor product updater; PDT: Package deployment tool

TABLE I
UPDATE TECHNIQUE PROCESS COVERAGE

[1]http://www.lokigames.com/
[2]http://www.kleemann.org/rpm-update/oldindex.html
[3]http://swup.trustix.org/
[4]http://www.gentoo.org/doc/en/portage-manual.xml
[5]www.installshield.com
[6]www.powerupdate.com

current package deployment tools do not support any form of vendor feedback. We will not discuss each type of updater.

The generic product updaters (GPUs) cover many of the process steps. Especially in the area of licensing and customer interaction the GPUs are strongly represented. Firstly, the GPUs have to be used by different parties, sometimes even using different platforms, and therefore need to provide as many different update scenarios as possible. Secondly, the GPUs in this evaluation are, with the exception of the Software Dock [6], commercial tools, and therefore licensing and customer interaction are required. Finally, when compared to other updaters, the GPUs have most options for vendor feedback, which is a commercially attractive solution for getting feedback from customers.

The package deployment tools (PDTs) are tools specifically designed to deploy and install packages on (usually) open source based systems. These systems are often extended with external tools from which our evaluation abstracts. The tools therefore cover all standard process steps strongly, but in the areas of customer interaction and licensing they are not sufficient. The reasons for this are part of the nature of package deployment. Firstly, issues such as vendor feedback are solved on another level, usually through bug reporting systems and developer communities. Secondly, licensing is not an issue, since most of the software available in the open source community is free.

Vendor product updaters (VPUs) are generally weaker in the areas of transferral and deployment, yet stronger in the areas of customer interaction and licensing. In the area of customer interaction the VPUs are strongly represented, because that is their "bread and butter". One clear distinction between VPUs and GPUs is that removal and rollback is not supported in most VPUs. Whereas GPUs assume that the deployed products will be removed, VPUs assume their products and updates will remain deployed forever, which is not surprising in the case of updates for a virus removal tool or security updates. VPUs are have restricted functionality, because they have been designed to only perform these steps for one product and one way of vendor-customer interaction. We see that many of the methods used in VPUs are simplifications of the more complex software update models.

3. Delivery and Deployment

Two steps in the proposed process model form the core of our model, being delivery and deployment. In this Section the process steps of delivery and deployment are further explained. The updating techniques are then evaluated against the provided definitions.

3.1. Delivery

Delivery formats identify many characteristics of updaters. Some updaters, such as PDTs focus on the sole delivery of packages, whereas GPUs attempt to support the full myriad of delivery formats. Delivery formats affect the size of updates that are delivered to customers. The choice of delivery format therefore affects the total model of delivery, especially in an environment with limited resources.

New configurations can be delivered to customers in different ways. The configurations can be transferred in the following formats:

- **Packages of Components -** A package of components can be delivered to a customer. Usually these packages first need to be unpacked, before they can be installed and activated. Examples of techniques that use packages are RPM-update, APT, DeployMe, Red Carpet, Portage, and Nix.
- **Components -** A separate component consists of a batch of files.
- **Files -** The simplest form of transfer data are separate files. These files can be licenses, configuration settings, and binaries.
- **File deltas -** Differences between a customer site configuration and a vendor site configuration can be expressed as file deltas. File deltas can be transferred using efficient algorithms such as Rsync [7]. A file delta is a listing of differences between two file versions, with which any of the two versions can generate the other version. Sending just the difference between files is more efficient than sending the complete file.

Without some pre-processing at the customer site, each of these formats would place some restrictions on the final deployment environment. However, when correctly assembled before deployment these formats are interchangeable. For example, file deltas for a complete component can be used to generate the new component. The chosen delivery format(s) affect different factors, such as the size of updates and the deployment method, and together with the deployment issues and deployment policies uniquely identify an updater. Service packs are similar to component packages in our delivery formats.

3.2. Deployment

The process of installing updates introduces most complexity for software vendors. The software architecture of a system determines the extensibility of the system, whether the update can occur at runtime or not, and whether there are scripting tools available to perform certain tasks (such as *Make*). Finally, dependencies need to be checked during deployment, such as dependencies on the operating system, the presence of certain components, the compatibility between the update and the current customer configuration, and many others.

To deploy or install the delivered software, a choice for an appropriate deployment method needs to be made. Some of these methods are:

- **Overwrite -** The deployment method employed most often by software vendors is the method of overwriting the application files, license files, or configuration settings. The solution bases itself on the assumption that the deployed set of files or components does not change over time due to external forces. There is no way to rollback an overwrite, unless the customer is using a versioned file system. One example of an overwriting update method is the Windows Updater which will first unregister a dll, overwrite it with a newer version, and register it again. Another example is the Exact Software Product Updater, which compares all the versions of the locally available files to the available files on the release site. When there are differences, the product updater overwrites only the different files on the customer site.
- **Plug-in -** Plug-ins are often used to create extensible configurations. The method of using Plug-in architectures simply support the extensions of a configuration by addition and removal of unique Plug-ins. Other Plug-in [8] can handle different versions of the same Plug-in as well.
- **Deinstall/Reinstall -** For many applications an update constitutes the uninstallation of all previous installed versions of that application[7].

In the open source community applications are often delivered and deployed as source distributions. These source distributions first need to be compiled, which can be seen as a separate step in the deployment process. Well known systems that assist with source distributions are Maak [9] and RTools [10]. It should be noted that the three deployment methods mentioned above can just as well be applied to source distributions.

[7]Examples are: NullSoft Winamp, LavaSoft Ad-Aware, etc

Other issues that deal with deployment are the ability of a technique to provide scripting, to do dependency analysis, to perform integrity checking, to deploy multiple versions of the same component, and to enable push technology. Each of these abilities puts specific requirements on the deployment and implementation architecture.

Scripting is used to perform post deployment configuration on an update. Such scripts can be used to execute, activate, configure, compile and build an update. Scripts can be shell scripts, which are often used by package deployment tools, but also a specifically designed language that registers or unregisters Plug-ins. In the process model presented in Figure 1 we did not yet introduce verification of an update, such as synchronisation checks, signatures, and completeness checkers. In each of the three final states, a customer should be able to perform verification steps.

Dependency analysis is a much studied area of deployment [11] and aims to provide a complete and consistent set of components. To achieve this goal many problems need to be tackled, such as support for multiple versions of components, automatic resolution of dependencies, and explicit management of the dependencies. One specific ability of dependency checking that places extra requirements on the deployment architecture is the support for multiple versions of a component. Multiple version support is therefore part of the evaluation process model and is a technology that enables switching between configurations and having two components depend on different versions of another component. Finally, push technology puts extra requirements on the implementation of the messaging architecture of an updater. A customer needs to be able to receive updates automatically and the vendor needs to be aware of all the customer workspaces.

3.3. Evaluation of Delivery and Deployment

The evaluation of the following techniques includes more specific definitions of the delivery and deployment process steps than the evaluation done by Carzaniga et al [12], because the definitions need to be made more explicit. The evaluation shows that updaters grouped by just the process coverage do not distinguish subtle yet important differences in delivery formats and deployment policies. These differences have been listed here, and provide a more detailed and defined evaluation framework. To obtain the detailed framework,

we have focussed on delivery and deployment. Delivery and deployment are more complex than the other process steps, because there are more alternatives to efficiently achieve the goals that are part of these process steps.

The evaluation in Table II includes a description of what formats of delivery are used by each updater. The evaluation also describes what deployment methods and architectures are supported by each updater. Finally, some issues that uniquely identify an update technique are evaluated. The criteria for evaluation are similar to those for Table I.

From the evaluation of the updaters against the descriptions of delivery and deployment we deduce the following. To begin with, the generic product updaters (GPUs) support all different delivery formats. Especially the two most advanced tools in this category, PowerUpdate and InstallShield, are the only tools able to deal with all formats of delivery. These are also the only tools that are able to send across file deltas, instead of complete files. The GPUs are not well represented in the deployment feature area, because these features are specific to deployment environments, from which the GPUs wish to abstract. However, GPUs are quite able when it comes to commercially interesting push technology, especially when compared to the other updater categories. GPUs generally do not make use of Plug-in technology, which can be explained by the fact that Plug-ins are largely dependent on the Plug-in software architecture. GPUs are strongly represented for the feature of scripting since it is required to perform post installation configuration steps.

The package deployment tools (PDTs) support only package deployment and generally only support deinstallation and reinstallation to update a package. Scripting and dependency analysis are always present in package deployment systems, to enable post deployment configuration and completeness checking with other components. PDTs do not use push technology, which can be explained by the fact that (open source) users of these PDTs often do not want others to be in charge of their software. PDTs are strongly represented in the areas of dependency analysis and integrity checking. The dependency analysis is required for PDTs because packages have many dependency relationships with other packages. Automatic resolution of these dependencies therefore is a valuable feature. Integrity checking prevents instability and ensures authenticity.

Finally, the Vendor product updaters (VPUs) all depend on files as the primary format of transfer to the customer. These files generally overwrite the previous installation, except when these files are special Plug-ins, such as virus definitions for LiveUpdate or unregistered dlls for Microsoft SUS. The VPUs do not incorporate much dependency analysis, scripting, or integrity checking. Finally, the VPUs do not make use of push technology.

4. Discussion and Future Work

The aim of this paper is to show that there is no product updater that provides all functionalities required by software vendors. On the other hand the development of VPUs is not an efficient solution, since each software vendor is implementing a subset of the process steps shown in our process model. It is surprising that no GPU has yet been adopted universally by the industry. One of the reasons for presenting the process model in Figure 1 and the typology is to redefine the requirements on and re-establish the need for such GPUs.

4.1. Typology

The types presented in the typology all have specific requirements and functionalities. To begin with GPUs are generally commercial tools focussed on deploying software on Windows based systems, with the exception of PowerUpdate, which is now focussing on multi platform deployment.

Secondly, the discussed PDTs have some interesting characteristics. Nix, for instance, is a “stop the world” system, whereas Portage and RPM simply extend current functionality. Nix, however, stores components in isolation from each other in a part of the file system called the store, where each component has a globally unique name that enables pointer scanning. The construction of component configurations and the resulting closures are described using Fix store expressions. Safe deployment is achieved by distributing these expressions, along with all components in the store referenced by them.

Another interesting PDT is Portage. Portage, as most other package management system, can resolve dependencies; but one feature that makes it different is the fact that it also supports conditional dependencies. By changing one configuration variable in a Portage configuration file it can disable optional support (and thus the need to depend on it) for particular features or libraries at compile time. In addition Portage enables multiple versions of packages installed simultaneously to satisfy the demands of other packages. The traditional approach to this problem has been to treat different versions

		Delivery Format				Deployment Policy			Deployment Issues					
	Type	Package	Component	Files	File delta	Overwrite	Plug-in	De-Reinstall	Scripting	Dependency Analysis	Integrity Checking	Multiple Versions	Push	Environment Checking
PowerUpdate	GPU	●	●	●	●	●		●	●		●			●
InstallShield	GPU	●	●	●	●	●		●	●					●
Red Carpet	GPU	●		●		●		●	●	●	●		○	○
Software Dock	GPU	●	●			●	●	●	●	●	○	○	●	○
FileWave	GPU			●		●					●			○
APT	PDT	●						●	●	●	●			
RPMupdate	PDT	●						●	●	●	●			○
Nix	PDT	●				●		●	●	●	●	●		○
SWUP	PDT	●						●		●	●			
Portage	PDT	●						●	●	●	●	●		○
Loki Update	VPU			●		●					●			○
Exact PU	VPU			●		●								○
Windows XP SUS	VPU			●		●	●			○	○			○
LiveUpdate	VPU			●		●	●				●			

Legend: ● Full support; ○ Partial Support
GPU: General product updater; VPU: Vendor product updater;
PDT: Package deployment tool

TABLE II
UPDATE TECHNIQUE BUSINESS AND DEPLOYMENT ISSUES

of the same package as different packages with slightly different names, such as with RPM and APT.

Thirdly, there are advantages and disadvantages to VPUs. To begin with there are commercial advantages to VPUs. An important reason for using VPUs instead of GPUs for software vendors is that they themselves are responsible for the update processes of their products. For Norton Anti-Virus for example, Norton is completely responsible for security procedures, network management, and all other aspects having to do with product updating. Often VPUs are a cheap solution over GPUs, however, VPUs can only cover a small problem area compared to general product updaters and the complexity of the software updating process grows as requirements increase. When requirements are stated for the product updater to support different versions, customers, customisations, and licenses, it soon becomes apparent to the software vendor that specialized knowledge is required. The limited availability of such tools and the cost of implementing a GPU, have lead many software vendors to develop their own VPUs and essentially reinvent the wheel. Another disadvantage is that the updaters commonly perform destructive updates. Microsoft Software Update Services, for instance, overwrites dlls, without any rollback functionality.

A category of update technology that is not specified in this paper is runtime updating, because run-time updating is not widely applied for software products yet. Much work has been done in the areas of runtime and dynamic updating [13]. Providing a service or system that is available 24 hours a day is a commercially attractive solution to many problems. These systems of course also evolve with time, thereby requiring some extensible mechanism. We shall not list these mechanisms here, but Ajmani has created a list of mechanisms and component frameworks [5]. There are two important factors to consider when looking at runtime updating, being continuity and state transfer [14] [15]. An interesting technique, designed by Ajmani and Liskov [8], attempts to support many different versions of one component at runtime, thereby enabling runtime extension. Runtime updaters, however, are generally focussed around one technology, such as CORBA or J2EE, and do not focus on

any other process modules than transferral and deployment. Simple versions of these technologies are often used in other product updaters, such as Microsoft SUS or LiveUpdate.

4.2. Delivery and Deployment

The discussed features of the deployment process introduce many questions about software updating techniques. To begin with, the file delta format and push technology is not (yet) strongly represented among the evaluated software updaters. The absence of the file delta format can be explained by the fact that bandwidth and diskspace are cheap nowadays and therefore the time and money invested in such technology is not profitable. The fact that push technology is hardly available can be explained by the type of software evaluated. Most of the techniques mentioned in this paper are product updaters and customers are more interested in having a working product than a product that is acutely and always up to date. Secondly, multiple versions are only supported by technologies from academia (software dock, Nix) and Portage. The complexity of dealing with multiple versions of the same component, which is crosscutting through a system, has not received sufficient attention. Finally, practically all tools perform some deployment environment checking, whether the tool checks for diskspace, such as the Exact Product Updater, or provides an advanced customizable checking mechanism, such as the PowerUpdate and InstallShield GPUs.

4.3. Future Work

One requirement that has as of yet been undiscussed is what Carzaniga et al [12] refer to as site abstraction, the ability to abstract from the vendor-customer model and introduce one or more redistribution sites into the model. Carzaniga et al already refer to a redistribution tool, the Interdock, in their model, yet no implementation has yet been created. An open research issue is to redefine such an architecture where (re)distribution of components, files, licenses, and configuration settings are modelled.

The aim of the issues listed in this paper is to explicitly define software update problems experienced in the field. One striking conclusion that can be drawn from the evaluation is that reconfiguration is highly underestimated for product updating. Another problem is that many of the requirements of software vendors for product updaters are not yet satisfied by GPUs.

The listed techniques can support the industry and can be inspirational for those designing their own technique. The presented material paves the way to build a generally applicable product updater. However, many of the problems mentioned in this paper have already been solved by tools such as Nix and the Software Dock. Our plan is to reuse some of these techniques and build a component framework to support all aspects of the software update process. Such a tool can contribute to the industry and open source community as a plaftform for development of update techniques and provide a standard architecture for such a tool.

4.4. Related Work

Carzaniga et al [12] described some of the techniques mentioned in this article, however, recent developments have lead to new insights and techniques. For the evaluation a list of techniques focussed on runtime updating from Ajmani [5] has been used. On the lower levels of component update architectures, Clegg [16] provides an evaluation of component update methods for implementers of run-time updating.

4.5. Conclusion

The contribution of this article is threefold. To begin with we present a process model that represents the software update process and uncovers the areas of deployed software evolution that require more research. Also, we provide a typology that classifies software updaters. Finally, we use the process model and typology to compare current update tools.

REFERENCES

[1] S. Jansen, G. Ballintijn, and S. Brinkkemper, "Software Release and Deployment at Exact, A Case Study Report." Technical Report CWI, 2004.

[2] 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.

[3] S. Jansen, G. Ballintijn, and S. Brinkkemper, "Integrated SCM/PDM/CRM and Delivery of Software Products to 160.000 Customers," in *Technical Report CWI, sumbmitted for publication*, 2005.

[4] R. S. Hall, D. Heimbigner, and A. L. Wolf, "Evaluating software deployment languages and schema," in *ICSM*, 1998, pp. 177–196.

[5] S. Ajmani, "A review of software upgrade techniques for distributed systems," Aug. 2002.

[6] R. 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] A. Tridgell, "Efficient algorithms for sorting and synchronization," Ph.D. dissertation, 1999.

[8] S. Ajmani, "Automatic software upgrades for distributed systems," Apr. 2003, ph.D. thesis proposal.

[9] E. Dolstra, "Integrating software construction and software deployment," in *11th International Workshop on Software Configuration Management (SCM-11)*, ser. Lecture Notes in Computer Science, B. Westfechtel, Ed., vol. 2649. Portland, Oregon, USA: Springer-Verlag, May 2003, pp. 102–117.
[10] H. E. Harrison, S. P. Schaefer, and T. S. Yoo, "Rtools: Tools for software management in a distributed computing environment," Summer 1988, pp. 85–93.
[11] M. Larsson and I. Crnkovic, "Configuration management for component-based systems," in *Proc. Int. Conf. on Software Engineering (ICSE), May 2001.*, 2001.
[12] A. Carzaniga, A. Fuggetta, R. Hall, A. van der Hoek, D. Heimbigner, and A. Wolf, "A characterization framework for software deployment technologies," 1998.
[13] M. W. Hicks, J. T. Moore, and S. Nettles, "Dynamic software updating," in *SIGPLAN Conference on Programming Language Design and Implementation*, 2001, pp. 13–23.
[14] V. Mencl, Z. Petrova, and F. Plasil, "Update description language," in *Week of Doctoral Students WDS 99*, 1999.
[15] R. Bialek and E. Jul, "A framework for evolutionary, dynamically updatable, component-based systems," in *The 24th IEEE International Conference on Distributed Computing Systems Workshops*, Hachioji, Tokyo, Japan, March 23-24 2004, pp. 326–331.
[16] S. Clegg, "Msc independent study: Evolution in extensible component-based systems," 2003.

Acknowledgment

We would like to thank Eelco Dolstra for our fruitful discussions on configuration settings. We would also like to thank Sameer Ajmani for providing an unpublished list of update techniques on-line and Tijs van der Storm for extensively reviewing the paper. Finally, we would like to thank Arie van Deursen for his helpful review.

Appendix
Short Description of Update Technologies Used

PowerUpdate - PowerUpdate is a commercial multiplatform software updating and delivery tool designed to maintain software applications. PowerUpdate can be integrated into integrated development environments and supports features such as environment analysis and cross platform deployment. PowerUpdate can also check integrity of products on the customer side.

InstallShield - InstallShield is PowerUpdate's largest competitor and differs from PowerUpdate in the facts that it is only suitable for deployment on Microsoft based environments and cannot do integrity checking.

Red Carpet - Red Carpet is a software deployment tool for Linux. Red Carpet works through installtion channels that can be used to communicate and deploy updates at customers. Red Carpet supports automatic dependency and conflict resolution. One important feature of Red Carpet is that they provide Ximian, which is basically a server that contains many different packages that can be deployed for free.

Software Dock - The Software Dock, a project that started at the University of Colorado, is a system of loosely coupled, cooperating, distributed components that are bound together by a wide area messaging and event system. The components include field docks for maintaining site specific configuration information by consumers, release docks for managing the configuration and release of software systems by producers, and a variety of agents for automating the deployment process.

FileWave - FileWave is quite similar to Red Carpet with a lot less features. Mostly, FileWave focusses on deployment of applications on Mac OS X environments, though recently they have started to support Microsoft based environments as well.

APT - The Advaced Package Tool installs packages and manages dependencies automatically for Debian environments. APT has been implemented for Red Hat by Connectiva.

RPMupdate - RPM is the Red Hat Package Manager.

Nix - Nix is a system for software deployment developed by the Trace research group. It supports the creation and distribution of software packages, as well as the installation and subsequent management of these on target machines.

SWUP - Swup is short for "Software Updater" and can automatically update packages together with *cron*, independent of the package manager.

Portage - Portage is the package manager for Gentoo Linux. Portage has some slight advantages over the other package deployment tools, such as conditional dependencies.

Loki Update - The Loki Update Tool is a small tool written to support the most trivial tasks of updating, such as downloading and installing.

Exact PU - The Exact Software Product Updater provides the mechanisms for delivering packages and updates to the customer. When the product updater is run at the customer site, it needs to be provided with an installation location (CD ROM or the Web), a license file and a local installation that is updated.

Microsoft SUS - Microsoft Software Update Service is used for Microsoft Office Update and Windows Update to deliver service packs, bug fixes, and security updates to customers. The updater works mainly at runtime.

LiveUpdate - Symantec provides different types of protection systems for computers connected through a network. Symantecs Antivirus and Firewall software are widely used, and are updated through LiveUpdate. Our evaluation also includes the license tool LiveSubscription, because it covers a relevant part of the update process.