Living on the Cutting Edge: Automating Continuous Customer Configuration Updating

Slinger Jansen
Information and Computing Sciences Institute
Utrecht University, Utrecht, The Netherlands
slinger.jansen@cs.uu.nl

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

Tijs van der Storm

Abstract

Software producing organisations cannot continuously update their end-users' configurations. By not automating continuous updating, costs to test, release, and update a software product remain exorbitantly high and time to market exorbitantly long. This paper displays the feasibility and profitability of continuous customer configuration updating, with the help of two practical case studies and a technique to model and estimate time to market for a software product. Automating continuous updating enables vendors and customers to flexibly define release, delivery and deployment policies, and subsequently reduce time to market with minimum effort. Such flexibility enables customers to be in full control of how and when product knowledge and updates are delivered, tested, and deployed. Furthermore, such flexibility enables vendors to focus effort on its product development instead of release, knowledge delivery, and deployment support tools.

1. Customer Configuration Updating

To date product software is a packaged configuration of software components or a software-based service, with auxiliary material, which is released for and traded in a specific market [26]. Manufacturing product software is an expensive and non-trivial task for software vendors. Software vendors daily face the challenge to divide their resources to develop, release, and deliver high quality software, for which there are always more requirements and opportunities than a tenfold of developers could implement. Furthermore, extreme competition is forcing software vendors to reduce time to market and to release new features as often as possible.

Time-to-market can be shortened by component reuse, reallocation of resources, and by automating processes in the development and delivery cycle. In this paper we focus on process automation of the release, delivery, deployment, and feedback processes, also known as Customer Configuration Updating (CCU). In earlier work customer configuration updating has been positioned as the release, delivery, deployment, and usage and activation processes of product software [11]. The release process describes how and how frequent products and updates are made available to testers, pilot customers, and customers. The delivery process describes the method and frequency of update and knowledge delivery from vendor to customer *and* from customer to vendor. The deployment process describes how a system or customer configuration evolves between component configurations due to the installation of products and updates. Finally, the activation and usage process concerns license activation and knowledge creation on the end-user side.

The importance of CCU is often criticized as systems are more often supplied to customers as an on-line service, thus requiring no more software delivery and deployment on the customer side. There are multiple sides to this "servicization" trend, however. Services are more often deployed on home user systems as well. Simultaneously, an increase is seen in the use of product software on mobile devices, requiring different deployment mechanisms. Examples can even be found of services and products that are deployed on mobile and embedded devices, offered as services, and on customer PCs. Two such examples are TomTom, offering its product on embedded devices in the TomTom Go, as an online service with TomTom Maps, and deployed on your portable device with TomTom Navigator, and Google, offering the Google Mini embedded device, Google on-line websearch, and Google Desktop for the home PC. As such, the demand for research in the area of CCU is increasing, rather than decreasing. Furthermore, interesting blends of software are becoming more common than traditional software packages.

Automating CCU contributes to product software vendors in two ways. First, software vendors can serve a larger number of customers when less overhead is required per customer [9]. Secondly, software vendors become "quicker on their feet" by automating CCU. Due to more frequent integration builds and automated test runs developers see results of their contributions quicker and testers can test more recent versions of the software. Also, developers and testers lose less time (re)deploying the application locally after different developers have contributed to the project. Finally, if customers are involved in testing new features, they can work with newer versions than with traditional weekly or daily builds. Finally, customers are better protected against security holes that can be patched quicker. Overall, by automating CCU a software vendor can reduce its time to market and battle the demand for flexibility of its software products [2].

1.1 Continuous CCU

By automating CCU as much as possible, CCU cost and effort can be reduced effectively and larger numbers of customers can be served [9]. Automating CCU also enables continuous CCU (CCCU) to developers, testers, and even end-users to always be fully up to date. Before an organization can properly set up CCCU, however, policies must be defined for release management, knowledge management and distribution, and software product and deployment architecture. These policies are displayed in Figure 1, which models the CCCU process for any product release, such as a new product, a major update, or a minor bug fix.

We will start by describing the *debug policy*. A software organisation assigns priority to different debugging tasks and enhancement requests and allocates resources to process and perform such tasks and requests. This debug policy ranges from rather static [9] to completely dynamic, such as in SCRUM [21].

Release package planning, which is part of the *release policy*, is the process of defining what features and bug fixes are included in a release package and the process of identifying these packages as bug fix, minor, or major updates, taking into account releases that have been published in the past and the possible update process required to go from one release of the product to another release [12]. Such a planning includes the types of packages a vendor releases (major, minor, bug fix), the channels through which these are released (CD, e-mail, website), and the frequencies when these are released (yearly, monthly, weekly).

With regards to knowledge management and distribution, CCCU forces a software vendor to explicitly manage the flows of knowledge between customer and vendor. A vendor must decide on its (knowledge) *delivery policy* and how customers are informed about new releases, license expiry, product content, etc. On the other hand the vendor must define its *logging policy* and manage all knowledge that comes from customers (*feedback policy*), such as automatically generated error reports, pay-per-usage feedback, etc. Furthermore, the vendor must make explicit the channels that are used, such as web portals, data mining servers (for error reports), release repositories, and newsletters.

Finally, Software vendors must clearly define their *deployment policy* and product architecture. A product's architecture has clear effects on the update, installation, and knowledge creation processes. For instance, the use of a component plug-in architecture facilitates run-time updates, whereas the use of aspect oriented programming allows for easier implementation of logging and error reporting functions. Also, if a software vendor explicitly manages component relations (requires version X of component Y), it becomes possible to calculate complete and correct configurations [10, 5]. Please note that a product's architecture strongly influences the deployment process for product updates. This probably explains the large number of different proprietary deployment technologies noted in [13].

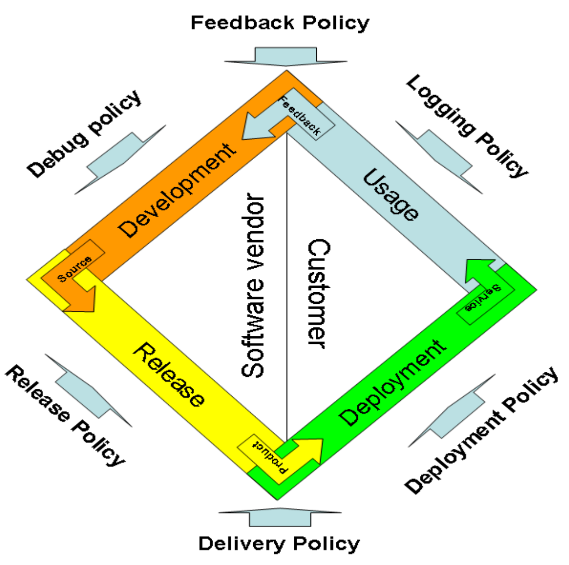


Figure 1. Continuous CCU and its Inputs

CCU automation and reliability are largely dependent on product structure information, such as software component relationships, external product relationships, and hardware compatibility. Such information must explicitly be managed within the development organization to guarantee that a product can reliably be installed, updated, and used [18]. Even though using this type of information is similar to product lifecycle and data management for non-software products, the yields can be greater due to a software product's context.

As stated earlier, CCU automation enables a software vendor to manage more customers by reducing overhead. As such, CCCU should not restrict customers or vendors in any way, it is just a facilitator for more responsive product software management. Automated CCCU enables a software vendor to plan releases in a more structured way, time releases specifically to customer requirements, provide developers and testers with the most recent (working) version available, perform continuous integration builds, perform continuous testing, evolve customer configurations in a less intrusive manner, and provide customers with a guarantee of quality when code is released. If CCCU is fully automated for a software product, the vendor can even consider delivering any version of its software to a customer, such that at any point in time all customers might have a different version of the software running. As long as the software vendor and customer share knowledge and manage it explicitly, processes such as (software and hardware) configuration exploration, bug fix provision and deployment, become trivial.

1.2 Research Approach

The aim of this research is to find out whether automating

CCCU is feasible and profitable for software products. CCCU automation feasibility is established by showing a number of cases that have automated CCCU. Profitability is established by the low overhead for automating CCU, in two presented case studies, while still obtaining a considerable reduction in time to market. First, two cases were selected with different development technology, working on different platforms. Secondly, a tool selection was made to automate and implement the CCCU process for these tools, based on an inventorization of tools. Thirdly, a model has been designed to estimate time to market in the old and new situations. Finally, some aspects of CCCU have been automated for the two tools, implementation time was written down per feature, and the new situation evaluated using the model.

The CCCU automation was considered successful when the following three criteria held true. Updates to the software had to be seamless, without any manual intervention, but with low quality requirements, time to market had to be considerably reduced, and the investment had to be small. Threats to validity are that the tools are not representative, process changes are underestimated, and the model to measure time to market insufficient.

Two applications were selected, Joomla and the Meta-Environment, because we had access to the source. Furthermore, both of the applications' development technologies supported SOAP calls, and the applications are diverse and very different in regards to development methodology and technology. Process changes that are required after automating CCCU are far-reaching and this research only lifts a tip of the veil. The current trend towards agile development, however, only strengthens the belief that CCCU is essential in the current market and that the return on investment into CCCU automation is high. Finally, due to the alikeness between the presented model and development planning models, such as presented in [16], we strongly believe that the presented model is a safe method to estimate time-to-market. Part of our future work is to evaluate this model with industrial data.

2. Related Work and Useful CCCU Tools

In this section some tools and related research projects are put into CCCU perspective. These tools range from scientific open source prototypes to commercial products with a very high turnover. Furthermore, a number of research projects are discussed with objectives in the same area as CCCU automation.

Currently there is a trend in research to focus more on software quality and profiling of software products in their operational environments. One such example is the Skoll project [17], which focuses on the quality assurance processes of software from which more is required due to the growth of the configuration space of software products. Skoll differs from our research due to the fact that Skoll focuses on quality assurance processes surrounding development. EDEM [8], a second research project, uses agents to collect feedback information to improve the development of user interfaces. Finally, the GAMMA project [19] uses a technique called *software tomography* to get useful information from running deployed software. The GAMMA project focuses on data analysis for testing purposes and probe insertion.

2.1 Nix

Nix [4] is a package management system that ensures safe and complete installation of packages for Linux based systems. Nix treats software configuration management as if it were a memory management problem, concurrently storing different versions and variants of components (to satisfy dependencies). Nix ensures that atomic updates and rollbacks are performed separately from dependent configurations, guaranteeing that component dependencies remain satisfied. Such updates can be downloaded from channels in which updates for specific components are published, and can be deployed as binary patches and as source patches. Finally, because Nix satisfies dependencies consistently and completely, it can continuously build and integrate components in a development environment.

With regards to CCCU Nix supports deployment policy management, and can easily be adjusted to fit any CCCU need. However, writing Nix expressions, which describe dependencies amongst components, is not a trivial task. Furthermore, Nix is currently a scientific research tool, that requires components only available under *nix and Cygwin platforms.

2.2 Sisyphus

Sisyphus[1] is a continuous build/release tool for software component configurations. It improves upon other continuous integration systems by maintaining a database that keeps track of which version of which component has been built in what project against which dependencies with what result [23]. Formal reasoning on this database about both the evolution history and the build history of a component including its dependencies, facilitates the optimization of continuous integration. Sisyphus can be implemented with different release criteria, such as "passes the integration build", "passes all tests", and allows for human involvement and quality checking. As such, Sisyphus provides enough flexibility to automate release related CCCCU steps.

2.3 FLEXnet

Macrovision's FLEXnet[2] is a suite of release, delivery, and deployment products, such as InstallShield, InstallAnywhere, FLEXnet Connect, and AdminStudio. Macrovision's product suite provides tooling to create releases for any platform, install them on any platform, and let them be managed by a system administrator. Furthermore, their tools provide licensing, copy protection, and patch delivery solutions. Macrovision's strength can be found in the fact that their tools cover many of the process steps required for CCCU. Their tools have not been used for these case studies, however, due to the products' inflexibility in regards to release, delivery, and deployment policies and the absence of a possibility to distribute software through redistribution points.

2.4 Software dock

The Software Dock [7], 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

[1]http://sisyphus.sen.cwi.nl:8080/

[2]Macrovision

the deployment process. The software dock is an early attempt at correctly, consistently, and in some cases even automatically deliver and deploy software. The Software Dock does not focus on release management and continuous release practices and is lacking in the area of knowledge delivery from customer to software vendor.

3. Tool support: Pheme

Pheme is an infrastructure that enables a software vendor to communicate about software products with end-users and enables system administrators to perform remote deployment, policy propagation, and policy adjustment. The infrastructure consists of a tool (Pheme), a protocol between software product and Pheme, a protocol between Phemes, and a GUI. The Pheme tool is a server that resides on each system that acquires and distributes software knowledge through subscribe/unsubscribe channels. The server can accept and distribute all types of knowledge, including software knowledge delivery and deployment policies that describe behaviour of the Pheme tool. These policies can be manipulated securely and remotely.

Pheme enables software vendors to publish software knowledge in the form of licenses (for one end-user), software updates (for a group of end-users), software content, and software news (for another group of end-users). Pheme enables customers to send knowledge in the form of usage, error, and support feedback. A system administrator can use Pheme to (with the right permissions) instruct other Phemes, change and distribute delivery and deployment policies, control all communication between end-users and vendor, and redistribute software (knowledge). Finally, an end-user can edit policies, execute deployment policies (such as remove/install/update software product), determine when and how feedback will be sent to the vendor, and refresh all types of knowledge such as licenses.

The processes in figure 1 are all covered by advanced tools and methods, such as release by Sisyphus and deployment by Nix. However, none of the tools fully cover deployment policy and delivery policy management for multiple participants in a software supply network (SSN) [14]. Pheme was created to enable these participants to explicitly manage and share knowledge about software components, and thus provide coverage for the release publication, feedback, and knowledge delivery process steps.

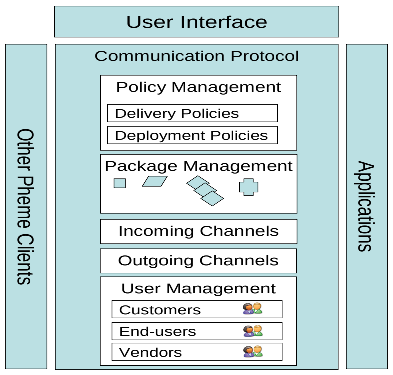


Figure 2. Pheme Architecture

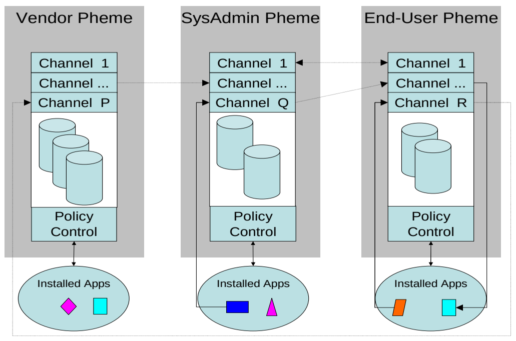


Figure 3. Example Environment for Pheme

3.1 Pheme Architecture

Pheme's architecture is modelled in figure 2. The core components are policy management, package management, user management, and channel management. The policy management component enables the user of Pheme to define delivery policies (check for and download Adobe Acrobat updates on a weekly basis, for instance) and deployment policies (check every time Adobe Acrobat is closed off whether any new updates have been downloaded and deploy the newest one). The package management component supplies Pheme with knowledge package support, in the form of files, reports, and human readable product news.

The user management component enables different types of users to be known to the Pheme instance. Such users can be local administrators, knowledge suppliers, and knowledge customers. The user management component also enables the Pheme administrator to contact other Phemes and indirectly change policies of other Pheme instances. Finally, the channel management component manages different channels that can automatically and manually push or pull knowledge packages to and from other Phemes and URLs.

Pheme can be interacted with through its user interface, through another Pheme, and through a software product using SOAP procedure calls. Software products generally use Pheme as a gateway to a vendor's release repositories and feedback repositories. Typically such interactions include updates, product news, and the like.

An example setting is presented in figure 3, where three different instances of Pheme on three different systems are shown. The vendor can distribute knowledge through its channels to the system administrator. The system administrator can use its channels to communicate with the end-user of a potential product. Finally, the end-user sends knowledge generated by the product back to the vendor. The Joomla case study, presented in the next section, takes place in a similar setting.

4. Results

4.1 Joomla

The first case study is into Joomla[3], a leading open source content management system that is widely adopted as one of the easiest content management systems around, with a large customer base and large collection of components, user generated content, and a large team of developers. Joomla can be installed on any web server that supports PHP and MySql. Traditionally, Joomla applications are updated by hand. Minor releases exclude datamodel changes, so these can overwrite current deployments. For major releases tutorials are released that explain the proper method for updating the datamodel and the source code. The manual process of updating is error-prone, since it is possible to overwrite source files from a Joomla configuration with older versions of files.

The main advantages for the Joomla product in automating CCU are found in effort savings. Each developer, tester, and customer will at some point have to deal with an evolving instance of the product. Their time can be saved by not having to manually evolve the configuration of their Joomla instance. Furthermore, the Joomla development team can automatically generate releases and publish them on-line, which will no longer require the manual effort of checking the release for completeness and such. Finally, Joomla customers can be sure that their website contains the most recent security updates.

To automate customer configuration updating for Joomla the processes of release, delivery, and deployment are automated. With regards to release, the source code is checked out from the Joomla Sourceforge repository on a regular basis. To deliver the source code to the customer the source bundle is sent to the customer system. A policy control system is set up on the customer system to automatically deploy any new versions that come in for Joomla. To automate deployment the system automatically calculates the differences between a new version and the currently deployed version. The changes are then made to the deployed data model and source code.

4.1.1 Automation of CCCU for Joomla

To automate CCCU for Joomla the tools Pheme, SubVersion, DiffTools and MySqlDiff are used. In this case study scenario a release system, an administrator system, and a customer system are used. The release system locally prepares the releases (by simply bundling the source code into one compressed file), the administrator system is used to instruct the customer system with delivery and deployment policies, such as "check for Joomla bundles from the release system every hour" and "deploy a new Joomla version as soon as it arrives". On the customer system Apache and MySql are already running to accommodate Joomla.

The release system downloads a new release when different on a daily basis, by using an automated check-out script. Furthermore, the release system zips the contents of the new release, and places it in a Pheme release channel. The system administrator system also runs Pheme, to instruct the customer system with new policies. The customer system runs Joomla, Apache, and MySql for its daily operations. Pheme is used to check hourly for new updates from the release system and to run the appropriate commands when a new release bundle comes in. When a new release bundle comes in the release is deployed as if it were a "fresh" deployment. The configuration script, configuration.php, is overwritten with the configuration script from the deployed version, with a change in the database name. The datamodel is then created, using joomla.sql. The difference between the datamodels is calculated using MySqlDiff and a data model update script is generated. The update process is a two part process where the database is first updated and then the changed files are overwritten.

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

4.1.2 Technical Challenges

Clearly, our approach is not safe. The automatic data model update does not apply any knowledge from the development process, which can lead to data loss, empty columns, and other data update problems [22, 20]. Furthermore, there is no assurance whether the downloaded bundle is of reliable quality. Also, since update deployments occur at unplanned intervals, the system's state can become unsafe (and thus unreliable). For such runtime updating state-safe techniques can provide solutions [25, 1]. To complete the CCCU process an extension was built to Joomla's error handling mechanism, such that errors and warnings were reported back to Pheme and stored for further analysis. Pheme receives all warnings and errors through a local SOAP call.

4.1.3 Process Challenges

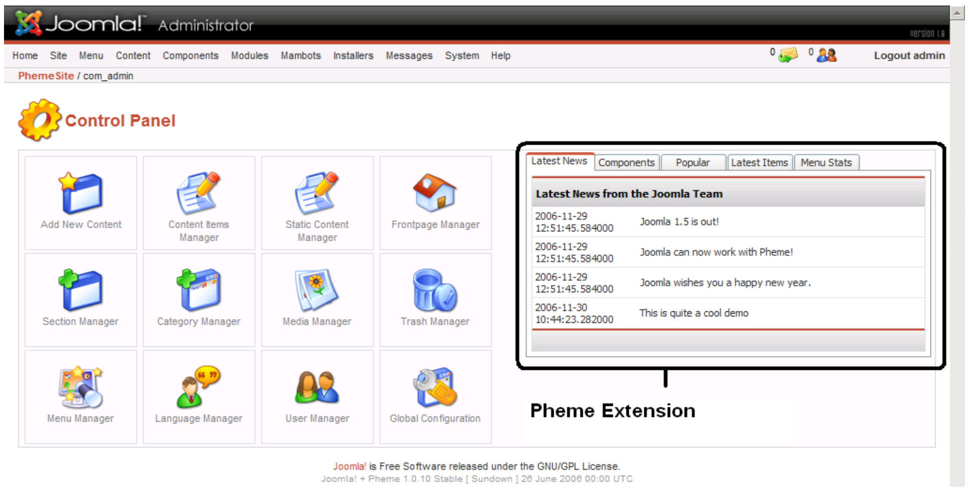
With CCCU in place, a number of process changes are required from the Joomla team. To begin with, the release process needs to be redesigned. To guarantee that high quality code is released to customers, a number of criteria can be devised to delay releases, such as test failures. Joomla can also, instead of opting for continuous CCU, define intervals at which new versions will be released (such as a once weekly update). After all CCCU is a concept that provides maximum flexibility to customers and developers, and should not introduce extra risks or effort. For a future deployment system, knowledge about "preferred update routes" must be specified and shared between the Joomla release team and the Joomla customer, such that quality guarantees can be provided for defined sequences of updates.

4.1.4 Joomla's Software Supply Network

To display news from Pheme in the Joomla backend (see figure 4) a module has been created called ModPhemeNews. This module will only work with approved versions of Joomla resulting in the fact that a customer will have to wait until the ModPhemeNews team approves of any new version of Joomla before an update of the full Joomla package can take place on the customer side, which is a common way of working with add-ons. If CCCU were to be automated in such a process, customers could possess new code quicker. The Joomla creators could, for instance, provide an early release for module and add-on builders. Furthermore, the ModPhemeNews creators could automatically download any new releases for Joomla immediately, automatically test the new combination of Joomla and ModPhemeNews, and notify customers of its approval of the Joomla update.

4.2 Meta-Environment

The second case study concerns the Meta-Environment, a framework for language development, source code analysis and source code transformation that consists of Syntax analysis tools, semantic analysis and transformation tools, and an interactive development environment. The Meta-Environment is an open framework



Latest News from the Joomla Team

2006-11-29 12:51:45.584000	Joomla 1.5 is out!
2006-11-29 12:51:45.584000	Joomla can now work with Pheme!
2006-11-29 12:51:45.584000	Joomla wishes you a happy new year.
2006-11-30 10:44:23.282000	This is quite a cool demo

Joomla! is Free Software released under the GNU/GPL License.
Joomla! + Pheme 1.0.10 Stable [Sundown] 26 June 2006 00:00 UTC

Figure 4. Joomla with Pheme Interaction

that can be easily extended with third-party components, can be easily tailored, modified, or extended, and is supported by an open source community. It is a generalization of the ASF+SDF Meta-Environment that has been successfully used in a wide variety of analysis, transformation and renovation projects. At present the Meta-Environment is a collection of dependent components from different developer and research communities.

For the Meta-Environment the introduction of CCCU decreases development, release creation, and deployment effort. Before introducing CCCU the Meta-Environment was using a DailyBuild system, which performed daily integration builds of components. Dependency information for components comes from package descriptions that are included with each Meta-Environment component, which were only recently (2003) separated using component based software engineering techniques [3]. Under this regime release creation was already a fairly simple task, using tools such as AutoBundle[4] to bundle source components together. These releases, however, could never be guaranteed to work together, due to the fact that component incompatibilities could only be discovered by actually building the component collection.

4.2.1 Automation of CCCU for the MetaEnv

To automate CCCU for the Meta-Environment the release, delivery, and deployment processes need to be automated. The release process needs to provide a new release as soon as one of the Meta-Environment components is changed. Also, the delivery process needs to be automated, such that an end-user system would automatically download a new release every hour. Finally, with regards to deployment it is required that the Meta-Environment is not running at the time of the update.

In regards to release, Sisyphus is used to automatically release the most recent version that fully passed an integration build. To automate delivery, Pheme has been installed on the end-user system to automatically download updates on an hourly basis and deploy the latest version on a daily basis. Pheme checks whether the Meta-Environment is running, and if it is not the new release is installed. A symbolic link to the product is updated, and the previously installed release is deleted afterwards.

[4]http://www.program-transformation.org/Tools/AutoBundle

4.2.2 Technical Challenges

The Meta-Environment does not support any error recovery or reporting. To build in such error recovery, the architecture needs to be redesigned in such a way that the component configuration remains robust and can send an error report before a crash occurs. Even though this is a valuable property, it takes the development effort away from the components' main features.

4.2.3 Process Challenges

The CCCU process for the Meta-Environment is not perfect. To begin with the quality criteria (it must pass the integration build) are rather weak. Sisyphus does provide for stronger criteria and thus better code quality can be delivered. Furthermore, an hourly update, though interesting for the case study, is overkill when it comes to an operational environment. This holds especially if the release frequency is changed by the introduction of new release criteria. Another argument against frequent updates is that a user might be disturbed by frequent user interface and feature changes.

4.2.4 MetaEnv's Software Supply Network

To run the Meta-Environment seven common *nix tools, such as GCC, Bison, and Curl need to be installed. The Meta-Environment is a component composition, with different open source components coming from different development locations. When a Meta-Environment release is created stable versions of components are explicitly labelled to be part of the next release. This method of releasing reduces risk, although it does force a customer to install components that are potentially outdated. In future releases of the Meta-Environment, automation of Sisyphus and strict quality checking enables releases with the most up-to-date compatible components. Components of the Meta-Environment are not strongly coupled, meaning that in the future components could be updated separately if quality guarantees are explicitly defined.

5. Product Development Cycle-Time

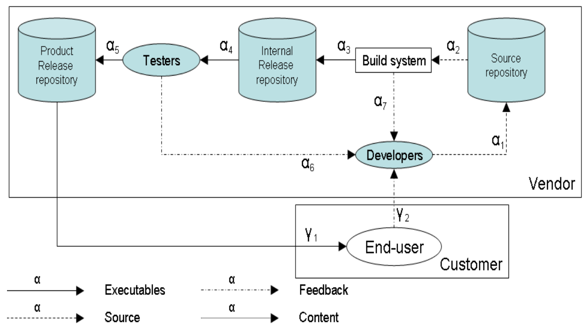


Figure 5. Example Development Cycle-Time Model

To prove that automating CCU in a development organization reduces time-to-market, a model is required that assists in estimating time-to-market of a software product. To estimate time-to-market for a software product the **Product Development Cycle-Time (PDCT) model** is introduced, in part inspired by Ford's dynamic model of product development processes [6]. A PDCT model displays the development cycle of a product release, that is the organizations involved, participants of the development process, repositories in which versions are stored, automatic testing and build systems, and the flows amongst participants, repositories, and systems. Such flows can be source code, executables, feedback, and product content. These flows are annotated with average durations in between transactions of source code, executables, feedback, and product content.

In figure 5 an example is shown of a software vendor that has a traditional product development cycle. A software developer changes code and commits changes to the source repository. The time between commits is named α_1. Furthermore, a build system attempts to build the source code (every α_2) into an executable system. The build time (α_3) determines when the executables are done and uploaded into the internal release repository. Should the build fail, feedback is generated and sent to the developer. Testers decide when to download a newly built release (α_4), when to approve it (α_5), and when not to approve it (α_6). When approved the product release can be downloaded and installed by a customer (γ_1). Finally, the customer will occasionally provide feedback and send this to the software vendor (γ_2).

The example model in figure 5 has three cycles, being the **build feedback cycle (BFC)**, the **test feedback cycle (TFC)**, and the **customer feedback cycle (CFC)**. The PCDT model provides a method to estimate the length of these cycles. The build feedback cycle is the addition of the development and commit time (α_1), the build frequency (α_2), and the time it takes to provide feedback (generally the build time, α_7). Please see equation 1 in figure 6.

The TFC is calculated using the BFC. However, before a release reaches the quality assurance department, the product will go multiple times through the BFC (when code is committed that does not pass the build), of which we call the average A. To calculate the length of the TFC, the BFC is multiplied with $A + 1$, and the feedback time is subtracted. The feedback time must be subtracted once, because in the final cycle when the source code does pass the build, the feedback time is not relevant anymore. We furthermore add the build time (α_3), the test time (α_4), and test feedback time (α_4). This leads to equation 1 in figure 6.

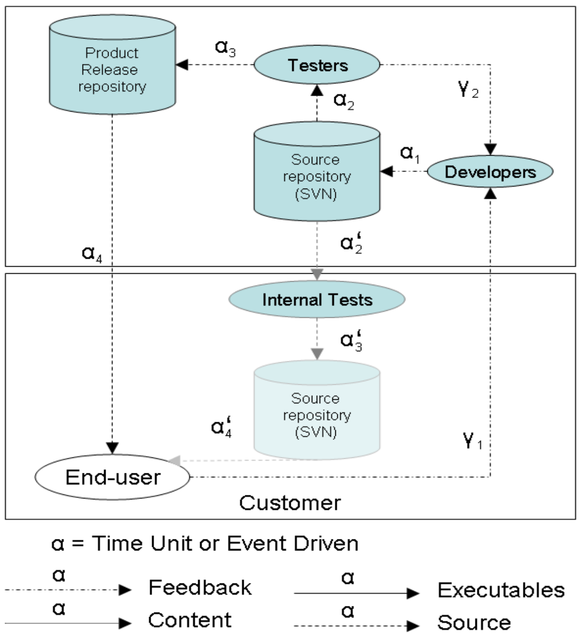


Figure 7. Joomla PDCT Model

To calculate the CFC the TFC duration is required. Similar to the BFC, the TFC is multiplied with the average times the release does not pass the quality assurance team plus once, minus the test feedback duration. Furthermore the time to publish the release in the release repository (α_5), the time to deliver the release to the customer (γ_1), and the time to publish feedback to the software vendor (γ_2) are added. This leads to equation 3 in figure 6. Please note that the **customer delivery time (CDT)** for new features is included in the CFC, such that $CDT(Vendor) = CFC(Vendor) - \gamma_2$ (equation 4).

5.1 Joomla

The PDCT model is used to demonstrate that the market delivery time of Joomla can be reduced. The model in figure 7 shows both the old and the new situation for Joomla. In the old situation major releases are published every three years, minor releases every three months, and bug fix releases are released approximately every 2 months. Upgrading without the DiffTools is a complex task. There are no evolution scripts for the database, and no guarantees can be given whether the update will damage a working website. A common way to update Joomla thus is to migrate all the data from one release to another. This method is tedious and results into Joomla configurations that are rarely, if ever, updated.

Due to the fact that Joomla is built using HTML and PHP code, there is no build cycle. The old test feedback cycle consists of the development and commit time, the test time, and the test feedback time. Furthermore, the customer delivery time consists of the sum of A test feedback cycles plus one, the release publish time, the delivery time, minus the test feedback time. The equations describing the old TFC time and the old CDT can be found in figure 8. In the new model tests run on the customer side to decide whether the

$$BFC(Vendor) = \alpha_1 + \alpha_2 + \alpha_7 \tag{1}$$

$$TFC(Vendor) = (A+1) \cdot BFC(Vendor) - \alpha_7 + \alpha_3 + \alpha_4 + \alpha_6 \tag{2}$$

$$CFC(Vendor) = (B+1) \cdot TFC(Vendor) - \alpha_6 + \alpha_5 + \gamma_1 + \gamma_2 \tag{3}$$

$$CDT(Vendor) = (B+1) \cdot TFC(Vendor) - \alpha_6 + \alpha_5 + \gamma_1 = CFC(Vendor) - \gamma_2 \tag{4}$$

Figure 6. PDC Times for a Software Product

new release will be installed or not. Furthermore, delivery of new releases α'_3 and deployment of these releases α'_4 happens much more frequently. We claim that customer delivery time (and thus the customer feedback cycle time) in the new situations is shorter (equation 8 in figure 8). A lot of quality guarantees are lost in this case because unapproved files are downloaded from the CVS repository. However, by automating CCU, customer feedback will be generated earlier, which increases product quality.

5.2 Meta-Environment

The Meta-Environment PDCT model exhibits some interesting properties. Since the Meta-Environment is a composition of components that are developed simultaneously by different developers, the BFC time has some optional components. The BFC for the component in source repository 1 is the development and commit time α_1, the build time $alpha_2$, and the build feedback time β_1. The BFC time becomes the longest of any of the separate component BFC times (please see equation 9 in figure 10). The TFC time is the BFC time multiplied by the number of times the product does not build plus one, minus the build feedback time (of the longest BFC time). Furthermore the time to build α_7, the tester latency α_8, and the longest feedback time are added.

In the old case the Meta-Environment is built daily. Major releases of the Meta-Environment occur once every two years, minor upgrades once every couple of months, and bug fixes are simply added to the source repository or included in the DailyBuild releases. For these reasons it is not uncommon for end-users to use the DailyBuild release over the latest production release. In the new situation the DailyBuild system is replaced by (or actually run concurrently with) Sisyphus, which already reduces the BFC greatly. BFC duration is reduced because components that are developing slowly are not holding back those components that are developing quickly. Furthermore, end-users are always guaranteed to have the latest version of all components that have successfully been built together (see section 2.2). Finally, due to the fact that Sisyphus continuously builds components instead of daily, feedback time is reduced.

By implementing an automatic update function, users of the Meta-Environment are sure to use the latest version. Due to the fact that the Meta-Environment is constantly updated and does not use a version older than 48 hours, CFC times are reduced and CDT becomes smaller. By automating CCU for the Meta-Environment quantifiable results are yielded.

5.3 Software Supply Networks

A software supply network (SSN) is a series of linked software,

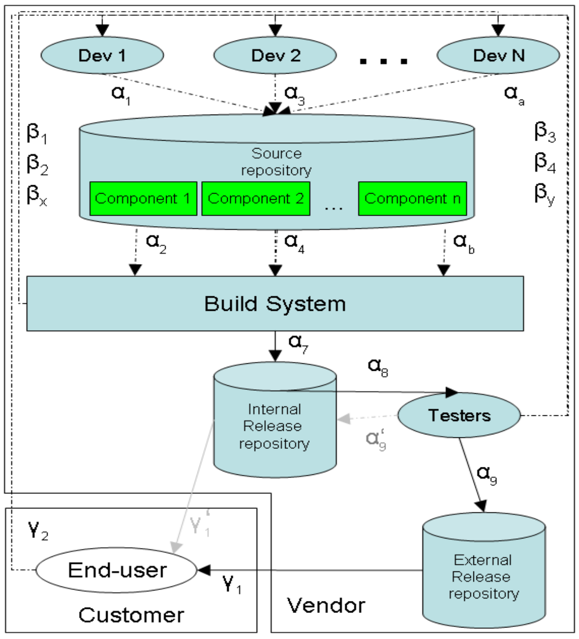


Figure 9. Metaenv Product Development Cycle-Time Model

$$TFC(Joomla) = \alpha_1 + \alpha_2 + \gamma_2 \tag{5}$$

$$CDT(Joomla) = (A+1) \cdot TFC(Joomla) - \gamma_2 + \alpha_3 + \alpha_4 \tag{6}$$

$$CDT'(Joomla) = \alpha_1 + \alpha'_2 + \alpha'_3 + \alpha'_4 \tag{7}$$

$$CDT(Joomla) > CDT'(Joomla) \tag{8}$$

Figure 8. PDC Times for Joomla

$$BFC(MetaEnv) = max((\alpha_1 + \alpha_2 + \beta_1), (\alpha_3 + \alpha_4 + \beta_2), (\alpha_a + \alpha_b + \beta_x)) \tag{9}$$

$$TFC(MetaEnv) = (A+1) \cdot BFC(MetaEnv) - max(\beta_1, \beta_2, \beta_x) + \alpha_7 + \alpha_8 + max(\beta_4, \beta_5, \beta_y) \tag{10}$$

$$CDT(MetaEnv) = (B+1) \cdot TFC(MetaEnv) - max(\beta_3, \beta_4, \beta_y) + \alpha_9 + \gamma_1 \tag{11}$$

$$CDT'(MetaEnv) = (B'+1) \cdot TFC(MetaEnv) - max(\beta_3, \beta_4, \beta_y) + \alpha'_9 + \gamma'_1 \tag{12}$$

$$CDT(MetaEnv) > CDT'(MetaEnv) \tag{13}$$

Figure 10. PDC Times for MetaEnv

hardware, and service organizations cooperating to satisfy market demands. These SSNs have interesting effects on CCCU. To begin with, software vendors must focus on continuously integrating the newest version of COTS components, to achieve high quality of a product. Furthermore, when different organizations work together, they must focus on timing their releases right. In essence, this paces the heartbeat of software development [24].

5.4 Applications

An interesting aspect of the PDCT model is that it enables modelling delivery speeds. As software vendors are trying to decrease their delivery time on new features they are held back by technical restrictions, but also by resource restrictions. One such example is that of a large ERP-Vendor with a customer feedback loop of three and a half months. To decrease delivery time, their development team starts two weeks early on developing a new release. By (re)starting the development cycle of a new release two weeks earlier and disregarding the fact that no customer feedback has come in yet, the development cycle is up and running simultaneously as customers are receiving the previous release. In these two weeks the developers work on new features and remove bugs found earlier in quality assurance processes. This has shortened the release delivery time by two weeks, because now the processes of development and customer feedback partly overlap.

Another interesting case is presented by the Meta-Environment, where components built on different physical locations with their own life-cycle, are required by the product. As can be seen in equation 9 of the Meta-Environment the build feedback cycle time depends on the weakest link, the component that takes most time to develop. Clearly, this enables identification of development activities and critical path analysis. This holds for the Meta-Environment as much as for any developer in a SSN.

6. Conclusions and Future work

To automate CCCU in such a manner that updates are safely and securely deployed (possibly at runtime) the knowledge about a product's structure and architecture must be used. The two presented case studies do not make use of product architecture information and can thus not guarantee safe and secure updates. Possibly, a generic tool can be built that automates the update process for different types of software architectures, such as service oriented architectures, plug-in systems, and monolithic systems (each with its own update mechanism).

A question remains whether software vendors that automate CCU are more prone to code injections. For an open source software product that has penetrated great portions of the market, such as Firefox, getting injected harmful code released becomes easier if CCCU has been fully automated. However, automated CCU also enables a vendor to assume a stronger hold on release quality assurance, by increasing testing, releasing for certain user groups (pilot customers) only, and by formal code verification scripts.

A weakness of the PDCT models is that all durations are averages based on earlier events. The actual values are dependent on a large number of variables, such as number of features implemented in a release, number of developers working on this task, number of bug reports that need to be solved, etc. We do see the PDCT models as a powerful tool, however, to demonstrate how CCCU can reduce time-to-market and as a planning tool. Part of our future work is to evaluate the PDCT in more real life situations.

This paper demonstrates that using a tool for software knowledge, product, and update delivery reduces overhead to automating CCU. With the availability of such tools, software vendors can no longer object to automating CCU, since it enables a software vendor to *pace the heartbeat* of software development and delivery [15, 24]. Furthermore, we make a claim that such pacing shortens development and testing cycles in a software supply network.

This paper presents the concept of continuous customer configuration updating. Secondly, a new tool that facilitates delivery of software knowledge between vendors and customers is presented. Thirdly, a number of tools and prototypes are evaluated and put into CCCU context. Finally a modelling technique is presented that models product development cycle times.

Software vendors must automate CCCU and the presented tools to reduce time-to-market and get quicker feedback from testers and customers. Two case studies show that the effort invested into automating CCU is minimal compared to the reduction in time-to-market. By using the product development cycle time modelling technique, software vendors can easily spot their weaknesses and increase development speed. Furthermore, we hope to make software vendors and end-users aware of the **age of the code** they are working with.

Part of our future work is to build the Pheme prototype into a commercial product in cooperation with a number of industrial partners. Furthermore, we will build an automatic updater into the Meta-Environment and hopefully we will develop an updater for Joomla. Also, the PDCT modelling technique will be used in a number of software businesses, to evaluate their usefulness and establish typical weaknesses and effective tools for CCCU. Finally, we will research whether developers use and commit to a Sisyphus controlled differently than a traditional DailyBuild system.

7. References

[1] S. Ajmani. *Automatic Software Upgrades for Distributed Systems*. Ph.D., MIT, Sept. 2004. Also as Technical Report MIT-LCS-TR-1012.

[2] B. W. Boehm. The future of software processes. In *International Software Process Workshop*, pages 10–24, 2005.

[3] M. de Jonge. Build-level components. *IEEE Transactions on Software Engineering*, 31(7):588–600, July 2005.

[4] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In *Proceedings of the 18th Conference on Systems Administration (LISA 2004), Atlanta, USA, November 14-19, 2004*, pages 79–92, 2004.

[5] E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 583–592. IEEE, 2004.

[6] D. N. Ford and J. Sterman. Dynamic modeling of product development processes. In *System Dynamics Review*, pages 31–68. John Wiley and Sons, Ltd., 1998.

[7] 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*, pages 174–183, 1999.

[8] D. M. Hilbert and D. F. Redmiles. Extracting usability information from user interface events. volume 32, pages 384–421, New York, NY, USA, 2000. ACM Press.

[9] S. Jansen, G. Ballintijn, S. Brinkkemper, and A. van Nieuwland. Integrated development and maintenance for the release, delivery, deployment, and customization of product software: a case study in mass-market erp software. In *Journal of Software Maintenance and Evolution: Research and Practice*, volume 18, pages 133–151. John Wiley & Sons, Ltd., 2006.

[10] S. Jansen and S. Brinkkemper. Modelling deployment using feature descriptions and state models for component-based software product families. In *3rd International Working Conference on Component Deployment (CD 2005)*, LNCS. Springer–Verlag, 2005.

[11] S. Jansen and S. Brinkkemper. Definition and validation of the key process areas of release, delivery and deployment of product software vendors: turning the ugly duckling into a swan. In *proceedings of the International Conference on Software Maintenance (ICSM2006, Research track)*, September 2006.

[12] S. Jansen and S. Brinkkemper. Ten misconceptions about product software release management explained using update cost/value functions. In *First International Workshop on Software Product Management*. IEEE, 2006.

[13] S. Jansen, S. Brinkkemper, and G. Ballintijn. A process framework and typology for software product updaters. In *Ninth European Conference on Software Maintenance and Reengineering*, pages 265–274. IEEE, 2005.

[14] S. Jansen, A. Finkelstein, and S. Brinkkemper. Analyzing the business of software: A modelling technique for software supply networks. In *Conference on Advanced Information Systems Engineering Forum*, 2007.

[15] S. Jansen and W. Rijsemus. Balancing total cost of ownership and cost of maintenance within a software supply network. In *proceedings of the IEEE International Conference on Software Maintenance (ICSM2006, Industrial track), Philadelphia, PA, USA, September, 2006*, 2006.

[16] A. Kusiak. *Engineering Design: Products, Processes, and Systems*. Academic Press, Inc., Orlando, FL, USA, 1999.

[17] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 459–468. IEEE Computer Society, 2004.

[18] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Trans. Inter. Tech.*, 2(2):151–185, 2002.

[19] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: Continuous evolution of software after deployment. In *Proc. of the ACM International Symposium on Software Testing and Analysis (ISSTA'02)*, pages 65–69, 2002.

[20] J. F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

[21] K. Schwaber and M. Beedle. Development with scrum. Prentice-Hall, 2002.

[22] D. Sjoberg. Quantifying schema evolution. *Inf. Softw. Tech.*, 35(1):35–44, 1993.

[23] T. van der Storm. Continuous release and upgrade of component-based software. In *Proceedings of the 12th International Workshop on Software Configuration Management (SCM-12)*, 2005.

[24] T. van der Storm. The Sisyphus continuous integration system. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE

Computer Society Press, 2007. To Appear.

[25] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. An alternative to quiescence: Tranquility. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 73–82, Washington, DC, USA, 2006. IEEE Computer Society.

[26] L. Xu and S. Brinkkemper. Concepts of product software: Paving the road for urgently needed research. In *First International Workshop on Philosophical Foundations of Information Systems Engineering*. LNCS, Springer-Verlag, 2005.