Continuous Release and Upgrade of Component-Based Software*

Tijs van der Storm

Centrum voor Wiskunde en Informatica (CWI)
P.O. Box 94079, 1090 GB Amsterdam
The Netherlands, storm@cwi.nl

Abstract. We show how under certain assumptions, the release and delivery of software updates can be automated in the context of component-based systems. These updates allow features or fixes to be delivered to users more quickly. Furthermore, user feedback is more accurate, thus enabling quicker response to defects encountered in the field.
Based on a formal product model we extend the process of continuous integration to enable the agile and automatic release of software components component. From such releases traceable and incremental updates are derived.
We have validated our solution with a prototype tool that computes and delivers updates for a component-based software system developed at CWI.

1 Introduction

Software vendors are interested in delivering bug-free software to their customers as soon as possible. Recently, *ACM Queue* devoted an issue to update management. This can be seen as a sign of an increased awareness that software updates can be a major competitive advantage. Moreover, the editorial of the issue [7], raised the question of how to deliver updates in a component-based fashion. This way, users only get the features they require and they do not have to engage in obtaining large, monolithic, destabilizing updates.

We present and analyse a technique to automatically produce updates for component-based systems from build and testing processes. Based on knowledge extracted from these processes and formal reasoning it is possible to generate incremental updates.

Updates are produced on a per-component basis. They contain fine-grained bills of materials, recording version information and dependency information. Users are free to choose whether they accept an upgrade or not within the bounds of consistency. They can be up-to-date at any time without additional overhead from development. Moreover, continuous upgrading enables continuous user feedback, allowing development to respond more quickly to software bugs.

The contributions of this paper are:

* This work was sponsored in part by the Netherlands Organisation for Scientific Research, NWO, Jacquard project DELIVER.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SCM 2005, September 5-6, 2005, Lisbon, Portugal. Copyright © 2005 ACM

- An analysis of the technical aspects of component-based release and update management.
- The formalisation of this problem domain using the relational calculus. The result is a formal, versioned product model [4].
- The design of a continuous release and update system based on this formalisation

The organisation of this paper is as follows. In Section 2 we will elaborate on the problem domain. The concepts of continuous release and upgrade are motivated and we give an overview of our solution. Section 3 presents the formalisation of continuous integration and continuous release in the form of a versioned product model. It will be used in the subsequent section to derive continuous updates (Section 4). Section 5 discusses the prototype tool that we have developed to validate the product model in practice. In Section 6 we discuss links to related work. Finally, we present a conclusion and list directions for future work in Section 7.

2 Problem Statement

2.1 Motivation

Component-based releasing presumes that a component can be released only if its dependencies are released [18]. Often, the version number of a released component and its dependencies are specified in some file (such as an RPM spec file [1]). If a component is released, the declaration of its version number is updated, as well as the declaration of its dependencies, since such dependencies always refer to released components as well. This makes component-based releasing a recursive process.

There is an substantial cost associated with this way of releasing. The more often a dependent component is released, the more often components depending on it should be released to take advantage of the additional quality of functionality contained in it. Furthermore, on every release of a dependency, all components that use it should be integration tested with it, before they can be released themselves.

We have observed that in practice the tendency is to not release components in a component-based way, but instead release all components at once when the largest composition is scheduled to be released. So instead of releasing each component independently, as suggested by the independent evolution history of each component, there implicitly exists a practice of big-bang releasing (which inherits all the perils of big-bang integration[1]).

One could argue, that such big-bang releases go against the philosophy of component-based development. If all components are released at once as part of a whole (the system or application), then it is unlikely that there ever are two components that depend on different versions of the same component. Version

[1] See `http://c2.com/cgi/wiki?IntegrationHell` for a discussion.

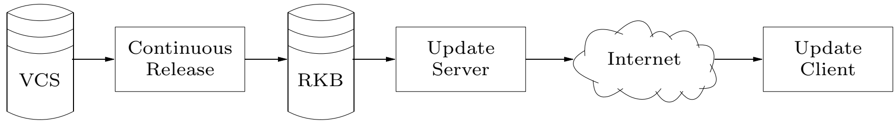

Fig. 1. Continuous Release Architecture

numbers of released components can thus be considered to be only informative annotations that help users in interpreting the status of a release. They have no distinguishing power, but nevertheless produce a lot of overhead when a release is brought out.

So we face a dilemma: either we release each component separately and release costs go up (due to the recursive nature of component-based releasing). Or we release all components at once, which is error-prone and tends to be carried out much less frequently.

Our aim in this paper is to explore a technical solution to arrive at feasible compromise. This means that we sacrifice the ability to maintain different versions of a component in parallel, for a more agile, less error-prone release process. The assumption of one relevant version, the current one, allows us to automate the release process by a continuous integration system. Every time a component changes it is integrated *and* released. From these releases we are then able to compute incremental updates.

2.2 Solution Overview

The basic architecture of our solution is depicted in Fig. 1. We assume the presence of a version control system (VCS). This system is polled for changes by the continuous release system. Every time there is a change, it builds and tests the components that are affected by the change. As such the continuous release process subsumes continuous integration [6]. In this paper, we mean by "integration" the process of building and testing a set of related components.

Every component revision that passes integration is released. Its version is simply its revision number in the version control system. The dependencies of a released component are also released revisions. The system explicitly keeps track of against which revisions of its declared dependencies it passed the integration. This knowledge is stored in a release knowledge base (RKB). Note that integrated component revisions could pass through one or more quality assurance stages before they are delivered to users. Such policies can easily be superimposed on the continuous release system described in this paper.

The RKB is queried by the update server to compute updates from releases. Such updates are incremental relative to a certain user configuration. The updates are then delivered to users over the internet.

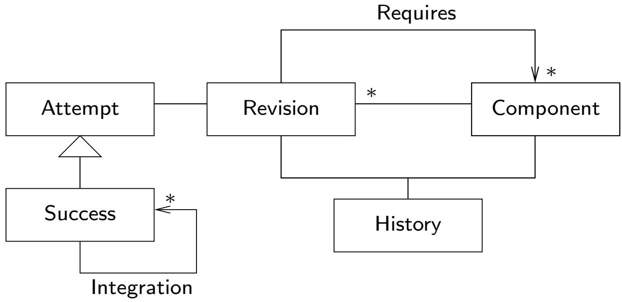

Fig. 2. Continuous Integration Component Model

3 Continuous Release

3.1 Component Model

Our formalisation is based on the calculus of binary relations [16]. This means that essential concepts are modelled as sets and relations between these sets. Reasoning is applied by evaluating standard set operations and relational operations.

We will now present the sets and relations that model the evolution and dependencies of a set of components. In the second part of this section we will present the continuous release algorithm that takes this versioned product model as input. As a reference, the complete model is displayed in a UML like notation in Fig. 2.

The most basic set is the set of components Component. It contains an element for each component that is developed by a certain organisation or team. Note that we abstract from the fact that this set is not stable over time; new components may be created and existing components may be retired.

To model the evolution of each component we define the set of component revisions as follows:

$$\mathsf{Revision} \subseteq \mathsf{Component} \times \mathbb{N}$$

This set contains tuples $\langle C, i \rangle$ where C is a component and i is a *revision identifier*. What such an identifier looks like depends on the Version Control System (VCS) that is used to store the sources of the components. For instance, in the case of CVS this will be a date identifying the moment in time that the last commit occurred on the module containing the component's sources. If Subversion is used, however, this identifier will be a plain integer identifying the revision of one whole source tree. To abstract from implementation details we will use natural numbers as revision identifiers. A tuple $\langle C, i \rangle$ is called a "(component) revision".

A revision records the state of a component. It identifies the sources of a component during a period of time. Since it is necessary to know when a certain component has changed, and we want to abstract from the specific form of

revision identifiers, we model the history of a component explicitly. This is done using the relation History, which records the revision a component has at a certain moment in time:

$$\mathsf{History} \subseteq \mathrm{Time} \times (\mathsf{Component} \times \mathsf{Revision})$$

This relation is used to determine the state of a set of components at a certain moment in time. By taking the image of this relation for a certain time, we get for each component in Component the revision it had at that time.

Components may have dependencies which may evolve because they are part of the component. We assume that the dependencies are specified in a designated file within the source tree of a component. As a consequence, whenever this file is changed (e.g., a dependency is added), then, by implication, the component as a whole changes.

The dependencies in the dependency file do not contain version information. If they would, then, every time a dependency component changes, the declaration of this dependency would have to be changed; this is not feasible in practice. Moreover, since the package file is part of the source tree of a component, such changes quickly ripple through the complete set of components, increasing the effort to keep versioned dependencies in sync.

The dependency relation that can be derived from the dependency files is a relation between component revisions and components:

$$\mathsf{Requires} \subseteq \mathsf{Revision} \times \mathsf{Component}$$

Requires has Revision as its domain, since dependencies are part of the evolution history of a component; they may change between revisions. For a single revision, however, the set of dependencies is always the same.

The final relation that is needed, is a relation between revisions, denoting the actual dependency graph at certain moment in time. It can be computed from Requires and History. It relates a moment in time and two revisions:

$$\mathsf{Depends} \subseteq \mathrm{Time} \times (\mathsf{Revision} \times \mathsf{Revision})$$

A tuple $\langle t, \langle A_i, B_j \rangle \rangle \in \mathsf{Depends}$ means that at point in time t, the dependency of A_i on B referred to B_j; that is: $\langle A_i, B \rangle \in \mathsf{Requires}$ and $\langle t, \langle B, B_j \rangle \rangle \in \mathsf{History}$.

3.2 Towards Continuous Release

A continuous integration system polls the version control system for recent commits and if something has changed, builds all components that are affected by it. After each integration, the system usually generates a website containing results and statistics. In this section we formalise and extend the concept of continuous integration to obtain a continuous release system.

The continuous release system operates by populating three relations. The first two are relations between a number identifying an integration attempt and

Algorithm 1 Continuous Integration

```
 1: procedure INTEGRATECONTINUOUSLY
 2:     i := 0
 3:     loop
 4:         deps := Depends[now]
 5:         changed := carrier(deps) \ range(Attempt)
 6:         if changed ≠ {} then
 7:             todo := deps^{-1*}[changed]
 8:             order := reverse(topsort(deps)) ∩ todo
 9:             INTEGRATEMANY(i, order, deps)
10:             i := i + 1
11:         end if
12:     end loop
13: end procedure
```

a component revision:

$$\mathsf{Attempt} \subseteq \mathbb{N} \times \mathsf{Revision}$$
$$\mathsf{Success} \subseteq \mathsf{Attempt}$$

Elements in Success indicate successful integrations of component revisions, whereas Attempt records attempts at integration that may have failed. Note that Success is included in Attempt.

The second relation records how a component was integrated:

$$\mathsf{Integration} \subseteq \mathsf{Success} \times \mathsf{Success}$$

Integration is a dependency relation between successful integrations. A tuple $\langle\langle i, r\rangle, \langle j, s\rangle\rangle$ means that revision r was successfully integrated in iteration i against s, which, at the time of i was a dependency of r. Revision s was successfully integrated in iteration $j \leq i$. The fact that $j \leq i$ conveys the intuition that a component can never be integrated against dependencies that have been integrated later. However, it is possible that a previous integration of a dependency can be reused. Consider the situation that there are two component revisions A and A' which both depend on B in iterations i and $i+1$. First A is integrated against the successful integration of B in iteration i. Then, in iteration $i+1$, we only have to integrate A' because B did not change in between i and $i+1$. This means that the integration of B in iteration i can be reused.

We will now present the algorithms to compute Success, Attempt and Integration. In these algorithms all capitalised variables are considered to be global; perhaps it is most intuitive to view them as part of a persistent database, the RKB.

Algorithm 1 displays the top-level continuous integration algorithm in pseudocode. Since continuous integration is assumed to run forever, the main part of the procedure is a single infinite loop.

The first part of the loop is concerned with determining what has changed. We first determine the dependency graph at the current moment in time. This

Algorithm 2 Integrate components

1: **procedure** $\textsc{IntegrateMany}(i,\ \mathit{order},\ \mathit{deps})$
2: **for each** r **in** *order* **do**
3: $D := \{\langle i, d\rangle \in \mathsf{Attempt} \mid d \in \mathit{deps}[r], \neg\exists\langle j, d\rangle \in \mathsf{Attempt} : j > i\}$
4: **if** $D \subseteq \mathsf{Success}$ **then**
5: **if** $\textsc{IntegrateOne}(r,\ D) =$ **success then**
6: $\mathsf{Success} := \mathsf{Success} \cup \{\langle i,\ r\rangle\}$
7: $\mathsf{Integration} := \mathsf{Integration} \cup (\{\langle i,\ r\rangle\} \times D)$
8: **end if**
9: **end if**
10: $\mathsf{Attempt} := \mathsf{Attempt} \cup \{\langle i,\ r\rangle\}$
11: **end for**
12: **end procedure**

is done by taking the (right) image of relation Depends for the current moment of time (indicated by **now**). The variable *deps* represents the current dependency graph; it is a relation between component revisions. Then, to compute the set of changed components in *changed*, all component revisions occurring in the dependency graph for which integration previously has been attempted, are filtered out at line 5. Recall that Attempt is a relation between integers (integration identifiers) and revisions. Therefore, taking the range of Attempt gives us all revisions that have successfully or unsuccessfully been integrated before.

If no component has changed in between the previous iteration and the current one, all nodes in the current dependency graph (*deps*) will be in the range of Attempt. As a consequence *changed* will be empty, and nothing has to be done. If a change in some component did occur, we are left with all revisions for which integration never has been attempted before.

If the set *changed* is non-empty, we determine the set of component revisions that have to be (re)integrated at line 7. The set *changed* contains all revisions that have changed themselves, but all current revisions that depend on the revisions in *changed* should be integrated again as well. These so-called *co-dependencies* are computed by taking the image of *changed* on the transitive-reflexive closure of the inverse dependency graph. Inverting the dependency graph gives the co-dependency relation. Computing the transitive-reflexive closure of this relation and taking the image of *changed* gives all component revisions that (transitively) depend on a revision in *changed* including the revisions in *changed* themselves. The set *todo* thus contains all revisions that have to be rebuilt.

The order of integrating the component revisions in *todo* is determined by the topological sort of the dependency graph *deps*. For any directed acyclic graph the topological sort (topsort in the algorithm) gives a partial order on the nodes of the graph such that, if there is an edge $\langle x, y\rangle$, then x will come before y. Since dependencies should be integrated before the revisions that depends on them, the order produced by topsort is reversed.

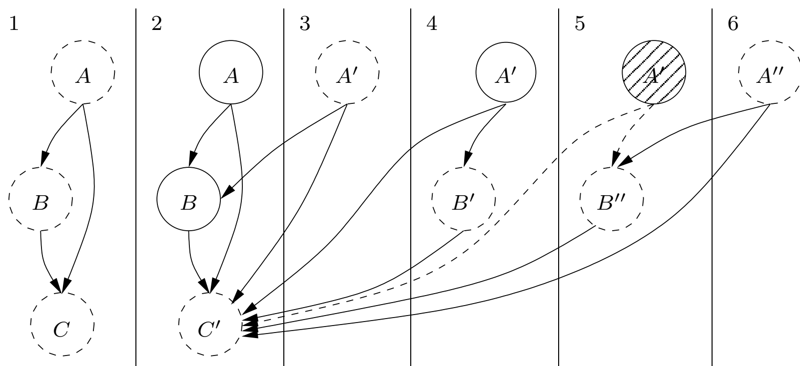

Fig. 3. Six iterations of integration

The topological order of the dependency graph contains all revisions participating in it. Since we only have to integrate the ones in *todo*, the order is (list) intersected with it. So, at line 8, the list *order* contains each revision in *todo* in the proper integration order.

Finally, at line 9, the function INTEGRATEMANY is invoked which performs the actual integration of each revision in *order*. After INTEGRATEMANY finishes, the iteration counter i is incremented.

The procedure INTEGRATEMANY, displayed as Alg. 2, receives the current iteration i, the ordered list of revisions to be integrated and the current dependency graph. The procedure loops over each consecutive revision r in *order*, and tries to integrate r with the most recently attempted integrations of the dependencies of r. These dependencies are computed from *deps* at line 3. There may be multiple integration attempts for these dependencies, so we take the ones with the highest i, that is: from the most recent iteration.

At line 4 the actual integration of a single revision starts, but only if the set D is contained in Success, since it is useless to start the integration if some of the dependencies failed to integrate. If there are successful integrations of all dependencies, the function INTEGRATEONE takes care for the actual integration (i.e. build, smoke, test etc.). We don't show the definition of INTEGRATEONE since it is specific to one's build setup (e.g. build tools, programming language, platform, searchpaths etc.). If the integration of r turns out to be successful, the relations Success and Integration are updated.

3.3 A Sample Run

To illustrate how the algorithm works, and what kind of information is recorded in Integration, let's consider an example. Assume there are three components, A, B, C. The dependencies are so that A depends on B and C, and B depends on C. Assume further that these dependencies do not evolve.

Figure 3 shows six iterations of INTEGRATECONTINUOUSLY, indicated by the vertical swimlanes. In the figure, a dashed circle means that a component has evolved in between swimlanes, and therefore needs to be integrated. Shaded circles and dashed arrows indicate that the integration of a revision has failed.

So, in the first iteration, the current revisions of A, B, and C have to be integrated, since there is no earlier integration. In the second iteration, however, component C has changed into C', and both A and B have remained the same. Since A and B depend on C', both have to be reintegrated.

The third iteration introduces a change in A. Since no component depends on A' at this point, only A' has to be reintegrated. In this case, the integrations of B and C in the previous iteration are reused.

Then, between the third and the fourth iteration B evolves into B'. Since A' depends on B', it should be reintegrated, but still the earlier integration of C' can be reused. In the next iteration B' evolves into B''. Again, A' should be reintegrated, but now it fails. The trigger of the failure is in B' or in the interaction of B' and C'. We cannot be sure that the bug that triggered the failure is in the changed component B''. It might be so, that a valid change in B'' might produce a bug in A' due to unexpected interaction with C'. Therefore, only complete integrations can be reused.

Finally, in the last iteration, it was found out that the bug was in A', due to an invalid assumption. This has been fixed, and now A'' successfully integrates with B'' and C'.

4 Continuous Upgrade

4.1 Release Packages

In this section we will describe how to derive incremental updates from the sets Success and Integration. Every element $\langle i, r\rangle \in \mathsf{Success}$ represents a *release* i of revision r. The set of revisions that go into an update derived from a release, the *release package*, is defined as:

$$\text{package}(s) = \text{range}(\mathsf{Integration}^*[s])$$

This function returns the bill of materials for a release $s \in \mathsf{Success}$.

As an example, consider Fig. 4. It shows the two release packages for component A'. They differ in the choice between revisions B and B'. Since a release package contains accurate revision information it is possible to compare a release package to an installed configuration and compute the difference between the current state (user configuration) and the desired state (a release package).

If ugrades are to be delivered automatically they have to satisfy a number of properties. We will discuss each property in turn and assert that the release packages derived from the RKB satisfy it.

Correctness Releases should contain software that is correct according to some criterion. In this paper we used integration testing as a criterion. It can be seen from the algorithm INTEGRATEMANY that only successfully integrated components are released.

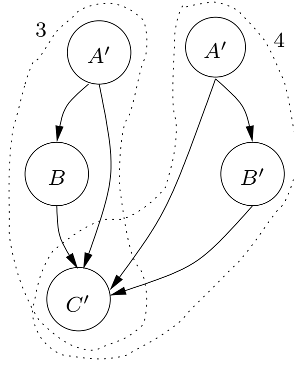

Fig. 4. Two release packages for A'

Completeness A component release should contain all updates of its dependencies if they are required according to the correctness criterion. In our component model, the source tree of each component contains a special file explicitly declaring the dependencies of that component. If a dependency is missed, the integration of the component will fail. Therefore, every release will reference *all* of its released dependencies in Integration.

Traceability It should be possible to relate a release to what is installed at the user's site in a precise way. It is for this reason that release version numbers are equated with revision numbers. Thus, every installed release can be traced back to the sources it was built from. Tracing release to source code enables the derivation of incremental updates.

Determinism Updating a component should be unambiguous; this means that they can be applied without user intervention. This implies that there cannot be two revisions of the same component in one release package. More formally, this can be stated as a knowledge base invariant. First, let:

$$\text{components}(s) = \text{domain}(\text{package}(s))$$

The invariant that should be maintained now reads:

$$\forall s \in \mathsf{Success} : |\text{package}(s)| = |\text{components}(s)|$$

We have empirically verified that our continuous release algorithm preserves this invariant. Proving this is left as future work.

4.2 Deriving Updates

The basic use case for updating a component is as follows. The sofware vendor advertises to its customers that a new release of a product is available [9]. Depending on certain considerations (e.g. added features, criticality, licensing etc.)

the customer can decide to update to this new release. This generally means downloading a package or a patch associated to the release and installing it.

In our setting, a release of a product is identified by a successful integration of a top component. There may be multiple releases for a single revision r due to the evolution of dependencies of r. The user can decide to obtain the new release based on the changes that a component (or one of its dependencies) has gone through. So, a release of an application component is best described by the changes in all its (transitive) dependencies.

To update a user installation one has to find a suitable release. If we start with the set of all releases ($\mathsf{Success}$), we can apply a number of constraints to reduce this set to (eventually) a singleton that fits the requirements of a user.

For instance, assume the user has installed the release identified by the first iteration in Fig. 3. This entails that she has component revisions A, B, and C installed at her site.

The set of all releases is $\{1, 2, 3, 4, 5, 6\}$. The following kinds of constraints express policy decisions that guide the search for a suitable release.

- State constraints: newer or older than some date or version. In the example: "newer than A". This leaves us with: $\{3, 4, 5, 6\}$.
- Update constraints: never remove, or patch, or a add, a certain (set of) component(s). For example: "preserve the A component". The set reduces to: $\{3, 4, 6\}$.
- Trade-offs: conservative or progressive updates, minimizing bandwidth and maximizing up-to-dateness respectively. If the conservative update is chosen, release 3 will be used,—otherwise 6.

If release 3 is used, only the patch between C and C' has to be transferred and applied. On the other hand, if release 6 is chosen, patches from B to B'' and A to A'' have to be deployed as well.

5 Implementation

We have validated our formalisation of continuous release in the context the Asf+Sdf Meta-Environment [17], developed within our group SEN1 at CWI. The Meta-Environment is a software system for the definition of programming languages and generic software transformations. It consists of around 25 components, implemented in C, Java and several domain specific languages. The validation was done by implementing a prototype tool called Sisyphus. It is implemented in Ruby[2] and consists of approximately 1000 source lines of code, including the SQL schema for the RKB.

In the first stage Sisyphus polls the CVS repository for changes. If the repository has changed since the last iteration, it computes the $\mathsf{Depends}$ relation based on the current state of the repository. This relation is stored in a SQLite[3] database.

[2] www.ruby-lang.org

[3] www.sqlite.org

The second stage consists of running the algorithm described in Sect. 3. Every component that needs integration is built and tested. Updates to the relations Attempt, Succes and Integration are stored in the database.

We let Sisyphus reproduce a part of the build history of a sub-component of the ASF+SDF Meta-Environment: a generic pretty-printer called `pandora`. This tool consists of eight components that are maintained in our group. The approximate size of `pandora` including its dependencies is ≈190 KLOC. The Sisyphus system integrated the components on a weekly basis over the period of one year (2004). From the database we were then able to generate a graphical depiction of all release packages. In the future we plan to deploy the Sisyphus system to build and release the complete ASF+SDF Meta-Environement.

A snapshot of the generated graph is depicted in Fig. 5. The graph is similar to Fig. 3, only it abstracts from version information. Shown are three integration iterations, 22, 23, and 24. In each column, the bottom component designates the minimum changeset inbetween iterations.

Iteration 22 shows a complete integration of all components, triggered by a change in the bottom component `aterm`. In iteration 23 we see that only `pt-support` and components that depend on it have been rebuilt, reusing the integration of `error-support`, `tide-support`, `toolbuslib` and `aterm`.

The third iteration (24) reuses some of these component integrations, namely: `tide-support`, `toolbuslib` and `aterm`. The integration of component `error-support` is not reused because it evolved in between iteration 23 and 24. Note that the integration of `pt-support` from iteration 23 cannot be reused here since it depends on the changed component `error-support`.

6 Related Work

6.1 Update Management

Our work clearly belongs to the area of update management. For an overview of existing tools and techniques we refer to [9]. Our approach differs from the techniques surveyed in that paper, mainly in the way how component releases and the updates derived from them are linked to a continuous integration process.

The package deployment system Nix [3] also automatically produces updates for components. This system uses cryptographic hashes on *all* inputs (including compilers, operating system, processor architecture etc.) to the build process to identify the state of a component. In fact this more aggressive than our approach, since we only use revision identifiers.

Another difference is that Nix is a generic deployment system similar to Debian's Advanced Package Tool [15], Redhat's RPM [1] and the Gentoo/BSD ports [14, 20] systems. This means that it works best if all software is deployed using it. Our approach does not prohibit that different deployment models peacefully coexist, although not across compositions.

Updates produced by Nix are always non-destructive. This means that an update will never break installed components by overwriting a dependency. A

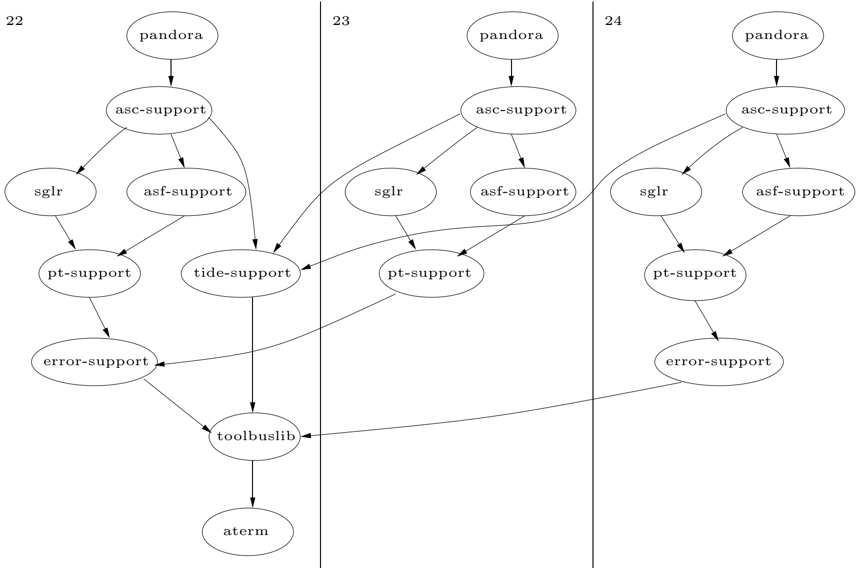

Fig. 5. Three weekly releases of the `pandora` pretty printing component in 2004

consequence of this is that the deployment model is more invasive. Our updates are always destructive, and therefore the reasoning needed to guarantee the preservation of certain properties of the user configuration is more complex. Nevertheless, this makes the deployment of updates simpler since no side-by-side installation of different versions of the same component is needed.

6.2 Relation Calculus

The relational calculus [16] has been used in the context of program understanding (e.g. [10, 12]), analysis of software architecture [8, 5], and configuration management [11, 2]. However, we think that use of the relational calculus for the formalisation of continuous integration and release is novel.

Our approach is closest to Bertrand Meyer's proposal to use the calculus for a software knowledge base (SKB). In [13] he proposes to store relations among programming artifacts (e.g., sources, functions) in an SKB to support the software process. Many of the relations he considers can be derived by analyzing software artifacts. Our approach differs in that respect that only a minimum of artifacts have to be analyzed: the dependencies between components that are specified somewhere. Another distinction is that our SKB is populated by a software program. Apart from the specification of dependencies, no intervention from development is needed.

7 Conclusion and Future Work

Proper update management can be a serious advantage of software vendors over their comptetitors. In this paper we have analysed how to successfully and quickly produce and deploy such updates, without incurring additional overhead for development or release managers.

We have analysed technical aspects of continuous integration in a setting of component-based development. This formalisation is the starting point for continously releasing components and deriving updates from it that are guaranteed to have passed integration testing.

Finally we have developed a prototype tool to validate the approach against the component repository of a medium-sized software system, the ASF+SDF Meta-Environment. It proved that the releases produced are correct with respect to the integration predicate.

As future work we will consider making our approach more expressive and flexible, by adding dimensions of complexity. First, the approach discussed in this paper assumes that all components are developed in-house. It would be interesting to be able to transparently deal with third-party components, especially in the context of open source software.

Another interesting direction concerns the notion of variability. Software components that expose variability can be configured in different ways according to different requirements [19]. The question is how this interacts with automatic component releases. The configuration space may be very large, and the integration process must take the binding variation points into account. Adding variation to our approach would, however, enable the delivery of updates for product families.

Finally, in many cases it is desirable that different users or departments use different kinds of releases. One could imagine discerning different levels of release, such as alpha, beta, testing, stable etc. Such stages could direct component revisions through an organisation, starting with development, and ending with actual users. We conjecture that our formalisation and method of formalisation are good starting points for more elaborate component life cycle management.

Acknowledgements Gratitude goes to Paul Klint, Jurgen Vinju and Gerco Ballintijn, who suggested important improvements to drafts of this paper. We thank the anonymous referees for providing many insightful comments.

References

1. E. C. Bailey. *Maximum RPM. Taking the Red Hat Package Manager to the Limit.* Red Hat, Inc., 2000. Online: `http://www.rpm.org/max-rpm` (August 2005).
2. E. Borison. A model of software manufacture. In *Proceedings of the IFIP International Workshop on Advanced Programming Environments*, pages 197–220, Trondheim, Norway, June 1987.

3. E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In Lee Damon, editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, Atlanta, Georgia, USA, November 2004. USENIX.
4. J. Estublier, J.-M. Favre, and P. Morat. Toward SCM / PDM integration? In *Proceedings of the Eighth International Symposium on System Configuration Management (SCM-8)*, 1998.
5. L. Feijs, R. Krikhaar, and R. van Ommering. A relational approach to support software architecture analysis. *Software Practice and Experience*, 4(28):371–400, April 1998.
6. M. Fowler and M. Foemmel. Continuous integration. Available at: `http://www.martinfowler.com/articles/continuousIntegration.html` (February 2005).
7. E. Grossman. An update on software updates. *ACM Queue*, March 2005.
8. R. C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, 1998.
9. S. Jansen, G. Ballintijn, and S. Brinkkemper. A process framework and typology for software product updaters. In *9th European Conference on Software Maintenance and Reengineering (CSMR)*, 2005.
10. P. Klint. How understanding and restructuring differ from compiling—a rewriting perspective. In *Proc. of the 11th International Workshop on Program Comprehension (IWPC03)*, pages 2–12. IEEE Computer Society, 2003.
11. D. A. Lamb. Relations in software manufacture. Technical report, Department of Computing and Information Science, Queen's University, Kingston, Ontario K7L 3N6, october 1994.
12. M. A. Linton. Implementing relational views of programs. In P. Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 132–140, Pittsburgh, PA, May 1984. Association for Computing Machinery, Association for Computing Machinery.
13. B. Meyer. The software knowledge base. In *Proc. of the 8th Intl. Conf. on Software Engineering*, pages 158–165. IEEE Computer Society Press, 1985.
14. FreeBSD Ports. Online: `http://www.freebsd.org/ports` (August 2005).
15. G. Noronha Silva. *APT HOWTO*. Debian, 2004. Online: `http://www.debian.org/doc/manuals/apt-howto/index.en.html` (August 2005).
16. A. Tarski. On the calculus of relations. *J. Symbolic Logic*, 6:73–89, 1941.
17. M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.
18. A. van der Hoek and A. L. Wolf. Software release management for component-based software. *Software—Practice and Experience*, 33(1):77–98, 2003.
19. T. van der Storm. Variability and component composition. In J. Bosch and C. Krueger, editors, *Software Reuse: Methods, Techniques and Tools: 8th International Conference (ICSR-8)*, volume 3107 of *Lecture Notes in Computer Science*, pages 86–100. Springer, June 2004.
20. S. Vermeulen, R. Marples, D. Robbins, C. Houser, and J. Alexandratos. *Working with Portage*. Gentoo. Online: `http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=3` (August 2005).