About the cover: picture of the SISYPHUS II machine, constructed by Bruce Shapiro. SISYPHUS II operates in a continuous fashion in the Science Museum of Minnesota's Learning Technology Center. Reproduced with permission.

Component-Based Configuration, Integration and Delivery

Component-Based Configuration, Integration and Delivery

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het
college voor promoties ingestelde commissie,
in het openbaar te verdedigen
in de Aula der Universiteit
op dinsdag 20 november 2007, te 10:00 uur

door

Tijs van der Storm

geboren te Veldhoven

Promotores: prof. dr. P. Klint, prof. dr. S. Brinkkemper
Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The work in this thesis has been carried out at Centrum voor Wiskunde en Informatica (CWI) in Amsterdam under the auspices of the research school IPA (Institute for Programming research and Algorithmics) This research was supported by NWO, in the context of Jacquard Project 638.001.202 “Deliver”.

Preface

After four years of doing research and finally finishing this dissertation, it is time to look back and thank all people that have made this possible. First of all, I would like to thank my first promotor, supervisor, and all round scientific guide, Paul Klint. Thanks for all the support and the freedom I was allowed to enjoy when doing this research.

Secondly, thanks go to my second promotor, Sjaak Brinkkemper. Paul and you both conceived the Deliver project that made this work possible in the first place. Thank you for many constructive comments and the incentive for looking at software from the "other" side.

There is no promotion without a committee. I thank all members for taking the time to examine this thesis, in alphabetical order: Prof dr. Pieter Adriaans, Prof. dr. Jan Bergstra, Prof. dr. Peter van Emde-Boas, dr. André van der Hoek, dr. René Krikhaar and dr. Eelco Visser. I also want to thank René in particular for our collaboration on the topic of software integration.

CWI proved to be an ideal environment for doing research. This is in great part due to some colleagues who I would like to thank here. Slinger was my direct co-worker on the Deliver project. How you managed to run you own company, follow management courses, do teaching, preside workshops, and still finish you PhD before me (even though you started later) still amazes me. Jurgen and I shared the love of programming languages and compiler technology, and although I could not readily apply these subjects in my research, we did manage to write a paper together. Thank you for many stimulating brain dump sessions. Then there is Magiel, roommate and scientific role model. We had numerous discussions about the state of software engineering. Rob, parsing guru and all time champion slow cooking (but with optimal results), was always willing to go out and have a beer. Thanks for many non-work related conversations as well as some excellent meals. Rob's roommate, Taeke always came to the rescue when I screwed up or had to type something in VI. Finally I enjoyed many stimulating discussions with Jan Heering (especially on the history of software, including LISP and Smalltalk), Jan van Eijck and Arie van Deursen. Thank you all for creating such a great atmosphere.

Outside work, my best friends kept me motivated and prevented me from taking things too seriously. I thank them in no particular order: Marcel, Thijs, Inge, Xavier, Arieke, Eugène. Finally, I want to thank Susanna, my love. I know how much you suffered. I have failed you many times,—yet you never gave up on me. I cannot express how much I owe you. Thank you.

Contents

I Overview **1**

1 Introduction **3**
1.1 Towards Continuous Delivery 3
1.1.1 Introduction 3
1.1.2 Motivation 5
1.2 Related Research Areas In Software Engineering 8
1.2.1 Software Configuration Management 8
1.2.2 Component-Based Software Engineering 10
1.2.3 Product Line Engineering 11
1.2.4 Release, Delivery and Deployment 11
1.2.5 Continuous Delivery 12
1.3 Research Perspective 14
1.3.1 Goals and Requirements 14
1.3.2 Research Questions 18
1.3.3 Summary of Contributions 19

II Configuration **21**

2 Variability and Component Composition **23**
2.1 Introduction 23
2.2 Towards Automated Management of Variability 24
2.2.1 Why Component Variability? 24
2.2.2 The Software Knowledge Base 25
2.3 Degrees of Component Variability 25
2.4 Component Description Language 26
2.5 Guaranteeing Consistency 28
2.5.1 Transformation to Relations 28
2.5.2 Querying the SKB 29
2.6 Discussion 30
2.6.1 Related Work 30
2.6.2 Contribution 31
2.6.3 Future Work 32

3 Generic Feature-Based Composition 33
3.1 Introduction . . . 33
3.1.1 Problem-Solution Space Impedance Mismatch . . . 34
3.1.2 Related Work . . . 35
3.1.3 Contributions . . . 35
3.2 Problem and Solution Space Models . . . 37
3.2.1 Introduction . . . 37
3.2.2 Problem Space: Feature Diagrams . . . 37
3.2.3 Solution Space: Implementation Artifacts . . . 37
3.2.4 Artifact Dependency Graphs . . . 38
3.3 Mapping Features to Artifacts . . . 39
3.3.1 Introduction . . . 39
3.3.2 Feature Diagram Semantics . . . 40
3.3.3 Configuration Consistency . . . 40
3.4 Selection and Composition of Artifacts . . . 41
3.4.1 Introduction . . . 41
3.4.2 Configuration and Selection . . . 42
3.4.3 Composition Methods . . . 43
3.5 Conclusions . . . 45
3.5.1 Discussion: Maintaining the Mapping . . . 45
3.5.2 Conclusion & Future Work . . . 45

III Integration & Delivery 47

4 Continuous Release and Upgrade of Component-Based Software 49
4.1 Introduction . . . 49
4.2 Problem Statement . . . 50
4.2.1 Motivation . . . 50
4.2.2 Solution Overview . . . 51
4.3 Continuous Release . . . 52
4.3.1 Component Model . . . 52
4.3.2 Towards Continuous Release . . . 53
4.3.3 A Sample Run . . . 56
4.4 Continuous Upgrade . . . 57
4.4.1 Release Packages . . . 57
4.4.2 Deriving Updates . . . 58
4.5 Implementation . . . 59
4.6 Related Work . . . 60
4.6.1 Update Management . . . 60
4.6.2 Relation Calculus . . . 61
4.7 Conclusion and Future Work . . . 62

5 Backtracking Continuous Integration 63
5.1 Introduction . . . 63
5.2 Background . . . 64
5.2.1 Component-Based Development . . . 64
5.2.2 Continuous Integration . . . 65
5.2.3 Motivation: Continuous Release . . . 66
5.3 Overview . . . 67
5.3.1 Introduction . . . 67
5.3.2 Build Sharing . . . 68
5.3.3 Simple Backtracking . . . 69
5.3.4 True Backtracking . . . 70
5.4 Formalization . . . 71
5.4.1 Preliminaries . . . 71
5.4.2 Schematic Integration Algorithm . . . 73
5.4.3 Incremental Continuous Integration . . . 74
5.4.4 Backtracking Incremental Continuous Integration . . . 75
5.5 Evaluation . . . 78
5.6 Related Work & Conclusion . . . 81
5.6.1 Related Work . . . 81
5.6.2 Conclusion . . . 82

6 Techniques for Incremental System Integration 83
6.1 Introduction . . . 83
6.2 Component-Based Integration . . . 85
6.2.1 Introduction . . . 85
6.2.2 Integration of Subsystems in Practice . . . 86
6.2.3 Integration Conflicts . . . 87
6.2.4 Causes of Integration Conflicts . . . 88
6.2.5 Resolving Conflicts . . . 89
6.2.6 Solution Overview . . . 90
6.3 Towards Automating Integration . . . 91
6.3.1 Introduction . . . 91
6.3.2 Bill of Materials . . . 91
6.3.3 Integration . . . 92
6.3.4 Resolving Conflicts . . . 92
6.3.5 Minimum Build Penalty . . . 93
6.3.6 Evaluation . . . 94
6.3.7 Parsed Lines of Code (PLOC) . . . 94
6.3.8 PLOC Build Penalty . . . 96
6.3.9 Evolutionary PLOC . . . 96
6.4 Discussion: Compatibility . . . 97
6.4.1 Introduction . . . 97
6.4.2 Compatibility as SCM Relation . . . 97
6.4.3 Requirements of SCM Systems . . . 99
6.4.4 Weakening the Rebuild Criterion . . . 100
6.5 Conclusions . . . 100

6.5.1 Related Work . . . 100
6.5.2 Conclusion . . . 101

7 Binary Change Set Composition 103
7.1 Introduction . . . 103
7.2 Background . . . 104
7.2.1 Requirements for Application Upgrade . . . 104
7.2.2 Related Work . . . 105
7.2.3 Overview of the Approach . . . 107
7.3 Binary Change Set Composition . . . 108
7.3.1 Incremental Integration . . . 108
7.3.2 Build and Release Model . . . 109
7.3.3 Prefix Composition . . . 109
7.3.4 Change Set Delivery . . . 110
7.3.5 Change Set Composition . . . 111
7.4 Implementation using Subversion . . . 112
7.4.1 Composition by Shallow Copying . . . 112
7.4.2 Upgrade is Workspace Switch . . . 114
7.4.3 Techniques for Relocatability . . . 114
7.5 Evaluation . . . 115
7.5.1 Experimental Validation . . . 115
7.5.2 Release Management Requirements . . . 115
7.5.3 Update Management Requirements . . . 116
7.6 Conclusion and Future Work . . . 116

8 The Sisyphus Continuous Integration and Release System 119
8.1 Introduction . . . 119
8.2 Functional Requirements . . . 120
8.2.1 Component-Based . . . 120
8.2.2 Platform Independent . . . 121
8.2.3 Configurable . . . 121
8.2.4 Traceable . . . 122
8.3 Architectural Overview . . . 122
8.3.1 Introduction . . . 122
8.3.2 Deployment Architecture . . . 123
8.3.3 Dynamics . . . 125
8.4 Build Configuration . . . 126
8.4.1 Global Configuration . . . 126
8.4.2 Per Builder Configuration . . . 128
8.5 Data Model . . . 128
8.6 Details About the Implementation . . . 130
8.6.1 Front-End: Web Of Sisyphus . . . 130
8.6.2 The Size of Sisyphus . . . 136
8.7 Related Work . . . 136
8.7.1 Similar Tools . . . 136
8.7.2 Discussion . . . 138

8.8 Conclusion & Future Work 139
8.8.1 Future Work 1: Branch Support 139
8.8.2 Future Work 2: Federated Integration 140

IV Conclusions 141

9 Summary and Conclusions 143
9.1 Configuration 143
9.2 Integration & Delivery 145

Acronyms & Used Notation 161

Nederlandse Samenvatting 163

Part I

Overview

Chapter 1

Introduction

1.1 Towards Continuous Delivery

1.1.1 Introduction

Software evolves all the time. In a world where software is getting more and more intertwined with our daily lives, constant availability and maintenance becomes more and more important. Additionally, the constant threat of malware (viruses, worms etc.) requires the frequent repair and adaptation of software that may have been deployed to many users. Especially in the context of "software as a service", the frequent modification and repair of software systems presents new opportunities and challenges.

Of course, for such fast-changing software to have any real benefit, the changes to the software will have to be propagated to the actual users of the system. After a feature has been added to a software product, the new version has to be delivered to the users of the system. This thesis is concerned with the technical and conceptual aspects of this process which is usually called the update process [96]. The notion of *software delivery* is part of that process. It is defined as transferring changes made to a software product at the vendor side to the customer side where they can be deployed.

Recently, the notion of "software as a service" (SaaS) has raised considerable interest. In this case the larger part of the software is operational at the vendor side. Customers use the software over the Internet, often through a web interface. One of the advantages of SaaS from the point of view of software delivery, is that this process can be completely managed by the vendor. Moreover, upgrading a fixed number of instances of in-house deployments is enough to make new features and bug fixes available to all users at once. An example of this is Google Mail (GMail). This software product is continuously evolved without users having to download and install explicit updates. They only notice feature enhancement and quality improvement. However, there are also challenges. If for instance the software service is realized by a server part on the one hand (running at the vendor side) and a client part on the other (installed by the customer), updates to either of them have to ensure that the other does not break. In other words the server and client deployments co-evolve [47]. This kind of setup is exemplified by Google Earth which can be used to "browse" the earth. The maps and

satellite images are hosted and served by Google. However, the user accesses these images through a user interface client that has to be downloaded and installed separately. The impact of service orientation and software evolution on society requires further research on the question how to keep the customer side synchronized to the vendor side. The work in this thesis can be seen in that perspective.

This thesis in particular looks at the problem of automating software delivery in the context of heterogeneous, semi-large to large component-based product lines [107,108] with an emphasis on open source software (OSS). The emphasis on OSS entails that we abstract from certain organizational boundaries and process aspects that are common in industrial settings. Examples of these aspects include formal code reviewing, quality assurance, release planning, etc.

Heterogeneity of a software product means that the product is developed using multiple programming languages and/or platforms. A component-based product line then consists of a set of independently evolving software components reused across different variants of the product.

Heterogeneity of a software product adds to the complexity of the software delivery problem. Different programming languages may require different configuration, build, integration, and deployment solutions. This means that automation of delivery in such a setting requires abstract models of how a software product is structured. In this thesis we abstract from programming language and platform specific details, and present our solutions in mostly generic terms.

Ideally software delivery should be performed in a continuous fashion, hence the theme of this thesis: "continuous delivery". Continuous delivery is motivated by the fact that a shorter feedback loop between vendor and customer increases the quality and functionality of the product. We discuss the (historical) roots of continuous delivery in more detail below.

Our approach to continuous delivery does not imply that the updates are *actually* delivered to end users of a software product. However, if updates *can* be delivered continuously, there is always the possibility to deliver them in a less frequent fashion. Nevertheless, different groups within a software development organization may benefit from the ability to be flexible in this respect. For instance, our technology does not prohibit schemes where, for example, end users receive updates every month, whereas beta testers receive them every week. Developers themselves could update truly continuously to immediately evaluate the result of the changes they made. This thesis presents technological enablers for the practice of continuous delivery; how the actual process of release and update is organized is considered to be an orthogonal aspect.

The research covers three aspects, corresponding to the three parts of this thesis. These aspects are considered to be enablers for the update process: configuration, integration and delivery. Configuration consists of customizing a software product family by selecting the desired set of features and components. This way vendors are able to satisfy the requirements of a diverse customer base. Integration is defined as putting independently evolving software components together in order to obtain a complete, working software system. Finally, software delivery consists of transferring new versions of the product to its users.

1.1.2 Motivation

Introduction

A recent issue of *ACM Queue* was dedicated to the importance of automatic updates as a critical competitive advantage for software vendors [43]. One of the open problems discussed in that issue concerned the automatic updating of component-based or otherwise composite systems. Since software products nowadays are often *systems of systems* and not just a singular *program*, release, delivery and deployment processes are more complex, time-consuming and error-prone. In this section we give a high level description of the software update process and motivate the need for automation.

The Software Update Process

Software delivery is defined as transferring software from a vendor to a customer. Software updating usually means that an existing customer configuration is *updated* to a new version of the product. Delivery, however, also includes the transfer of a software product that has not been installed by the user previously.

During the evolution of a software system, usually not all subsequent versions are delivered. Only versions that exhibit sufficient levels of quality or feature completeness are usually made available to customers. This process is called *releasing*. Releasing consists of selecting the appropriate versions of software components that together make up a working, high-quality product that is ready to be exposed to end-users.

Before a software product is released it often is built from scratch and thoroughly tested. *Building* a software product consists of compiling all the sources and linking them into one or more libraries and/or executables. The testing process then makes sure the current state of the product meets the requirements and exhibits certain quality attributes (e.g., reliability, efficiency, etc.).

Figure 1.1 gives a graphical overview of this process. On the right a customer is shown that has installed version 1.0 of a certain software product. This installation is subsequently updated to versions 1.1 and 1.2. The enhancements in quality and functionality are indicated by triangles (deltas). The left-hand side of the figure shows the vendor perspective: for each release (indicated by the boxes in the middle) the sources go through a process of building, testing and releasing before the actual release packages are made available.

Obviously, the figure does not show the whole picture. It does not show, for instance, how the software product is structured, nor does it show how the build, test and release processes are affected by this structure. For instance, if a software product consists of a number of subsystems that evolve at individual paces, building, testing and releasing new versions can be quite time-consuming and error-prone. Furthermore, the figure only lists three subsequent releases of a single product and how each of these releases is delivered to a single user. In realistic settings, however, there are probably more versions, more product *variants* and more customers. Additionally, the acquisition of a software product may involve a process of tailoring and customization, which leads to the situation that different customers have installed different instances of the same release. Below we will discuss the dilemmas and trade-offs that come into play if the delivery and update processes have to deal with such multiplicities.

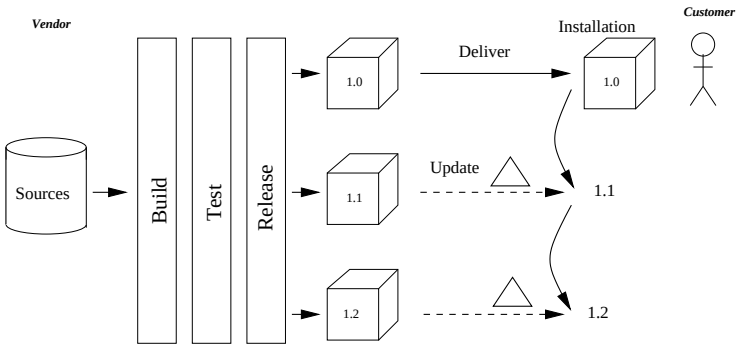


Figure 1.1: Graphical overview of the software update process

Many, Many, Many...

Software products are no longer developed as monolithic systems. Many software products consist of a host of subsystems, third-party components, tools, plug-ins, addons, etc. Some components of the system may be developed in house in one programming language whereas another component is licensed from a third-party and is developed in a different programming language. Software development has become increasingly complex. Similarly, a product often has dependencies on additional software at the user's site apart from the core services of the operating system. These problems are especially severe in the context of product software where many customers have to be served with new versions of the product [54], and where software has many external dependencies, such as web servers, database servers, virtual machines etc.

For instance, a content management system (CMS) often has to integrate with some kind of database server such as MySQL or Microsoft SQL Server at the customer side. Updates to the CMS might require or break certain versions of the database server that the customer acquired from a different vendor. Additionally, the CMS may support different database servers. Thus, different customers can have different software environments which complicates the update process even more so. Finally, since each customer has her specific configuration, customization and setup data, the update process must take care not to overwrite or invalidate such stateful information. In other words, updates should be backwards compatible with respect to state, or be able to migrate existing configuration data to the new version.

Different parts, originating from different parties, delivered in multiple configurations to many different customers who possibly maintain diverse environments. In this thesis we restrict ourselves to delivery from a single vendor. We note however that our delivery model can easily be extended to support multiple vendors via a process of *intermediate* integration. In addition to these structural multiplicities, we assume that software systems are increasingly developed using multiple technologies, i.e. systems are increasingly heterogeneous. Different programming languages and different platforms are used to develop such system of systems. The CMS is a case in point: parts of it may have been written in, for instance, PHP, C, SQL, HTML, Javascript and so on.

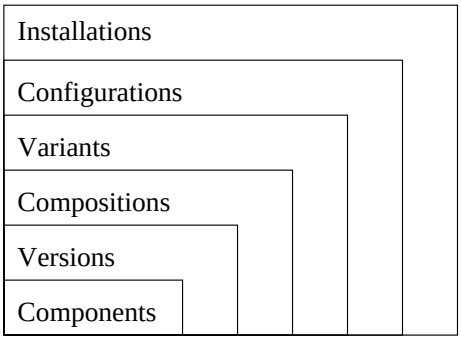


Figure 1.2: Dimensions of complexity in software Delivery

For the remaining of this thesis we therefore make no assumptions on how a certain component is implemented.

In this thesis we thus restrict ourselves to the setting of component-based development. This means that a product family consists of a set of components interrelated using a certain notion of composition. This leads to dependency graphs where each node represents a component and the edges represent dependencies. A product variant is represented by a root in such a graph. We then identify the following dimensions that complicate the process of software delivery and hence require the automation of this process:

- Multiple components, subsystems or parts: each part may have its own evolution history and development life-cycle and may thus be released as a separate entity.
- Multiple versions or revisions: every part evolves and as a consequence every part exists in multiple (sequential) versions; different components may evolve at a different pace.
- Multiple compositions: the architecture of a system may change so that different versions of components may have to be composed in different ways.
- Multiple product variants: different parts may be shared among different instances of a product family; the release of a new version of one part may induce a new release of certain variants.
- Multiple configurations: released products may be customized in different ways before they are delivered to customers.
- Multiple installations: different customers may have different versions and/or variants installed; the update process must take a heterogeneous customer base into account.

The multiplicity of components, versions, compositions, variants, configurations and installations makes the software delivery process increasingly complex. This is shown

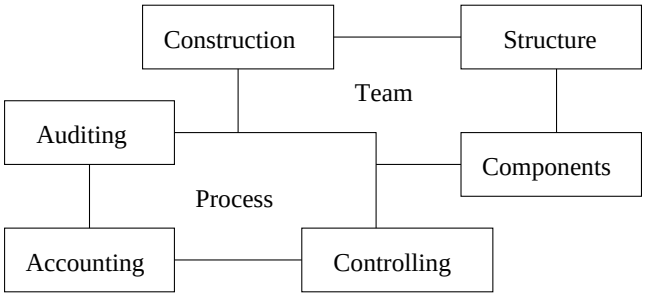


Figure 1.3: Concepts in Software Configuration Management

as the software delivery complexity tower in Figure 1.2. Starting from the lower left-hand corner, the fact that a software product is actually a composition of components is the first additional layer of complexity. Moving upwards, each layer adds a complexity dimension to the layer below.

The *more* we factor our software product in individual components in order to foster reuse, control complexity and reduce maintenance effort, the *more* versions there will be. The *more* versions there are, the *more* different ways of composition and integration there are since the architecture of the system evolves as well. Similarly, if we want to serve *more* customers, we will have to support the updating of *more* installations, which in turn requires updating *more* variants and configurations.

In other words, the goal of continuous delivery in the context of component-based software product lines leads to a large configuration space that clearly cannot be managed without automation: selecting the right versions of the right components after every change for every product variant, then building each variant, publishing a release for each variant and finally delivering the new versions of the appropriate variants to the corresponding customers can be a daunting task indeed. And this does not even consider constructing efficient incremental update packages from one version to another.

1.2 Related Research Areas In Software Engineering

Software engineering is about the efficient design and construction of software. Our work can be positioned at the intersection of several sub disciplines of software engineering: software configuration management (SCM), build management, component-based software engineering (CBSE), product line engineering (PLE) and software release management (SRM). Below we present short surveys of each field.

1.2.1 Software Configuration Management

An important aspect of large-scale software development is the fact that such software is not developed by one individual programmer but by teams of programmers. In

order to increase productivity, development efforts are parallelized. However, teams must coordinate their activity because they have the shared goal of delivering a quality software product. This is where Software Configuration Management (SCM) comes in [4, 12, 13, 23, 35, 64]. SCM is defined as the set of practices, tools and techniques to enable multiple (teams of) programmers to work at the same time on the same software without getting in each other's way.

SCM is used to manage the evolution of large, complex software products. The pinnacle of SCM is that it should at all times be known what the state of the system is. For instance, most version control systems (VCS) maintain accurate version histories for the different source files that make up a software system in development. Every change to the sources leads to a new version. At every stage in the development cycle the system can be identified using a version identifier. Although this example might seem simplistic, the identification of the system under development enables accurate tracking of progress and quality.

Dart [23] distinguishes two essential aspects of SCM: the process aspect and the team aspect. Figure 1.3 displays these two sides of the medal[1]. The team side of SCM has to do with how (teams of) programmers evolve a software system and how these activities are traced throughout the life-cycle. In order to trace development activity an SCM system should be knowledgeable about the parts of the system (components), how these parts are interrelated (structure) and how these parts are developed (construction). In this case construction includes how particular components are built from sources.

The team aspect binds construction, components, and structure together in the feature that any SCM system provides, namely the creation of private workspaces. To isolate programming activity of a single programmer, a local copy is made for the sources the programmer will be working on. The programmer makes the required changes and checks them in afterward. Which files will be in the workspace and how the programmer can change, build and test this particular part of the software system derives from how the system is structured.

The other side to SCM is the process view, which is about auditing, accounting and controlling the evolution of the system. Of prime importance here, is traceability. Traceability requires that both the software system and the system state can be reproduced for each moment in the system's history. This might, for instance, include source files and documentation, as well as information on what changes have been applied in a certain version, and which programmer was responsible for them. Auditing is about validating the completeness and consistency of the software product. The accounting part of the process is concerned with generated statistics and general status reports, for instance, on which milestones have been reached, which change requests are still open, or how many bugs have been fixed during a certain period of time. Finally, controlling the process of software development is about bug tracking, prioritizing change requests, and assigning tasks.

Our work on Integration, Part III, can be seen as an extension of SCM. In order to be able to release and deliver compositions of components we must ensure that releases are properly versioned. Without the ability to trace a release back to the sources, it is impossible to reproduce any problems found in the field. In particular, the Software

[1] This picture is derived from [23].

Knowledge Base (SKB) that is maintained by the Sisyphus Continuous Integration and Release system can be seen as an additional layer of versioning on top (or in addition to) ordinary SCM systems, such as Subversion [19] or ClearCase [10].

1.2.2 Component-Based Software Engineering

Advocates of component-based software engineering propose that a software system is decomposed into individual, reusable *components*. These components differ from ordinary modules in that they have an independent software life-cycle; that is, they are units of versioning, acquisition, third-party composition and deployment [87].

Components are not to be confused with objects although many component models, such as COM [84], presuppose object-oriented programming languages. In particular, components differ from objects in that they maintain no state at runtime and components do not inherit from each other.

Components may have explicitly specified context-dependencies. These may include the services provided by other components. Of course, a component can be required again by other components; composition thus has a recursive nature. The specification of dependencies is parametrized in the concrete implementation of the required functionality. In other words, if a component *A* has the explicitly specified dependency on component *B* (i.e. *A requires B*), "*B*" really refers to some abstract interface that is *provided* by some, as yet unidentified, component. There may be multiple (versions of) components implementing the interface required by *A*.

Parametrized dependencies enable variation because multiple component may implement the same interface. This means that a client component requiring the services declared in that interface can be composed with different components implementing this very same interface. More generally speaking, the use of provides and requires interface enables the replacement of concrete component implementations without affecting the operation of clients. This also benefits the smooth evolution of components because they can be upgraded independently (which is essential when component originate from different vendors).

The primary case-study used to validate our work, the ASF+SDF Meta-Environment [92], is developed in the style of package-based development [27]. This means that every source component has an abstract build interface—based on the GNU autotools [1]—and explicitly specifies its dependencies. Although no distinction is made between provides and requires *interfaces*—due to the heterogeneity of the components—the Meta-Environment can still be called a component-based product line. Every component can be released separately or as a closure. Certain components are used outside the scope of the Meta-Environment itself and are composed by third parties.

In this work, software components and their dependencies are first-class concepts during integration. The Sisyphus system (described in Chapter 8) inspects the source trees of the components in order to determine what their dependencies are (they are specified in a dedicated file). This knowledge is exploited to release closures for subsystems of the Meta-Environment as well as the Meta-Environment itself.

1.2.3 Product Line Engineering

In product line engineering [77, 106] the focus is on exploiting reuse and variability to develop different variants of a product using a shared code base. Often this involves component orientation [108] but this is not necessarily so. The notion of variability plays a crucial role here. Variability basically entails that a software product can be changed before delivery without having to change its source code. Product variants are instantiated by *configuring* the product line.

Product line engineering as a research field is closely related to domain engineering. One of the seminal papers in this field by Kang *et al.* [57] introduced a graphical notation for describing and analyzing variability: feature diagrams. Feature diagrams concisely capture commonality and variability in a certain problem domain. Such analyses can steer the software engineering process by indicating opportunities for reuse.

The first two chapters of this thesis view feature diagrams (in a textual form, called Feature Description Language) as a way of specifying configuration interfaces. Feature descriptions are given a formal, logic based semantics in order to check the consistency of such descriptions and to check whether feature selections ("configurations") are valid with respect to the description.

The first of the two chapters, *Variability and Component Composition* moreover, links feature descriptions to models of composition. This is motivated by the fact that selection of certain features may influence composition if components themselves are units of variation (e.g., one interface may be implemented by several components).

The second chapter, *Generic Feature-Based Composition* takes a more generic approach to feature models and composition. Any software artifact is considered to be a component that has dependencies on other components. A technique is proposed that maps every valid feature selection to certain selection of artifacts. This bridge between problem space (the domain of features and requirements) and solution space entities (i.e. components, files, programs etc.) can be checked for consistency using the semantics of Chapter 2.

1.2.4 Release, Delivery and Deployment

Research on release, delivery and deployment has only recently taken off [18,31,33,45, 46,55,96–98]. Generally speaking, *release* means making a particular piece of software available to its users. Deployment in general is about getting a piece of software to its users, installing it and activating it. Deployment thus may include transferring the software from the vendor to the user; however, in our work this falls under the heading of delivery.

Figure 1.4 shows an overview of the deployment life cycle[2]. The arrows indicate ordering constraints on the constituting process. The deployment life cycle of a software component starts when it is released. After it has been released it can installed by users. Then, finally is has to be activated and the software is ready to be used. Of course, a software component that is activated can de-activated, and subsequently be de-installed. Similarly, a software vendor may withdraw a release from support which is called retiring a release or terminating a product. While the software is installed

[2] This is a simplified version of the deployment activity diagram in [18].

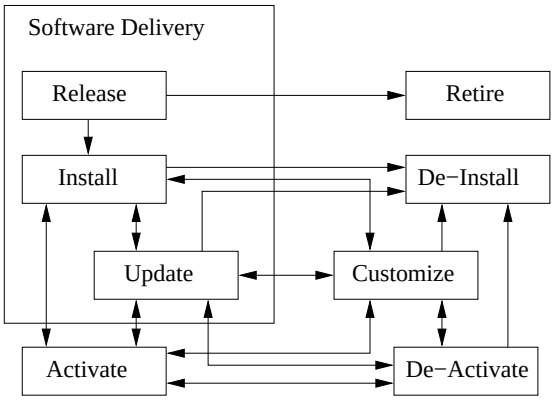


Figure 1.4: Software deployment life cycle

and/or activated the software can be updated (this may require de-activation) and customized. Customization in this context means any kind of tailoring, adapting, or changing, performed at the user's site. The processes of software delivery are enclosed by a larger rectangular area; it covers (parts of) releasing, installing, and updating a software product.

The focus of this thesis corresponds to the enclosed area in Figure 1.4. This thesis, however, is not concerned with the process of installation and activation per se. Although Part III of this thesis touches upon deployment issues, it does not describe a comprehensive approach to deployment. Our approach of binary change set composition described in Chapter 7 basically stops after the new version of a software product has been transferred to a user; what happens after that is up to the development organization of the product.

1.2.5 Continuous Delivery

Continuous delivery can be traced back to pioneering work by Harlan Mills, Tom Gilb and others in the seventies and eighties [6, 40, 41, 75]. Their work presents an alternative to the water-fall software process called evolutionary delivery. It can be described as an incremental and iterative mode of development where a *working program* is delivered to actual end-users from the very beginning, in very small, incremental steps. Note the emphasis on "working program". This means that it is not a prototype version of the end product that is delivered; the end-product exists from the very first increment. Keeping the current version working is a constant goal since that is the way that developers learn best about how their efforts live up to the requirements of the customer.

Tom Gilb's evolutionary delivery has had a strong influence on agile approaches to software development such as Extreme Programming (XP) [9]. In the introduction of *Extreme Programming*, Kent Beck explains that XP originated from looking at all the things that improve the quality of programming and then for each of those practices

turning the knob to ten. Two of those practices are of primary concern in the context of this thesis:

- Focus on small and frequent releases. This is motivated by two reasons: first user feedback is both more immediate and more accurate so development activity will not diverge too much from the customer's requirements. Second, updating to a small new release is less invasive and easier to revert in case of errors than large updates.

- Continuously integrate changes to the system, i.e. commit changes to the version control system (VCS) as fine-grained as possible and subsequently build and test the complete system. If the build fails, fixing it has top priority.

Both practices focus on shortening the feedback loop. In other words, however small a certain change to the software system may be, the time between commit and feedback should be as short as possible. This concerns on the one hand the feedback generated by automated builds and tests, and on the other hand the feedback generated by users (e.g. bug reports, crash reports etc.).

Postponing the integration and/or release increases the risk that different changes (even if committed to entirely different parts of the system) will interact, causing possible problems to be increasingly hard to track down and fix. In the presence of parallel development the complexity penalty of interacting changes is exponential, i.e postponing integration is exponentially more expensive in the long run than doing it right away. Another way of putting this is that "big bang integration" (all changes at once) leads to "integration hell"[3].

Our focus on continuous delivery can furthermore be put in the context of the recently proposed paradigm of *continuous coordination*. Redmiles *et al.* [83] claim that in a world where software development activities are increasingly globalized there is a need for collaboration infrastructures that create more awareness among geographically dispersed team developers. Moreover, such distributed development infrastructures should provide flexible ways of continuously integrating the produced artifacts. Continuous delivery extends this loop of continuous coordination with actual users.

The primary assumption that permeates this thesis is that continuous delivery is a desirable ideal. If we can deliver after every change, we can deliver any time we want. We set the bar high. This platonic goal effectively disposes of the question as to *what* (which version and/or changes) will be delivered and *when* (weekly, monthly, yearly); we basically assume that we deliver *everything*, and *as soon as possible*. Nevertheless, our work does not preclude any formal release process that is based on explicit selection of versions of configurations, but instead sees such processes as additional infrastructure on top of the techniques that are presented here.

[3] See `http://c2.com/xp/IntegrationHell.html` for more information.

1.3 Research Perspective

1.3.1 Goals and Requirements

The primary objective of this thesis is automating continuous delivery of component-based software. The requirements for realizing this goal can be understood by "reasoning backwards" from a running application back to the sources that were used to build it. Consider a prototypical customer who plans to acquire a certain piece of software. What are the typical steps she undertakes?

Assuming that the software package our prototypical customer is about to acquire is offered on the Web, the first step probably involves selecting the desired version and variant of the product. As an example, the product might be available in a light version, a professional version, and a server version, and possibly she has to choose the appropriate platform too. If the desired variant has been selected, the site could provide the customer with some configuration options, for instance in order to enable or disable certain features or add-ons that may make a difference in the final price. After this she may pay for the software using her credit card.

If an appropriate version of the product is chosen and the configuration and payment processes have been completed successfully, our customer is redirected to a download link to transfer the software to her computer on which she can then unpack and install the software.

Now assume that the vendor in question would provide an automatic update feature to upgrade the software every time a new release is published on the vendor's site. Assume moreover that the vendor implements this feature in such a way that every day new versions get published and the customers of the software can upgrade every single day by downloading and automatically installing new versions. What would the technical and customer requirements be for such a scenario?

Incremental Updates

First of all, the update process must be efficient so as to not annoy the customer too much with long download or redeployment times. Furthermore, the updates must not break the correct operation of the new version nor of any other software. This means that updates are insulated; they should not affect anything apart from the installed product version they are applied to. If the update *should* fail, which should be prevented at all cost, roll-back to the state previous to the update should be quick and painless. Thus, continuous delivery from the perspective of the customer should ideally be completely invisible apart from the improved functionality or quality. If applying updates is slow and brittle, there is no sense in delivering them continuously since they are an impediment for customer satisfaction.

From the customer perspective, continuous delivery should be fast and safe. How does this influence the software development, release and delivery processes of the vendor? A direct corollary of the customer requirements is the requirement that delivery should be implemented efficiently and safely. One way to efficiently implement updates is to apply *incremental* updates. Incremental updates work by transferring only the difference between the old and the new configuration. This is a great band-width

saver, especially for companies with large customer bases. Note also that this gives the "undo" update for free since the changes between old and new can be reversed, i.e. undo is "upgrading" (downgrading) from new to old.

Traceability

So incremental updates are the first technical requirement. Secondly, we observe that in order to update a customer configuration in an incremental fashion, the vendor must relate the exact version of the customer to the (newer) released versions on the website. Without knowing *what version* is installed at the customer's site, it is not possible to incrementally update the old configuration to the new configuration, because it is impossible to compute the difference between the two. More specifically, this means that accurate traceability links must be maintained. Additionally, not only must the vendor know the exact version of the installed product, but also which variant (pro, light, Windows, Linux etc.) and which configuration parameters were used for this particular installation. In other words, the vendor must maintain an accurate *bill of materials* (BOM) for every release and for every download/installation. The concept of *bill of materials* originates from the manufacturing industry and is defined as a document describing how a certain product is made [70]. BOMs should accurately capture the identity of the customer configuration including and most importantly its version. BOMs can be realized either through meta-data within the software product or through a vendor-site knowledge-base that maintains the update history of a particular customer.

As an aside, such traceability is also essential in linking customer feedback to the sources that made up that particular installation. For instance, if a customer files a bug report, it is of no use to the developers if they do not know in which version and configuration the bug surfaced, especially because there may be many different versions and configurations in the field. Without the exact version and configuration information it is hardly possible to accurately reproduce the bug which is a primary requirement for fixing it.

Automatic Release

A third technical requirement is that releasing a new version should be automatic. Recall that we assume that the vendor in the example implements continuous delivery. Therefore, in principle, after every change to the sources that has been approved for release, a new release package should be put on line so that customers can benefit from the improvements to the product. Clearly, performing a release after every change manually is a very time-consuming and error-prone process; it clearly does not scale. A tool is therefore required that monitors the changes approved by quality assurance (QA) and prepare a new release package for each of those changes, as soon and quickly as possible. Obviously to satisfy the second requirement—traceability—these releases should be versioned and configurable in such a way that installations can be linked to the originating sources.

Consistent Configuration

Finally, the vendor should ensure that the configuration of products by customers is done consistently. Customers should be able to make consistent feature selections or customizations. Moreover, if a customer makes a valid feature selection, the desired product instantiation should be created automatically, i.e. without human intervention from the side of the vendor. This means that the vendor must ensure that every valid configuration of the configuration interface corresponds to a valid product instantiation, and that the appropriate binding of variation points in the software takes place. For instance, the selection of certain features might cause certain values to be set in a configuration file in the release of the product. Every abstract feature selected by the customer should have a consequence for the package that is finally obtained by the customer. A specification of the set of enabled features, should also appear as part of the release notes in the BOM of the product. This fourth requirement—automatic configuration—is again motivated by the fact that if continuous delivery is the goal, manual intermediate binding steps when configuring a product do not scale.

Automatic Composition

Until now we have looked only from the outside at the problem of continuous delivery in the sense that we did not make any assumptions about the software development process employed by our example software vendor. We will now complicate the picture by making some assumptions about the software development process of the vendor. We will adapt the requirements where needed along the way.

Let's assume that the vendor is a proponent of component-based development [87]. This basically entails that the source base of the vendor is divided in a number of independently evolving components. Each component has its own life-cycle, which means that the version history of a component is independent from other components. Nevertheless, a component is hardly ever deployed in isolation. Components often require other components to operate correctly. These are the *dependencies* of a component.

The dependencies among components lead to composition graphs where the nodes of the graph are the components and the edges the dependencies. We assume that composition graphs are acyclic. We further assume that every product variant corresponds to a component in this graph and that such a variant is released and delivered as a composition of the component itself including all its (transitive) dependencies. Thus, every product variant is identified with its closure. In theory, every component in the composition graph represents a product (one can take the closure of any component in the graph), but obviously not all components represent *useful* products.

As an example of how component orientation is leveraged for the delivery of product variants, consider Figure 1.5. The nodes of the graph represent software components and the directed edges are the dependencies between components. So, for instance, component *A* has dependencies on *B* and *C*. On the other hand, components *C* and *D* have no dependencies (called *bottom* components).

The decomposition of a software system in separate components is primarily of interest to the vendor of the system. Component orientation create opportunities for reuse and variation. For instance, in the example of Figure 1.5, the components *A*, *B* and *C*

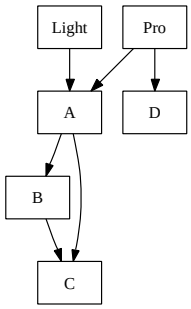


Figure 1.5: “Pro” and “light” product variants implemented using components

are reused (shared) among both the “light” and “pro” version. This kind of reuse is an enabler for product diversification which can be used to better satisfy market demands. From the process perspective, component orientation has additional benefits. The independent life-cycle of each component leverages parallel development. Different teams may work on different components without getting in each others way too much.

From the customer perspective, however, the fact that a software system is developed in a component-based fashion is completely uninteresting. Customers are interested in a working, quality software product that keeps on working at all times. The fact that our hypothetical products are delivered as closures is invisible to the customer and that should stay that way. Decomposition is a solution space concern and therefore only relevant for software developers, not for customers.

Hiding component structure from customers creates an additional requirement for continuous delivery, namely that products are released as compositions of components, or *closures*. Product variants are then represented by a top-level component, i.e. components that are not required by any other component (no incoming edges in the graph).

Figure 1.5 displays a “pro” variant and a “light” variant of the product. Both variants are identified by the two top-level components respectively. The light variant will be released as the closure of the *Light* component, i.e. it will be released as the composition of $\{Light, A, B, C\}$. On the other hand, the “pro” variant contains extra functionality, in this case implemented by the extra component *D*. If the “pro” variant is delivered, its release package consists of $\{Pro, A, B, C, D\}$.

The fact that every component represents, in essence, a software package that can released, delivered and deployed, leads to the requirement that the configuration of product variants (components in this case) should be fully recursive and compositional. In other words, the *locus* of configuration moves from the product level (configuration of the whole) to the level of individual components (a similar level of configurability can be observed in the KOALA component model [108]). If our prototypical customer is selecting the desired features for a certain product (variant) she is actually configuring a set of components. Consequently, the relation between composition (i.e. dependencies) and configuration (selection of features) should be clear. The configuration of the whole product is *derived* from the configuration of the sets of components that are in

the closure that represents the product. The other way round, the set of components that will end up in the closure may be dependent on the features the customer has selected. This is discussed in more detail in Chapter 2.

Summary of Requirements

The primary requirements for continuous delivery in the context of component-based development can now be summarized as follows:

1. Release packages should consist of compositions of components and be published automatically after every validated change.

2. It should be possible to trace compositions delivered to customers, back to the component sources that were used to create them.

3. Customers should be able to consistently configure compositions of components without further human intervention. Nevertheless, interfaces for configuration are to be specified on the level of components.

4. Transferring new releases to customers should be efficient and scalable.

We will now formulate the research questions that follow from these requirements.

1.3.2 Research Questions

The general research question addressed in this thesis is as follows:

> **General Research Question** *Can we automate the release of application updates in order to realize continuous delivery? This question is asked in the context of heterogeneous component-based software product lines.*

We attempt to answer this question by proposing techniques that satisfy the requirements identified above in order to enable the automation of the larger part of the configuration, integration, release and delivery phases of the software deployment life-cycle. More specifically, the techniques described in this thesis aim to answer the following questions:

Configuration The first step in automating delivery is automating the configuration of the product. This leads to the following two subsidiary research questions:

Q.1 Can we formally ensure consistent configuration of components and their compositions?

Q.2 Can abstract configurations be mapped to compositions of sets of software components at the programming level?

These questions are addressed in Part II.

Integration Integration consists of combining sets of components and see if they form a consistent whole; it is a precondition for releasing a product. In relation to our general research question this leads to the following research questions:

Q.3 Can the practice of continuous integration be extended to a practice of continuous release and delivery?

Q.4 Can we increase the frequency of delivery in the context of such a continuous release practice?

Delivery After a software product has been released, it has to be delivered efficiently to the end users. This leads to our final research question:

Q.5 Can we realize the delivery of release packages in an efficient way?

The questions on integration and delivery are addressed in Part III; the techniques proposed there all have been prototyped as part of the Sisyphus continuous integration and release system. The techniques proposed in Part II have been prototyped outside that context.

1.3.3 Summary of Contributions

The techniques developed represent stepping stones towards the goal of continuous delivery. Any form of continuous delivery, and especially in the context of component-based product lines, requires automation since implementing continuous delivery manually is clearly no viable alternative. The contributions of this thesis can be summarized as follows:

- Formalization of composite configuration interfaces and how to check their consistency. Consistency checking is performed using Binary Decision Diagrams (BDDs) [16], a scalable technique used in model-checking. This is described in Chapter 2 which has been previously published as: "Variability and Component Composition", in: *Proceedings of the 8th International Conference on Software Reuse (ICSR-8)*, LNCS 3107, Springer, 2004 [99].

- A consistent mapping of configuration interfaces to arbitrary software components (i.e. files) structured as dependency graphs. This is described in Chapter 3. We extend the checking algorithms of Chapter 2 and generalize the notion of component composition. This chapter has been published as "Generic Feature-Based Composition", in: *Proceedings of the Workshop on Software Composition (SC'07)*, LNCS, Springer, 2007 [102].

- A technique for incremental and continuous release of component-based systems realized in the Sisyphus prototype. Sisyphus integrates and releases component-based products after every change. The tool maintains accurate BOMs, thus enabling the automatic derivation of incremental updates. Sisyphus is described in Chapter 4. This work has been published as "Continuous Release and Upgrade of Component-Based Software", in: *Proceedings of the 12th International Workshop on Software Configuration Management (SCM-12)*, Lisbon, 2005 [100].

- The application of backtracking to incremental continuous integration in order to maximize opportunity for delivery. This is achieved by reusing earlier integration results in case of a failed build. Thus, build failure is no impediment to release and delivery of a working version. Backtracking is implemented as part of Sisyphus and is described in Chapter 5.

- Build penalty as a metric for ranking system integration strategies in order to reduce integration cost in the context of C/C++ based systems with explicit interfaces. Build penalty was validated in an industrial case-study at Philips Medical Systems. The metric and the results of the case-study are covered in Chapter 6. An earlier version of this chapter is under submission for publication as "Techniques for Incremental System Integration" This chapter is joint work with René Krikhaar and Frank Schophuizen.

- Binary change set composition as a technique for efficiently and consistently updating composite customer installations over the Internet. This lightweight approach to application upgrade can be easily implemented into any application thus leading to self-updating installations. Binary change set composition is presented in-depth in Chapter 7. This chapter has been published as "Binary Change-Set Composition", in: *Proceedings of the 10th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'07)*, LNCS, Springer, 2007 [101].

Continuous release, backtracking integration and binary change set composition have been prototyped as part of the Sisyphus continuous integration and release tool [103]. Sisyphus is currently in use to automatically build and release the ASF+SDF Meta-Environment [92] in a continuous fashion. The implementation of Sisyphus is discussed and evaluated in Chapter 8. An extended abstract on the Sisyphus tool implementation has been published as "The Sisyphus Continuous Integration System", in: *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, Tool track, IEEE, 2007 [103].

Part II

Configuration

Chapter 2

Variability and Component Composition

Abstract Automating configuration requires that users cannot make inconsistent feature selections which is a requirement for automated delivery. However, in component-based product populations, how product variants are configured, is to be described at the component level in order to be able to benefit from a product family approach: every component may be released as an individual product. As a consequence, configuration interferes with how components can be composed. We describe how the resulting complexity can be managed automatically. The concepts and techniques presented are first steps toward automated management of variability for continuous delivery.

This paper has been previously published as: T. van der Storm, Variability and Component Composition, in *Proceedings of the 8th International Conference on Software Reuse (ICSR-8)*, LNCS 3107, Springer, 2004 [99].

2.1 Introduction

Variability [106] is often considered at the level of one software product. In a product family approach different variants of one product are derived from a set of core assets. However, in component-based product *populations* [108] there is no single product: each individual component may represent a certain software product (obtained through component composition).

To let this kind of software products benefit from the product family approach, we present formal component descriptions to express component variability. To manage the ensuing complexity of configuration and component composition, we present techniques to verify the consistency of these descriptions, so that the conditions for correct component composition are guaranteed.

This chapter is structured as follows. In Sect. 2.2 we first discuss component-based product populations and why variability at the component-level is needed. Secondly, we propose a Software Knowledge Base (SKB) concept to provide some context to our work. We describe the requirements for a SKB and which kind of facts it is supposed to store. Section 2.3 is devoted to exploring the interaction of component-level variability with context dependencies. Section 2.4 presents the domain specific language CDL for the description of components with support for component-level variability. CDL will serve as a vehicle for the technical exposition of Sect. 2.5. The techniques in that section implement the consistency requirements that were identified in Sect. 2.2. Finally, we provide some concluding remarks and our future work.

2.2 Towards Automated Management of Variability

2.2.1 Why Component Variability?

Software components are units of independent production, acquisition, and deployment [87]. In a product family approach, different variants of one system are derived by combining components in different ways. In a component-based product population the notion of *one* system is absent. Many, if not all, components are released as individual products. To be able to gain from the product family approach in terms of reuse, variability must be interpreted as a component-level concept. This is motivated by two reasons:

- In component-based product populations no distinction is made between component and product.
- Components as unit of variation are not enough to realize all kinds of conceivable variability.

An example may further clarify why component variability is useful in product populations. Consider a component for representing syntax trees, called `Tree`. `Tree` has a number of features that can optionally be enabled. For instance, the component can be optimized according to specific requirements. If small memory footprint is a requirement, `Tree` can be configured to employ hash-consing to share equal subtrees. Following good design practices, this feature is factored out in a separate component, `Sharing`, which can be reused for objects other than syntax trees. Similarly, there is a component `Traversal` which implements generic algorithms for traversing tree-like data structures. Another feature might be the logging of debug information.

The first point to note, is that the components `Traversal` and `Sharing` are products in their own right since they can be used outside the scope of `Tree`. Nevertheless they are required for the operation of `Tree` depending on which variant of `Tree` is selected. Also, both `Traversal` and `Sharing` may have variable features in the very same way.

The second reason for component variability is that not all features of `Tree` can be factored out in component units. For example, the optional logging feature is strictly local to `Tree` and cannot be bound by composition.

The example shows that the variability of a component may have a close relation to component dependencies, and that each component may represent a whole family of (sub)systems.

2.2.2 The Software Knowledge Base

The techniques presented in this chapter are embedded in the context of an effort to automate component-based software delivery for product families, using a Software Knowledge Base (SKB). This SKB should enable the web-based configuration, delivery and upgrading of software. Since each customer may have her own specific set of requirements, the notion of variability plays a crucial role here.

The SKB is supposed to contain all relevant facts about all software components available in the population and the dependencies between them. Since we want to keep the possibility that components be configured before delivery, the SKB is required to represent their variability. To raise the level of automation we want to explore the possibility of generating configuration related artifacts from the SKB:

Configurators Since customers have to configure the product they acquire, some kind of user interface is needed as a means of communication between customer and SKB. The output of a configurator is a selection of features.

Suites To effectively deliver product instantiations to customers, the SKB is used to bundle a configured component together with all its dependencies in a configuration suite that is suitable for deployment. The configuration suite represents an abstraction of component composition.

Crucial to the realization of these goals is the consistency of the delivered configurations. Since components are composed into configuration suites before delivery, it is necessary to characterize the relation between component variability and dependencies.

2.3 Degrees of Component Variability

A component may depend on other components. Such a client component requires the presence of another component or some variant thereof. A precondition for correct composition of components is that a dependent component supports the features that are required by the client component. Figure 2.1 depicts three possibilities of relating component variability and composition.

The first case is when there is no variability at all. A component C_a requires components $C_1,...,C_n$. The component dependencies $C_1,...,C_n$ should just be present somehow for the correct operation of C_a. The resulting system is the composition of C_a and $C_1,...,C_n$, and all component dependencies that are transitively reachable from $C_1,...,C_n$.

Figure 2.1 (b) and (c) show the case that all components have *configuration interfaces* in the form of feature diagrams [57] (the triangles). These feature diagrams express the components' variability. The stacked boxes indicate that a component can

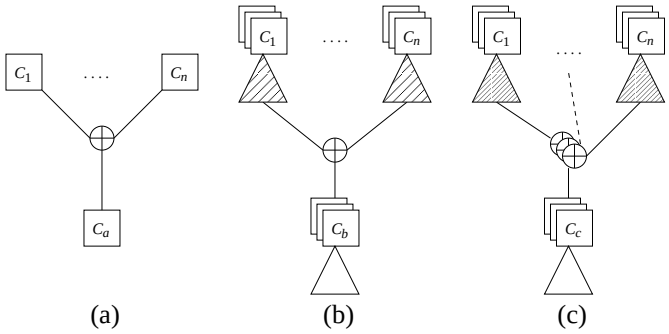


Figure 2.1: Degrees of component variability

be instantiated to different variants. The shaded triangles indicate that C_b and C_c depend on specific *variants* of $C_1, ..., C_n$. Features that remain to be selected by customers thus are local to the chosen top component (C_b resp. C_c).

The component dependencies of C_b are still fixed. For component C_c however, the component dependencies have become variable themselves: they depend on the selection of features described in the configuration interface of C_c. This allows components to be the units of variation. A consequence might be, for example, that when a client enables feature *a*, C_c requires component *A*. However, if feature *b* would have been enabled, C_c would depend on *B*. The set of constituent components of the resulting system may differ, according to the selected variant of C_c.

When composing C_a into a configuration suite, components $C_1, ..., C_n$ just have to be included. Components with variability, however, should be assembled into a suite guided by a valid selection of features declared by the top component (the component initially selected by the customer). Clients, both customers and requiring components, must select sets of features that are consistent with the feature diagram of the requested component.

How to establish these conditions automatically is deferred until after Sect. 2.4, where we introduce a domain specific language for describing components with variability.

2.4 Component Description Language

To formally evaluate component composition in the presence of variability, a language is needed to express the component variability described in Sect. 2.3. For this, a Component Description Language (CDL) is presented. This language was designed primarily for the sake of exposition; the techniques presented here could just as well be used in the context of existing languages. The language will serve as a vehicle for the evaluation of the situation in Fig. 2.1 (c), that is: component dependencies may depend themselves on feature selections.

component description ⟨"aterm-java", "1.3.2"⟩
features
ATerm : **all**(Nature, Sharing, Export, visitors?)
Nature : **one-of**(native, pure)
Sharing : **one-of**(nosharing, sharing)
Export : **more-of**(sharedtext, text)
sharedtext **requires** sharing
requires
when sharing {
⟨"shared-objects", "1.3"⟩ **with** fasthash
}
when visitors {
⟨"JJTraveler", "0.4.2"⟩
}

Figure 2.2: Description of `aterm-java`

For the sake of illustration, we use the ATERM library as an example component. The ATERM library is a generic library for a tree like data structure, called Annotated Term (ATerm). It is used to represent (abstract) syntax trees in the ASF+SDF Meta-Environment [60], and it in many ways resembles the aforementioned `Tree` component. The library exists in both Java and C implementations. We have elicited some variable features from the Java implementation. The component description for the Java version is listed in Fig. 2.2.

A component description is identified by a name (`aterm-java`) and a version (`1.3.2`). Next to the identification part, CDL descriptions consist of two sections: the features section and the requires section.

The features section has a syntax similar to Feature Description Language (FDL) as introduced in [105]. FDL is used since it is easier to automatically manipulate than visual diagrams due to its textual nature. The features section contains definitions of composite features starting with uppercase letters. Composite features obtain their meaning from feature expressions that indicate how sub-features are composed into composite features. Atomic features can not be decomposed and start with a lowercase letter.

The ATERM component exists in two implementations: a native one (implemented using the Java Native Interface, JNI), and a pure one (implemented in plain Java). The composite feature `Nature` makes this choice explicit to clients of this component. The feature obtains its meaning from the expression `one-of(native, pure)`. It indicates that either `native` or `pure` may be selected for the variable feature `Nature`, but not both. Both `native` and `pure` are atomic features. Other variable features of the ATERM-library are the use of maximal sub-term sharing (`Sharing`) and an inclusive choice of some export formats (`Export`). Additional constraints can be used to reduce the feature space. For example, the `sharedtext` feature enables the serialization of ATERMs, so that ATERMs can be written on file while retaining maximal sharing. Obviously, this feature requires the `sharing` feature. Therefore, the features section contains the constraint that `sharedtext` cannot be enabled without enabling

`sharing`.

The requires section contains component dependencies. A novel aspect of CDL is that these dependencies may be guarded by atomic features to state that they fire when a particular feature is enabled. These dependencies are *conditional* dependencies. They enable the specification of variable features for which components themselves are the unit of variation.

As an example, consider the conditional dependency on the `shared-objects` component which implements maximal sub-term sharing for tree-like objects. If the `sharing` feature is enabled, the ATERM component requires the `shared-objects` component. As a result, it will be included in the configuration suite. Note that elements of the requires section refer to *variants* of the required components. This means that component dependencies are configured in the same way as customers would configure a component. Configuration occurs by way of passing a list of atomic features to the required component. In the example this happens for the `shared-objects` dependency, where the variant containing optimized hash functions is chosen.

2.5 Guaranteeing Consistency

Since a configuration interface is formulated in FDL, we need a way to represent FDL feature descriptions in the SKB. Our prototype SKB is based on the calculus of binary relations, following [73]. The next paragraphs are therefore devoted to describing how feature diagrams can be translated to relations, and how querying can be applied to check configurations and obtain the required set of dependencies.

2.5.1 Transformation to Relations

The first step proceeds through three intermediate steps. First of all, the feature definitions in the features section are inlined. This is achieved by replacing every reference to a composite feature with its definition, starting at the top of the diagram. For our example configuration interface, the result is the following feature expression:

```
all(one-of(native, pure), one-of(nosharing, sharing),
    more-of(sharedtext, text),visitors?)
```

The second transformation maps this feature expression and additional constraints to a logical proposition, by applying the following correspondences:

$$\begin{aligned} \texttt{all}(f_1,...,f_n) &\mapsto \textstyle\bigwedge_{i\in\{1,...,n\}} f_i \\ \texttt{more-of}(f_1,...,f_n) &\mapsto \textstyle\bigvee_{i\in\{1,...,n\}} f_i \\ \texttt{one-of}(f_1,...,f_n) &\mapsto \textstyle\bigvee_{i\in\{1,...,n\}}(f_i \wedge \neg(\bigvee_{j\in\{1,..,i-1,i+1,...,n\}} f_j)) \end{aligned}$$

Optional features reduce to $\top$ since they are always independent of other features. Atomic features are mapped to logical variables with the same name. Finally, a `requires` constraint is translated to an implication. By applying these mappings to the inlined feature expression, one obtains the following formula.

$$\begin{aligned} &((native \wedge \neg pure) \vee (pure \wedge \neg native)) \wedge ((nosharing \wedge \neg sharing) \vee \\ &\qquad (sharing \wedge \neg nosharing)) \wedge (sharedtext \vee text) \wedge (sharedtext \rightarrow sharing) \end{aligned}$$

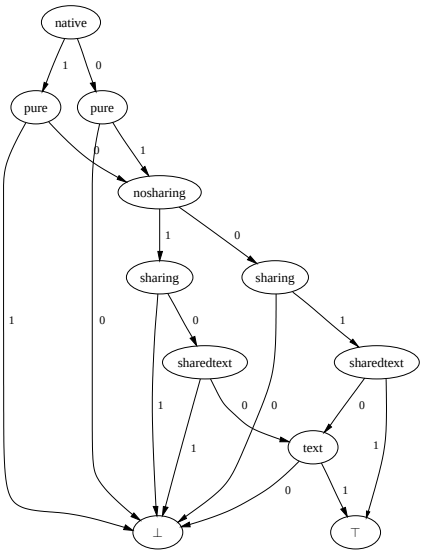


Figure 2.3: BDD for `aterm-java`

Checking the consistency of the feature diagram now amounts to obtaining *satisfiability* for this logical sentence. To achieve this, the formula is transformed to a Binary Decision Diagram (BDD) [16]. BDDs are logical expressions $\mathrm{ITE}(\varphi, \psi, \xi)$ representing if-then-else constructs. Using standard techniques from modelchecking any logical expression can be transformed into an expression consisting only of if-then-else constructs. If common subexpressions of this if-then-else expression are shared we obtain a directed acyclic graph which can easily be embedded in the relational paradigm. The BDD for the `aterm-java` component is depicted in Fig. 2.3.

2.5.2 Querying the SKB

Now that we have described how feature diagrams are transformed to a form suitable for storing in the SKB, we turn our attention to the next step: the querying of the SKB for checking feature selections and obtaining valid configurations.

The SKB is supposed to represent all kinds of software knowledge in order to better assess the possibilities for software delivery. For instance, if the SKB would store successor relations between versions of components, one can derive the set of possible update paths from one composition of components with a certain version, to another component. Similarly, this chapter aims to supplant such knowledge with configuration knowledge. The SKB is thus queried to answer questions like “Is this set of feature compatible with this component’s configuration interface” or “Is this component configurable at all?”.

The BDD graph consists of nodes labeled by guards. Each node has two outgoing edges, corresponding to the boolean value a particular node obtains for a certain as-

signment. All paths from the root to $\top$ represent minimal assignments that satisfy the original formula.

A selection of atomic features corresponds to a partial truth-assignment. This assignment maps for each selected feature the corresponding guard to 1 (true). Let φ be the BDD derived from the feature diagram for which we want to check the consistency of the selection, then the meaning of a selection is defined as: $\{a_1,...,a_n\} \mapsto \bigcup_{i\in\{1,...,n\}} [a_i/1]$ when $a_i \in \varphi$. In other words, a selection of features maps to a truth assignment mapping the guards corresponding to the feature to true. Checking whether this assignment can be part of a valuation amounts to finding a path in the BDD from the root to $\top$ containing the edges corresponding to the assignment. If there is no such path, the enabled features are incorrect. If there is such a path, but some other features must be enabled too, the result is the set of possible alternatives to extend the assignment to a valuation. The queries posted against the SKB use a special built-in query that generates all paths in a BDD. The resulting set of paths is then filtered according to the selection of features that has to be checked. The answer will be one of:

- $\{\{f_1,...,f_n\},\{g_1,...,g_m\},...\}$: a set of possible extensions of the selection, indicating an incomplete selection
- $\{\{\}\}$: one empty extension, indicating a correct selection
- $\{\}$: no possible extension, indicating incorrect selection

If the set of features was correct, the SKB is queried to obtain the set of configured dependencies that follow from the feature selection.

Take for example the selection of features $\{$`pure`,`sharedtext`,`visitors`$\}$. The associated assignment is $[pure/1][sharedtext/1]$. There is one path to $\top$ in the BDD that contains this assignment, so there is a valuation for this selection of features. Furthermore, it implies that the selection is not complete: part of the path is the truth assignment of `sharing`, so it has to be added to the set of selected features. Finally, as a consequence of the feature selection, both the JJTraveler and SharedObjects component must be included in the configuration suite.

2.6 Discussion

2.6.1 Related Work

CDL is a domain specific language for expressing component level variability and dependencies. The language combines features previously seen in isolation in other areas of research. These include: package based software development, module interconnection languages (MILs), and product configuration.

First of all, the work reported here can be seen as a continuation of package based software development [27]. In package based software development software is componentized in packages which have explicit dependencies and configuration interfaces. These configuration interfaces declare lists of options that can be passed to the build processes of the component. Composition of components is achieved through source tree composition. There is no support for packages themselves being units of variation.

A component description in CDL can be interpreted as a package definition in which the configuration interface is replaced by a feature description. The link between feature models and source packages is further explored in [104]. However, variability is described external to component descriptions, on the level of the composition.

Secondly, CDL is a kind of module interconnection language (MIL). Although the management of variability has never been the center of attention in the context of MILs, CDL complies with two of the main concepts of MILs [81]:

- The ability to perform static type-checking at an intermodule level of description.
- The ability to control different versions and families of a system.

Static type-checking of CDL component compositions is achieved by model checking of FDL. Using dependencies and feature descriptions, CDL naturally allows control over different versions and families of a system. Variability in traditional MILs boils down to letting more than one module implement the same module interface. So modules are the primary unit of variation. In addition, CDL descriptions express variability without committing beforehand to a unit of variation.

We know of one other instance of applying BDDs to configuration problems. In [86] algorithms are presented to achieve interactive configuration. The configuration language consists of boolean sentences which have to be satisfied for configuration. The focus of the article is that customers can interactively configure products and get immediate feedback about their (valid or invalid) choices. Techniques from partial evaluation and binary decision diagrams are combined to obtain efficient configuration algorithms.

2.6.2 Contribution

Our contribution is threefold. First, we have discussed variability at the component level to enable the product family approach in component-based product populations. We have characterized how component variability can be related to composition, and presented a formal language for the evaluation of this.

Secondly, we have demonstrated how feature descriptions can be transformed to BDDs, thereby proving the feasibility of a suggestion mentioned in the future work of [105]. Using BDDs there is no need to generate the exponentially large configuration space to check the consistency of feature descriptions and to verify user requirements.

Finally we have indicated how BDDs can be stored in a relational SKB which was our starting point for automated software delivery and generation of configurations.

The techniques presented in this chapter have been implemented in an experimental relational expression evaluator, called RSCRIPT. Experiments revealed that checking feature selections through relational queries is perhaps not the most efficient method. Nevertheless, the representation of feature descriptions is now seamlessly integrated with the representation of other software facts (such as, for example, the versions of the components).

2.6.3 Future Work

Future work will primarily be geared towards validating the approach outlined in this chapter. We will use the ASF+SDF Meta-Environment [60] as a case-study. The ASF+SDF Meta-Environment is a component-based environment to define syntax and semantics of (programming) languages. Although the Meta-Environment was originally targeted for the combination of ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism), directions are currently explored to parameterize the architecture in order to reuse the generic components (e.g., the user interface, parser generator, editor) for other specification formalisms [94]. Furthermore, the constituent components of the Meta-Environment are all released separately. Thus we could say that the development of the Meta-Environment is evolving from a component-based system towards a component-based product population. To manage the ensuing complexity of variability and dependency interaction we will use (a probably extended version of) CDL to describe each component and its variable dependencies.

In addition to the validation of CDL in practice, we will investigate whether we could extend CDL to make it more expressive. For example, in this chapter we have assumed that component dependencies should be fully configured by their clients. A component client refers to a variant of the required component. One can imagine that it might be valuable to let component clients inherit the variability of their dependencies. The communication between client component and dependent component thus becomes two-way: clients restrict the variability of their dependencies, which in turn add variability to their clients. Developers are free to determine which choices customers can make, and which are made for them.

The fact that client components refer to variants of their dependencies induces a difference in binding time between user configuration and configuration during composition [32]. The difference could be made a parameter of CDL by tagging atomic features with a time attribute. Such a time attribute indicates the moment in the development and/or deployment process the feature is allowed to become active. Since all moments are ordered in a sequence, partial evaluation can be used to partially configure the configuration interfaces. Every step effects the binding of some variation points to variants, but may leave other features unbound. In this way one could, for example, discriminate features that should be bound by conditional compilation from features that are bound at activation time (e.g., via command-line options).

Acknowledgments Paul Klint, Gerco Ballintijn and Jurgen Vinju provided helpful comments on earlier versions of this chapter.

Chapter 3

Generic Feature-Based Composition

Abstract In this chapter we propose a technique to consistently derive product instances from a set of software artifacts that can be configured in multiple ways. Feature descriptions are used to formally describe the configuration space in terms of the problem domain. By mapping features to one or more solution space artifacts and checking the consistency of this mapping, valid configurations are guaranteed to correspond to valid product instances. The approach is simple, generic and it alleviates the time-consuming process of instantiating product variants by hand. As such this a crucial step to achieve continuous delivery.

This chapter has been published previously as: T. van der Storm, Generic feature-Based composition, in *Proceedings of the Workshop on Software Composition (SC'07)*, LNCS, Springer, 2007 [102].

3.1 Introduction

In product line engineering, automatic configuration of product line instances still remains a challenge [7]. Product configuration consists of selecting the required features and subsequently instantiating a software product from a set of implementation artifacts. Because features capture elements of the problem domain, automatic product composition requires the explicit mapping of features to elements of the solution domain. From a feature model we can then generate tool support to drive the configuration process.

However, successful configuration requires consistent specifications. For instance, a feature specification can be inconsistent if selecting one feature would require another feature that excludes the feature itself. Because of the possibly exponential size of the configuration space, maintaining consistency manually is no option.

We investigate how to bridge the "white-board distance" between problem space and solution space [62] by combining both domains in a single formalism based on feature descriptions [105]. White-board distance pertains to the different levels of abstraction in describing problem domain on the one hand, and solution domain on the other hand. In this chapter, feature descriptions are used to formally describe the configuration space in terms of the problem domain. The solution domain is modeled by a dependency graph between artifacts.

By mapping features to one or more solution space artifacts, configurations resulting from the configuration task map to compositions in the solution domain. Thus it becomes possible to derive a configuration user interface from the feature model to automatically instantiate valid product line variants.

3.1.1 Problem-Solution Space Impedance Mismatch

The motivation for feature-based software composition is based on the following observations: solution space artifacts are unsuitable candidates for reasoning about the configurability in a product line. Configuration in terms of the problem domain, however, must stand in a meaningful relation to those very artifacts if it should be generally useful. Let's discuss each observation in turn.

First, if software artifacts can be composed or configured in different ways to produce different product variants it is often desirable to have a high-level view on which compositions are actually meaningful product instances. That is, the *configuration space* should be described at a high level of abstraction. If such configuration spaces are expressed in terms of problem space concepts, it is easier to choose which variant a particular consumer of the software actually needs. Finally, such a model should preferably be a formal model in order to prevent inconsistencies and configuration mistakes.

The second observation concerns the value of relating the configuration model to the solution space. The mental gap between problem space and solution space complicates keeping the configuration model consistent with the artifacts. Every time one or more artifacts change, the configuration model may become invalid. Synchronizing both realms without any form of tool support is a time-consuming and error-prone process. In addition, even if the configuration model is used to guide the configuration task, there is the possibility of inconsistencies in both the models and their interplay.

From these observations follows that in order to reduce the effort of configuring product lines and subsequently instantiating product variants, tool support is needed that helps detecting inconsistencies and automates the manual, error-prone task of collecting the artifacts for every configuration. This leads to the requirements for realizing automatic software composition based on features.

- The configuration interface should be specified in a language that allows *formal* consistency checking. If a configuration interface is consistent then this means there are valid configurations. Only valid configurations must be used to instantiate products. Such configurations can be mapped to elements of the solution domain.

- A model is needed that relates features to artifacts in the solution space, so that if a certain feature is selected, all relevant artifacts are collected in the final product. Such a mapping should respect the (semantic) relations that exist between the artifacts. For the mapping to be applicable in heterogeneous settings, no assumptions should be made about programming language or software development methodology.

3.1.2 Related Work

This work is directly inspired by the technique proposed in [29]. In that position paper feature diagrams are compared to grammars, and parsing is used to check the consistency of feature diagrams. Features are mapped to software packages. Based on the selection of features and the dependencies between packages, the product variant is derived. Our approach generalizes this technique on two accounts: first we allow arbitrary constraints between features, and not only structural ones that can be verified by parsing. Second, in our approach *combinations* of features are mapped to artifacts, allowing more control over which artifact is required when.

There is related work on feature oriented programming that provides features with a direct solution space semantics. For instance, in AHEAD [8] features form elements in an algebra that can be synthesized into software components. Although this leaves open the choice of programming language it assumes that it is class-based. Czarnecki describes a method of mapping features to model elements in a model driven architecture (MDA) setting [22]. By "superimposing" *all* variants on top of UML models, a product can be instantiated by selectively disabling variation points.

An even more fine grained approach is presented in [80] where features become first-class citizens of the programming language. Finally, a direct mapping of features to a component role model is described in [51].

These approaches all, one way or the other, merge the problem domain and the solution domain in a single software development paradigm. In our approach we keep both domains separate and instead relate them through an explicit modeling step. Thus our approach does not enforce any programming language, methodology or architecture beforehand, but instead focuses on the possibility of automatic configuration and flexibility.

Checking feature diagrams for consistency is an active area of research [17, 69, 105] but the level of formality varies. The problem is that checking the consistency is equivalent to propositional satisfiability, and therefore it is often practically infeasible. Our approach is based on BDDs [99], a proven technique from model checking, which often makes the exponential configuration space practically manageable.

3.1.3 Contributions

The contributions of this chapter can be summarized as follows:

- Using an example we analyze the challenges of bridging the gap between problem space and solution space. We identify the requirements for the explicit and controlled mapping of features to software artifacts.

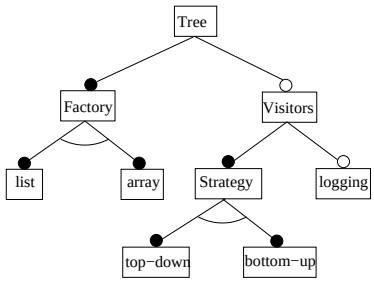


Figure 3.1: Problem space of a small example visualized as a feature diagram

- We propose a formal model that allows both worlds to be bridged in order to achieve (solution space) composition based on (problem space) configuration. Instances of the model are checked for consistency using scalable techniques widely used in model-checking.

- The model is unique in that it does not dictate programming language, is independent of software development methodology or architectural style, and does not require up-front design, provided that the solution domain supports an appropriate composition mechanism. Since no up-front design is needed, the approach can be adopted late in the development process or in the context of legacy software.

Organization of this chapter In the following section, Sect. 3.2, feature diagrams [57] are introduced as a model for the configuration space of product lines. Feature diagrams are commonly used to elicit commonality and variability of software systems during domain analysis [106]. They can be formally analyzed so they are a viable option for the first requirement.

Next, in Sect. 3.2.3 we present an abstract model of the solution space. Because we aim for a generic solution, this model is extremely simple: it is based on the generic notion of *dependency*. Thus, the solution space is modeled by a dependency graph between artifacts. Artifacts include any kind of file that shapes the final software product. This includes source files, build files, property files, locale files etc.

Then, in Sect. 3.3 we discuss how feature diagrams and dependency graphs should be related in order to allow automatic composition. The formalization of feature diagrams is described in Sect. 3.3.2, thus enabling the application of model-checking techniques for the detection of inconsistencies. How both models are combined is described in Sect. 3.4. This combined model is then used to derive product instances. Finally we present some conclusions and provide directions for future work.

3.2 Problem and Solution Space Models

3.2.1 Introduction

To be able to reason about the interaction between problem space an solution space, models are required that accurately represent the domains in a sufficiently formal way. In this section we introduce feature diagrams as a model for the problem space, and dependency graphs for the solution space.

3.2.2 Problem Space: Feature Diagrams

Figure 3.1 shows a graphical model of a small example's problem space using feature diagrams [57]. Feature diagrams have been used to elicit commonality and variability in domain engineering. A feature diagram can be seen as a specification of the configuration space of a product line.

In this example, the top feature, Tree, represents the application, in this case a small application for transforming tree-structured documents, such as parse trees. The Tree feature is further divided in two sub features: Factory and Visitors. The Visitors feature is optional (indicated by the open bullet), but if it is chosen, a choice must be made between the top-down or bottom-up alternatives of the Strategy feature and optionally there is the choice of enabling logging support when traversing trees. Finally, the left sub-feature of Tree, named Factory, captures a mandatory choice between two, mutually exclusive, implementations of trees: one based on lists and the other based on arrays.

Often these diagrams are extended with arbitrary constraints between features. For instance one could state that the array feature *requires* the logging feature. Such constraints make visually reasoning about the consistency of feature selection with respect to a feature diagram much harder. In order to automate such reasoning a semantics is needed. Many approaches exist, see e.g. [11, 14, 69]. In earlier work we interpreted the configuration problem as satisfiability problem and we will use that approach here too (cf. Chapter 2).The description consistency checking of feature diagrams is deferred to Sect. 3.3.2.

3.2.3 Solution Space: Implementation Artifacts

The implementation of the example application consists of a number of Java classes and AspectJ files [58]. Figure 3.2 shows a tentative design in UML. The implementation of the transformation product line is divided over two components: a tree component and visitors component. Within the tree component the Abstract Factory design pattern is employed to facilitate the choice among list- and array-based trees. In addition to the choice between different implementations, trees can optionally be enhanced with a `Visitable` interface by weaving an aspect. This enables that clients of the tree component are able to traverse the trees by using the visitors component. So weaving in the Visitability aspect causes a dependency on the visitors component.

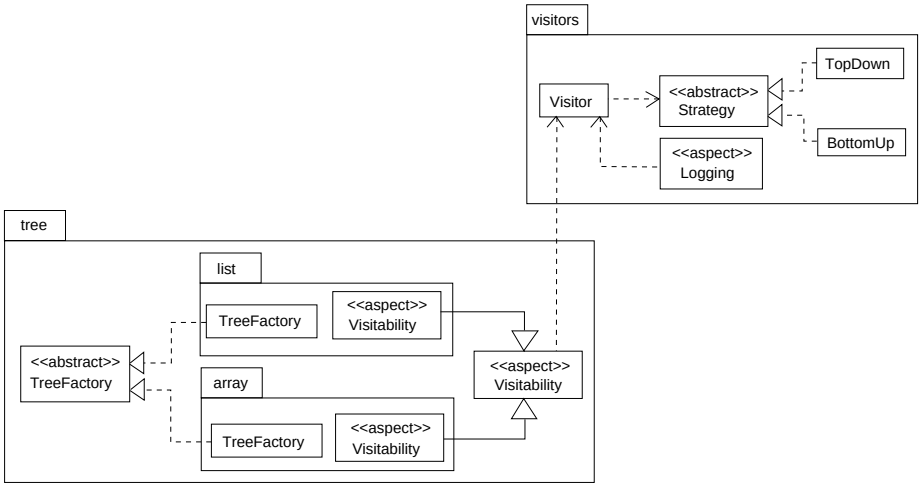


Figure 3.2: UML view of an example product line

3.2.4 Artifact Dependency Graphs

What is a suitable model of the solution space? In this chapter we take a an abstract stance and model the solution space by a directed acyclic dependency graph. In a dependency graph nodes represent artifacts and the edges represent dependencies between them. These dependencies may be specified explicitly or induced by the semantics of the source. As an example of the latter: a Java class file has a dependency on the class file of its superclass. Another example are aspects that depend on the classes they will be weaved into. For the example the dependencies are shown in Fig. 3.3. The figure shows dependencies of three kinds: subtype dependency (e.g. between list.Tree and Tree), aspect dependency (between Visitability and Tree), collaboration dependency (between Visitor and Strategy).

Dependency graphs are consistent, provided that the dependency relation conforms to the semantics of the artifacts involved and provided that every node in the graph has a corresponding artifact. A set of artifacts is consistent with respect to a dependency graph if it is closed under the dependency relation induced by that graph.

A nice property of these graphs is that, in theory, every node in it represents a valid product variant (albeit a useless one most of the time). If we, for instance, take the Visitability node as an example, then we could release this ‘product’ by composing every artifact reachable from the Visitability node. So, similar to the problem space of the previous section, the solution space is also a kind of configuration space. It concisely captures the possibilities of delivery.

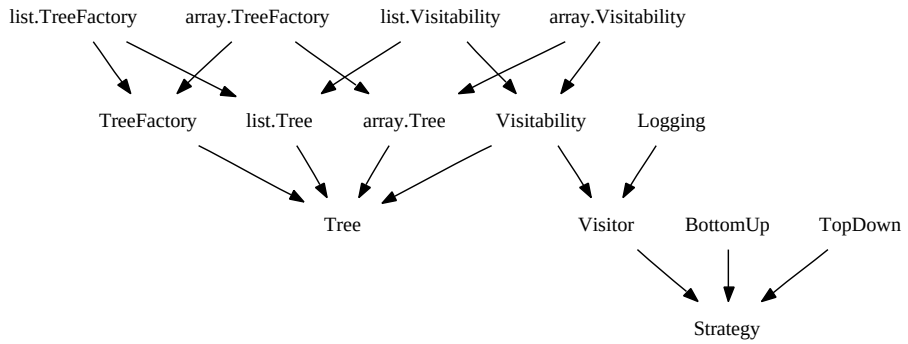


Figure 3.3: Solution space model of the example: dependency graph between artifacts

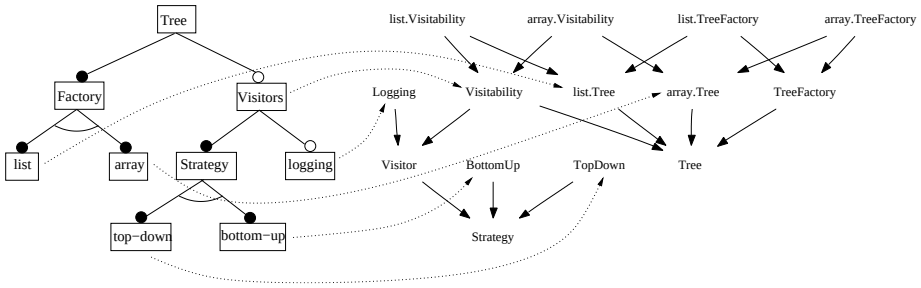


Figure 3.4: Partial mapping of features to artifacts

3.3 Mapping Features to Artifacts

3.3.1 Introduction

Now that the problem space is modeled by a feature diagram and the solution space by a dependency graph how can we bridge the gap between them? Intuitively one can map each feature of the feature diagram to one or more artifacts in the dependency graph. Such an approach is graphically depicted in Fig. 3.4.

The figure shows the feature diagram together with the dependency graph of the previous section. Arrows from features to the artifacts indicate which artifact should be included if a feature is selected. For instance, if the top-down strategy is chosen to visit trees, then the TopDown implementation will be delivered together with all its dependencies (i.e. the Strategy interface). Note that the feature mapping is incomplete: selecting the Visitors feature includes the Visitability aspect, but it is unspecified which concrete implementation (list.Visitability or array.Visitability) should be used. The graphical depiction thus is too weak to express the fact that if *both* array/list and Visitors is chosen, both the array.Visitability/list.Visitability and Visitability artifacts

Features	Logic
feature	boolean formula
atomic and composite features	atoms
configurability	satisfiability
configuration	valuation
validity of a configuration	satisfaction

Table 3.1: Feature descriptions as boolean formulas

Tree	:	**all**(Factory, Visitors**?**)
Factory	:	**one-of**(list, array)
Visitors	:	**all**(Strategy, logging**?**)
Strategy	:	**one-of**(top-down, bottom-up)

Figure 3.5: Textual FDL feature description of the example

are required. In Sect. 3.4 this problem will be addressed by expressing mapping as constraints between features and artifacts.

3.3.2 Feature Diagram Semantics

This section describes how feature diagrams can be checked for consistency. We take a logic based approach that exploits the correspondence between feature diagrams and propositional logic (see Table 3.1). Since graphical formalisms are less practical for building tool support, we use a textual version of feature diagrams, called Feature Description Language (FDL) [105]. The textual analog of feature diagram in Fig. 3.1 is displayed in Fig. 3.5. Composite features start with an upper-case letter whereas atomic features start in lower-case. Composing features is specified using connectives, such as, **all** (mandatory), **one-of** (alternative), **?** (optional), and **more-of** (non-exclusive choice). In addition to representing the feature diagram, FDL allows arbitrary constraints between features.

For instance, in the example one could declare the constraint “array **requires** logging”. This constraint has the straightforward meaning that selecting the array feature should involve selecting the logging feature. Because of these and other kinds of constraints a formal semantics of feature diagrams is needed, because constraints may introduce inconsistencies not visible in the diagram, and they may cause the invalidity of certain configurations, which is also not easily discerned in the diagram.

3.3.3 Configuration Consistency

The primary consistency requirement is internal consistency of the feature description. An inconsistent feature description cannot be configured, and thus it would not be possible to instantiate the corresponding product. An example of an inconsistent feature description would be the following:

A : **all**(b, c)
b **excludes** c

Feature b excludes feature c, but they are defined to be mandatory for A. This is a contradiction if A represents the product. Using the correspondence between feature descriptions and boolean formulas (cf. Table 3.1), we can check the consistency of a description by solving the satisfiability problem of the corresponding formula.

Configuration spaces of larger product lines quickly grow to exponential size and the problem of satisfiability is NP-complete. It is therefore essential that scalable techniques are employed for the verification and validation of feature descriptions and feature selections respectively. Elsewhere, we have described a method to check the logical consistency requirements of component-based feature diagrams [99]. That technique is based on translating component descriptions to logical formulas called binary decision diagrams (BDDs) [16]. BDDs are logical if-then-else expressions in which common subexpressions are shared; they are frequently used in model-checking applications because they often represent large search spaces in a feasible way. Any propositional formula can be translated to a BDD. A BDD that is different from falsum ($\bot$) means that the formula is satisfiable.

A slightly different mapping is used here to obtain the satisfiability result. The boolean formula derived from the example feature description is as follows:

$$(\mathit{Tree} \rightarrow \mathit{Factory}) \wedge$$
$$(\mathit{Factory} \rightarrow ((\mathit{list} \wedge \neg \mathit{array}) \vee (\neg \mathit{list} \wedge \mathit{array}))) \wedge$$
$$(\mathit{Visitors} \rightarrow \mathit{Strategy}) \wedge$$
$$(\mathit{Strategy} \rightarrow ((\mathit{top\text{-}down} \wedge \neg \mathit{bottom\text{-}up}) \vee (\neg \mathit{top\text{-}down} \wedge \mathit{bottom\text{-}up})))$$

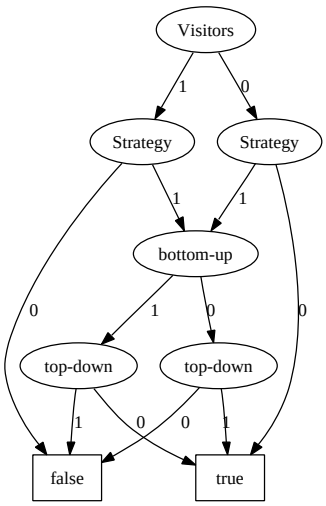
Note how all feature names become logical atoms in the translation. Feature definitions of the form $\mathit{Name} : \mathit{Expression}$ become implications, just like "requires" constraints. The translation of the connectives is straightforward. Such a boolean formula can be converted to a BDD using standard techniques (see for instance [42] for an elegant approach).

The resulting BDD can be displayed as a directed graph where each node represents an atom and has two outcoming edges corresponding to the two branches of the if-then-else expression. Figure 3.6 shows the BDD for the Visitors feature both as a graph and if-then-else expression. As one can see from the paths in the graph, selecting the Visitors feature means enabling the Strategy feature. This in turn induces a choice between the top-down and bottom-up features. Note that the optional logging feature is absent from the BDD because it is not constrained by any of the other variables.

3.4 Selection and Composition of Artifacts

3.4.1 Introduction

If a feature description is found to be consistent, it can be used to generate a configuration user interface. Using this user interface, an application engineer would select features declared in the feature description. Selections are then checked for validity




```
if Visitors then
  if Strategy then
    if bottom-up then
      if top-down then ⊥ else ⊤ fi
    else
      if top-down then ⊤ else ⊥fi
    fi
  else
    ⊥
  fi
else
  if Strategy then
    if bottom-up then
      if top-down then ⊥ else ⊤ fi
    else
      if top-down then ⊤ else ⊥fi
    fi
  else
    ⊤
  fi
fi
```

Figure 3.6: BDD for the Visitors feature

using the BDD. The selection of features, called the configuration, is then used to instantiate the product. Sets of selected features correspond to a sets of artifacts. Let's call these the (configuration) induced artifacts. The induced artifacts form the initial composition of the product. Then, every artifact that is reachable from any of the induced artifacts in the dependency graph, is added to the composition.

3.4.2 Configuration and Selection

In Sect. 3.3 we indicated that mapping single features to sets of artifacts was not strong enough to respect certain constraints among the artifacts. The example was that the concrete Visitability aspects (array.Visitability and list.Visitability) were not selected if the Visitors feature were only mapped to the abstract aspect Visitability. To account for this problem we extend the logical framework introduced in Sect. 3.3.2 with constraints between features and artifacts. Thus, mappings become requires constraints (implications) that allow us to include artifacts when certain *combinations* of features are selected. The complete mapping of the example would then be specified as displayed in Fig. 3.7.

The constraints in the figure – basically a conjunction of implications – are added to the feature description. Using the process described in the previous section, this hybrid 'feature' description is translated to a BDD. The set of required artifacts can then be found by partially evaluating the BDD with the selection of features. This results in a, possibly partial, truth-assignment for the atoms representing artifacts. Any artifact atom that gets assigned $\top$ will be included in the composition together with the artifacts reachable from it in the dependency graph. Every artifact that gets assigned $\bot$ will not be included. Finally, any artifact that did not get an implied assignment may or may not be included, but at least is not required by the selection of features.

list **and** Visitors **requires** *list.Visitability*
array **and** Visitors **requires** *array.Visitability*
list **requires** *list.TreeFactory*
array **requires** *array.TreeFactory*
top-down **requires** *TopDown*
bottom-up **requires** *BottomUp*
logging **requires** *Logging*

Figure 3.7: Mapping features to artifacts

Tree
array
list
array, top-down
list, top-down
array, bottom-up
list, bottom-up
array, top-down, logging
list, top-down, logging
array, bottom-up, logging
list, bottom-up, logging

Visitors
top-down
bottom-up
top-down, logging
bottom-up, loggin

Table 3.2: Deliverable configurations per component

Table 3.2 shows all possible configurations for the example product line. The configuration are identified by the set of selected atomic features as used in the feature diagram. The set of artifacts included in the distribution follows directly from each configuration. The table shows that even this very small product line already exposes 14 product variants.

3.4.3 Composition Methods

In the previous subsection we described how the combination of problem space feature models can be linked to solution space dependency graphs. For every valid configuration of the feature description we can derive the artifacts that should be included in the final composition. However, how to enact the composition was left unspecified. Here we discuss several options for composing the artifacts according to the dependency graph.

In the case of the example composing the Java source files entails collecting them in a directory an compiling the source files using `javac` and AspectJ. However, this presumes that the artifacts are actually Java source files, which may be a too fine granularity. Next we describe three approaches to composition that support different levels of granularity:

- Source tree Composition [26]

- Generation of a build scripts [108]
- Container-based dependency injection [38]

Source Tree Composition Source tree composition is based on *source packages*. Source packages contain source code and have an abstract build interface. Each source package explicitly declares which other packages it requires during build, deployment and/or operation. The source trees contained in these packages can be composed to obtain a composite package. This package has a build interface that is used to build the composition by building all sub-packages in the right order with the right configuration parameters.

Applying this to our configuration approach this would mean that artifacts would correspond to source packages. Every valid selection of features would map to a set of root packages. From these root packages all transitively required packages can be found and subsequently be composed into a composite package, ready for distribution.

Build Script Generation An approach taken in the KOALA framework [108] is similar to source tree composition but works at the level of C-files. In KOALA a distinction is made between *requires* interfaces (specifying dependencies of a component) and *provides* interfaces (declaring the function that a component has to offer). The composition algorithm of KOALA takes these interfaces and the component definitions (describing part-of hierarchies) and subsequently generates a `Makefile` that specifies how a particular composition should be built.

Again, this could be naturally applied in our context of dependency graphs. The artifacts would be represented by the interfaces and the providing components. The dependency graph then follows from the requires interfaces.

Dependency Injection Another approach to creating the composition based on feature selections would consist of generating configuration files (or configuration code) for a dependency injection container implementation [38]. Dependency injection is a object-oriented design principle that states that every class should only reference interfaces in its code. Concrete implementations of these interfaces are then "injected" into a class via the constructor or via setter methods. How component classes are connected together ("wiring") is specified separately from the components.

In the case of Java components, we could easily derive the dependencies of those clasess by looking at the interface parameters of their constructors and setters. Moreover, we can statically derive which classes implement those interfaces (which also induces a dependency). Features would then be linked to these implementation classes. Based on the dependencies between the interfaces and classes one could then generate the wiring code.

3.5 Conclusions

3.5.1 Discussion: Maintaining the Mapping

Since problem space and solution space are structured differently, bridging the two may induce a high maintenance penalty if changes in either of the two invalidate the mapping. It is therefore important that the mapping of feature to artifacts is explicit, but not tangled.

The mapping of features to artifacts presented in this chapter allows the automatic derivation of product instances based on dependency graphs, but the mapping itself must be maintained by hand. Maintaining the dependency relation manually is no option since it continually co-evolves with the code base itself, but often these relations can be derived from artifacts automatically (e.g., by static analysis).

It is precisely the separation of feature models and dependency graphs makes maintaining the mapping manageable if the dependency graphs are available automatically. For certain nodes in the graph we can compute the transitive closure, yielding all artifacts transitively required from the initial set of nodes. This means that a feature has to be mapped only to the *essential* (root) artifact; all other artifacts follow from the dependency graph.

Additionally, changes in the dependencies between artifacts (as follows from the code base) have less severe consequences on such mappings. On other words, the coevolution between feature model and mapping on the one hand, and the code base on the other is much less severe. This reduces the cost of keeping problem space and solution space in sync.

3.5.2 Conclusion & Future Work

The relation between problem space and solution space in the presence of variability poses both conceptual and technical challenges. We have shown that both worlds can be brought together by importing solution space artifacts into the domain of feature descriptions. By modeling the relations among software artifacts explicitly and interpreting the mapping of combinations of features to artifacts as constraints on the hybrid configuration space, we obtain a coherent formalism that can be used for generating configuration user interfaces. On the technical level we have proposed the use BDDs to make automatic consistency checking of feature descriptions and mapping feasible in practice. Configurations are input to the composition process which takes into account the complex dependencies between software artifacts.

This work, however, is by no means finished. The formal model, as discussed in this chapter, is still immature and needs to be investigated in more detail. More analyses could be useful. For instance, one would like to know which configurations a certain artifact participates in order to better assess the impact of certain modifications to the code-base. Another direction we will explore is the implementation of a feature evolution environement that would help in maintaining feature models and their relation to the solution space.

A case-study must be performed to see how the approach would work in practice. This would involve building a tool set that allows the interactive editing, checking and

testing of feature descriptions, which are subsequently fed into a product configurator, similar to the CML2 tool used for the Linux kernel [82]. The Linux kernel itself would provide a suitable case to test our approach.

Part III

Integration & Delivery

Chapter 4

Continuous Release and Upgrade of Component-Based Software

Abstract In this chapter 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 in a continuous fashion. 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. From such releases traceable and incremental updates are derived.

We have validated our solution with the prototype tool Sisyphus that computes and delivers updates for a component-based software system developed at CWI.

This chapter has been published previously as: 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)*, Lisbon, 2005 [100].

4.1 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 [43], 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 chapter are:

- 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 [34].
- The design of a continuous release and update system based on this formalisation

The organisation of this chapter is as follows. In Section 4.2 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 4.3 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 4.4). Section 4.5 discusses the prototype tool that we have developed to validate the product model in practice. In Section 4.6 we discuss links to related work. Finally, we present a conclusion and list directions for future work in Section 4.7.

4.2 Problem Statement

4.2.1 Motivation

Component-based releasing presumes that a component can be released only if its dependencies are released [98]. Often, the version number of a released component and its dependencies are specified in some file (such as an RPM spec file [5]). 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 a 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

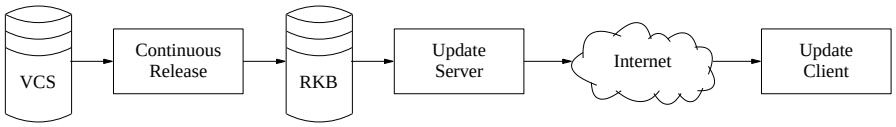


Figure 4.1: Continuous Release Architecture

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 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 chapter is to explore a technical solution to arrive at a 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.

4.2.2 Solution Overview

The basic architecture of our solution is depicted in Fig. 4.1. 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 [39]. In this chapter, 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

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

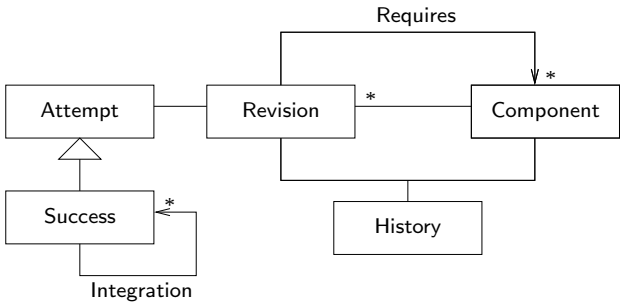


Figure 4.2: Continuous Integration Component Model

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 chapter.

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.

4.3 Continuous Release

4.3.1 Component Model

Our formalisation is based on the calculus of binary relations [88]. 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. 4.2.

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 give component revisions the following type:

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

This set contains tuples $\langle C, i \rangle$ where C represents 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:

$$\mathsf{History} \subseteq \mathsf{Time} \times \mathsf{Revision}$$

This relation is used to determine the state of a set of components at a certain moment in time; it is a total function from moments in time to revisions. 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:

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

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 $\langle t, B_j \rangle \in \mathsf{History}$. We further assume that no cyclic dependencies exist. This means that $\mathsf{Depends}[t]$ represents a directed acyclic graph for all $t \in \mathsf{Time}$.

4.3.2 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

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
```

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 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 2 Integrate components

```
1:  procedure INTEGRATEMANY(i, order, deps)
2:      for each r in order do
3:          D := {⟨i,d⟩ ∈ Attempt | d ∈ deps[r], ¬∃⟨j,d⟩ ∈ Attempt : j > i}
4:          if D ⊆ Success then
5:              if INTEGRATEONE(r, D) = success then
6:                  Success := Success ∪ {⟨i, r⟩}
7:                  Integration := Integration ∪ ({⟨i, r⟩} × D)
8:              end if
9:          end if
10:         Attempt := Attempt ∪ {⟨i, r⟩}
11:     end for
12: end procedure
```

Algorithm 1 displays the top-level continuous integration algorithm in pseudo-code. 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 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.

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.

4.3.3 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 4.3 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

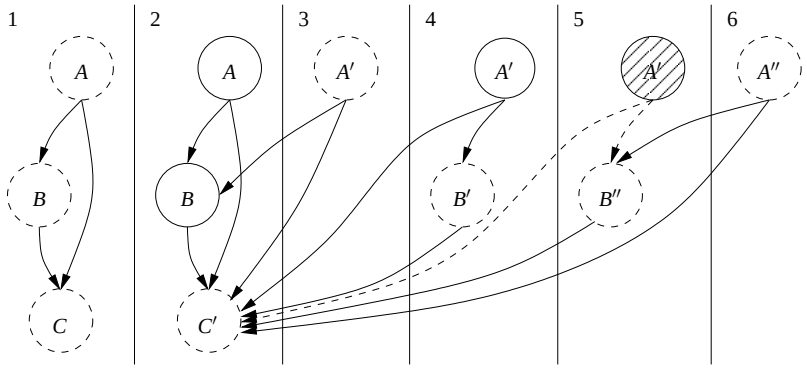


Figure 4.3: Six iterations of integration

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'.

4.4 Continuous Upgrade

4.4.1 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. 4.4. 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 chapter we used integration testing as a criterion. It can be seen from the algorithm INTEGRATEMANY that only successfully integrated components are released.

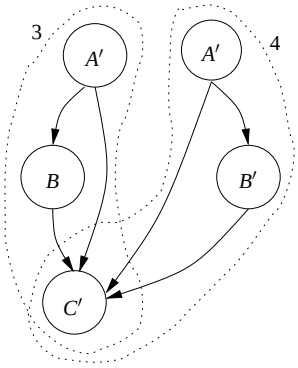


Figure 4.4: Two release packages for A'

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.

4.4.2 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 [53]. 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 (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. 4.3. 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.

4.5 Implementation

We have validated our formalisation of continuous release in the context the ASF+SDF Meta-Environment [95], 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 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. 4.3. 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. 4.5. The graph is similar to Fig. 4.3, 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`.

4.6 Related Work

4.6.1 Update Management

Our work clearly belongs to the area of update management. For an overview of existing tools and techniques we refer to [53]. 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 software deployment system Nix [31] 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 is 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 [85], Redhat's RPM [5] and the Gentoo/BSD ports [79, 109] 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 consequence of

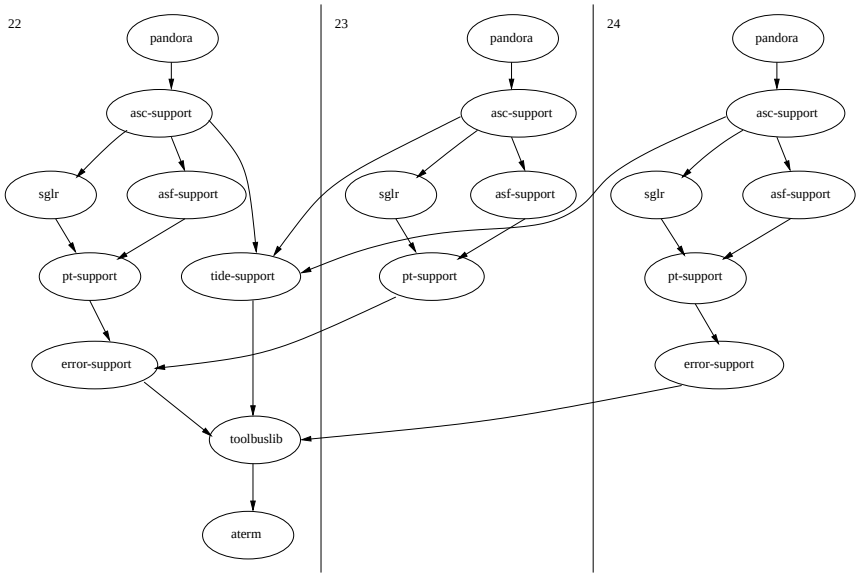


Figure 4.5: Three weekly releases of the `pandora` pretty printing component in 2004

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.

4.6.2 Relation Calculus

The relational calculus [88] has been used in the context of program understanding (e.g. [61, 66]), analysis of software architecture [36, 50], and configuration management [15, 65]. 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 [73] 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.

4.7 Conclusion and Future Work

Proper update management can be a serious advantage of software vendors over their comptetitors. In this chapter 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 chapter 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 [99]. 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.

Chapter 5

Backtracking Continuous Integration

Abstract Failing integration builds are show stoppers and hence an impediment to continuous delivery. Development activity is stalled because developers have to wait with integrating new changes until the problem is fixed and a successful build has been run. We show how backtracking can be used to mitigate the impact of build failures in the context of component-based software development. This way, even in the face of failure, development may continue and a working version is always available and release opportunities are increased.

5.1 Introduction

Continuous integration [39] has been heralded as a best practice of software development. After every change to the sources the complete system is built from scratch and the tests are run. If any of the tests fail, all effort is directed at fixing the problem in order to obtain a working version of the system. If the build fails, development is stalled. Continuous integration has therefore been called the "heartbeat of software". If it stops, you can't ship.

In this chapter, I describe a continuous integration scheme in component-based development settings. In this scheme I assume that integration is defined as building the source of a component against the (build artifacts of) its dependencies. Integrating the whole application then means building the topmost component in the dependency hierarchy.

The scheme employs two features to improve the feedback obtained from it. First, instead of building the complete system on every change, only the components that have affecting changes are rebuilt, and previous build results are reused otherwise [100]. Components are integrated in an incremental fashion, similar to the way the Unix tool MAKE can be used to selectively recompile files [37]. It turns out that due to the amount

of build sharing, the feedback is much quicker on average. As a result developers are can respond more quickly to problems encountered during integration.

The second feature, the primary focus of this chapter, is backtracking. If the build of a component has failed, it would make no sense to build any client components. Normally this would stall integration until the problem is fixed and the breaking component has been rebuilt. To prevent this from occurring, components that normally would depend on a broken component build, are built using *earlier* build results of the very same component. This way some measure of being completely up-to-date is traded for increased build feedback. In the end, *any* build is better than no build at all.

Contributions The contributions of this chapter can be summarized as follows:

1. I present a formalization of incremental continuous integration in the context of component-based development.
2. The formalization of incremental continuous integration is extended with two forms of backtracking, dubbed "simple backtracking" and "true backtracking"; both approaches are compared and I present and efficient algorithm for the latter.
3. Simple backtracking has been validated in practice; this has resulted in empirical data supporting its viability to improve continous integration.

Both build sharing and backtracking have been implemented as part of the continous integration and release system Sisyphus [103]. Sisyphus was used to validate the scheme in the setting of the ASF+SDF Meta-Environment [92], which is a language engineering workbench consisting of around 60 heterogeneous components. The results in this chapter derive from that case study.

5.2 Background

5.2.1 Component-Based Development

In component-based software configuration management (SCM) the sources of a software system are divided over individual components in the version control system (VCS). That is, the system is composed of different source trees that have independent evolution histories. The prime example of this approach to SCM is the Unified Change Management as implemented in IBM Rational's ClearCase [10].

Independent versioning of components promotes parallelism in development activity. Development on a component is more or less isolated from the rest of the system. Having a good architecture thus creates opportunities for reduced time to market. At the same time the traditional advantages of component-based software development apply: complexity is reduced, reuse and variation is stimulated.

Whereas most component models (e.g., COM [84]) separate the notions of interface and implementation, in this chapter I assume a more liberal notion: a component is considered to be just a logically coupled set of source files that has its own version history. Practically this means that a component is often represented as a directory entry in a VCS such as, for example, Subversion [19].

Components are often inter-related through dependency relations. In the presence of interfaces these are often specified as “provides” and “requires” interfaces. For instance, one component may implement a certain interface and is then said to provide it to client components. Client components may specify a dependency on such interfaces (i.e. they may require it). Hence, interfaces represent the “contractual window” between two components.

The intermediate step of interfaces introduces a level of indirection between two implementation components which is advantageous from the versioning perspective. This can be seen as follows: every time a component changes (i.e., a new version is produced), this has no impact on client code *unless* the provided interface changes in a backwards incompatible way. The evolution of two inter-dependent components is decoupled through the interface concept itself. Of course, this is only successful in practice if interfaces are sufficiently stable.

In some situations however, the notion of interface does not apply because it is too tightly bound to the implementation domain, i.e., platform and programming language. In heterogeneous systems the interface concepts simply does not apply. In this chapter, therefore, I simplify the provides/requires model of dependencies and discard the notion of interfaces. Components, i.e., implementation units (source trees), have dependencies on other components without the intermediary interface concept. In other words, components are allowed to require other components in order to be correctly built and/or deployed.

Such dependencies are specified without version identifier because that would introduce strong coupling between the client and the dependency component. As soon as the latter changes, the former is out of date. Keeping such dependency relations synchronized can be a true maintenance nightmare. We therefore let components reference their dependencies by name without version information. So, in a way, there still is some form of interface, albeit an empty one, which is the *name* of the required component.

However, it now becomes increasingly difficult to select configurations of components that make up consistent versions of a system. Any version of a component is a suitable candidate to satisfy the requirements of the client component that declares a dependency on its name. The configuration space has become exponentially large since we now have *complete* decoupling between the evolution histories of components.

This is where continuous integration comes in. Instead of explicitly searching the configuration space for the “right” configuration, we let an automated build system construct “bleeding edge” configurations as frequent and quick as possible. This means that always a working version is available without additional maintenance of selecting the right versions of the right components and doing the integration by hand.

5.2.2 Continuous Integration

Continuous integration proper originates from the Extreme Programming (XP) software development methodology [9]. There, the process of continuous integration also includes the continuous checking in of changes, however small they may be. Current usage of the term, however, most often refers to the process of building the complete system every time changes have been committed, whichever the frequency they oc-

cur in. As such it can be seen as a heavier instance of daily or nightly integration builds [71].

The goal of continuous integration is to know the *global* effects of *local* changes *as soon as possible*. Integration bugs are hard to track down because they originate from the interaction between (changes to) different subsystems, so they are hard to test for on a subsystem level. Post-poning integration makes things even worse: the interaction between changes increases very fast making integration bugs exponentially harder to find. It is therefore important that integration builds are executed quickly enough. As Martin Fowler states: "The whole point of Continuous Integration is to provide rapid feedback." [39] Failing builds, of course, are the main impediment to such rapid feedback if they are not fixed timely.

The word "integration" has yet another common meaning: that of putting parts together to obtain a coherent, meaningful whole. This overlaps partly with the integration of changes, but has additional force in the context of component-based software that is partioned in separate entities that have to be integrated at build time, deployment time or runtime. Indeed, any software product that has been broken up in parts to reduce complexity, increase time to market and promote reuse has be integrated at some time. It is better to do this quick and often, because the "very act of partioning the system introduces development process problems because interactive components are more complex than single entities" [75].

Component-based software *development* affects continous integration along both axes of the word integration. In component-based software development the *sources* of a product are partitioned in independently versioned components. This means that "change integration" (check in) can be highly parallelized since every component has its own development line. This increased parallellism poses even higher demands on continous integration. Furthermore, integration builds not only test the new changes, but also the *composition* of the different components (possibly containing those changes) to obtain a complete system.

5.2.3 Motivation: Continuous Release

The goal of backtracking continuous integration is to mitigate the effect of build failures in order to have a working version at all times and at the same time increase feedback for developers. Always having a working version of the system is a key requirement for continuous *release*, which entails making the software available after every change.

Releasing in this context means making the software available to a certain group of users. For instance, it might not be desirable to continuously release to actual end-users. However, it may be very benificial to release the software after every change to beta-testers or to the developers themselves (who want to see the effect of their changes).

Continuous release is motivated along the same line as the reason for continuous integration: early discovery of defects and a shortened feedback loop. Continuous release, however, requires automation of release and therefore the automation of integration.

In order to extend the process of continuous integration to continuous release, I distinguish the following goals:

- Feedback: *a* build (successful or not) is always better than no build. If there are changes to a component, the system should find a way to integrate them, even if builds of certain dependencies may have failed.
- Currency: the continuous integration system should always attempt to build the latest version of the software. Builds should be maximally up-to-date.
- Traceability: accurate *bills of materials* (BOMs) [70] should be maintained for the sake of tracing releases to the sources that were used to build them.
- Purity: the integration of components should be *pure*[1], i.e., the set of builds that transitively participate in an integration build should not involve multiple versions of the same component; this is a requirement for the derivation of release packages.
- Efficiency: the continuous integration system should perform no duplicate work. This means that previous build results should be reused if possible.

Not all of these goals can be achieved at once. For instance, we will see that there is a trade-off between maximal up-to-dateness and maximal feedback.

5.3 Overview

5.3.1 Introduction

Before I a describe incremental continuous integration and the two kinds of backtracking, I first introduce some preliminary assumptions. First of all, it is assumed that the dependencies of a component can be derived from the sources, for instance by analyzing a specific file that lists them explicitly. Since the specification of dependencies thus is part of the sources of a component, a change in the dependencies induces a change of the component. This allows for smooth evolution of a system's architecture.

In the context of our case-study the dependencies are specified in `pkgconfig` files [48]. For instance, the pretty-print subsystem of the ASF+SDF Meta-Environment corresponds to the `pandora` component [93]. Its `pkgconfig` file is shown below (slightly abridged for clarity):

```
Name: pandora
Requires: asc-support,aterm,pt-support,toolbuslib
```

The first line in the file declares the identity of this component, in this case `pandora`. The second line lists the required dependencies. Note that these dependencies do not have version identifiers attached. Requiring this would surely introduce a large maintenance penalty: on every change to one of the dependencies this files would have to be updated. Instead, at a certain moment in time the continuous integration system

[1] In Chapter 4 this was referred to as "homogeneous".

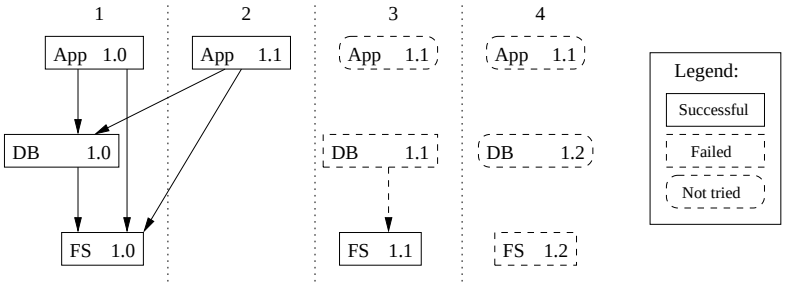


Figure 5.1: Incremental integration example

will take a snapshot of the repository and bind each unversioned dependency to the versions of those required components at that very moment in time. In other words, for all components the *latest* revision is taken.

Taking a snapshot of the repository results in a set of *source trees* which capture the state of every component at the moment of the snapshot. These source trees are related in a dependency graph that results from the requirements as specified within those source trees at the moment of the snapshot. The snapshot is now input to the continuous integration process.

5.3.2 Build Sharing

Now it is time to describe incremental continuous integration based on the snapshots introduced above. As an example, consider a small component-based application consisting of three components: App (the application) , DB (a database server) and FS (a file system library). The App component requires both DB and FS, whereas DB only requires FS; FS has no dependencies whatsoever. Of course, every build of a component has an implicit dependency on the build environment (e.g. compilers, build tools etc.). This dependency however, we assume, is managed by the continuous integration system itself.

Figure 5.1 shows four integration cycles, corresponding to each column. In the first iteration, all components have been successfully built (indicated by solid boxes and arrows). The arrows indicate the dependency relation in the snapshot at the time of integration.

In the second iteration, the App component has changed since the first iteration, but there are no changes in DB and FS. Instead of building all components from scratch—which would mean a waste of valuable resources—the incremental continuous integration system reuses the build results (e.g., binaries, libraries etc.) from earlier integrations for the dependencies of App. This is indicated by the arrows going from App 1.1 to DB 1.0 and FS 1.0. In other words, the builds of DB 1.0 and FS 1.0 are shared between the consecutive builds App, versions 1.0 and 1.1 respectively.

However, suppose that a build fails. This is shown in integration 3. Changes have been committed to both FS and DB, so all components require a build. In the case of

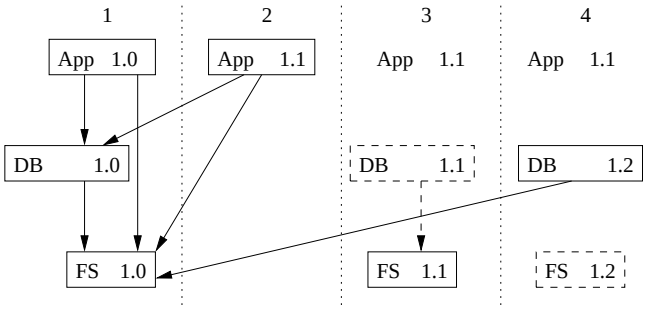


Figure 5.2: Incremental integration with simple backtracking

App a *re*build is required of version 1.1 in order to take the changes of DB and FS into account. So it is very well possible that a single component source tree will be built many times because of changes in dependencies.

Suppose that the build of DB 1.1 fails, however. This has been indicated by the dashed box around DB 1.1. This state of affairs prohibits a build of App 1.1 because one cannot build against failed dependencies. Builds that will not be attempted because of this reason are called “not tried”. In the figure this is indicated by dashed boxes with rounded corners. Clearly, “not tried” builds are to be avoided since we lose feedback. In the example no feedback is generated, for instance, on how the changes in FS affect App 1.1.

Finally, in the fourth cycle (column 4), again there are changes in DB (hopefully to fix the previous build) and in FS. However, now the build of FS 1.2 fails. As a consequence there is neither feedback on the changes in DB itself nor on the integration of changes in DB 1.2 with App 1.1. Again, feedback is less than optimal and, moreover, we still can only release App 1.1 with DB 1.0 and FS 1.0, and we can release FS 1.1 as a stand-alone component. I will now describe how a simple form of backtracking improves this situation slightly.

5.3.3 Simple Backtracking

In incremental continuous integration builds to satisfy component dependencies are always searched for within the current snapshot. For instance, in Figure 5.1, during the second integration the continuous integration system find builds for DB 1.0 and FS 1.0 to satisfy the dependencies of App 1.1, since both DB and FS have not changed since; both DB 1.0 and FS 1.0 are in the snapshot of cycle 2. In the next two cycles the current snapshot contains DB 1.1 and FS 1.1, and DB 1.2 and FS 1.2 respectively. However, in cycle 3 the build of DB 1.1 has failed, and in cycle 4, the build of FS 1.2 has failed. Hence it is not possible to build App 1.1 in either of the two cycles.

Figure 5.2 shows the application of simple bactracking. This means that, if there is a failed build in any of the dependencies of a component, say App, in the current cycle (with source trees in the current snapshot), the continuous integration goes back in time to find the first successful build of the component in question (in this case App), checks

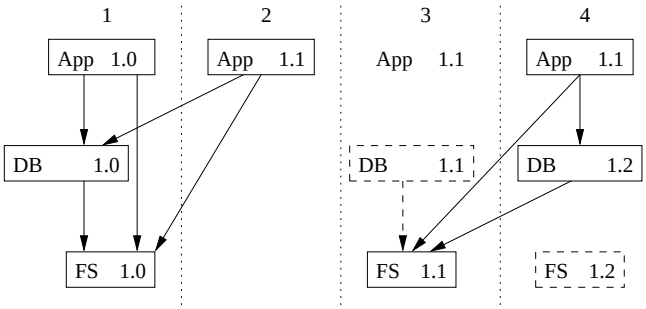


Figure 5.3: Incremental integration with true backtracking

if the requirements are still the same—does App still require both DB and FS?—and if so, uses the set of builds that were used back then.

To illustrate this, consider Figure 5.2. In cycle 3, there are failed dependencies for App 1.1. The most recent successful build of App with the same requirements, is the build of cycle 2. However, using that set of dependencies (DB 1.0 and FS 1.0) does not achieve anything: we would be merely *duplicating* the build of cycle 2 because App has not changed in between cycles 2 and 3. This is indicated by the absence of a box around App 1.1 in cycle 3. Note that this is a different outcome than "not tried", since with "not tried" builds we always lose something, either feedback or currency, and this is not the case here.

Another important detail here is that we cannot just use DB 1.0 and FS 1.1 for building App 1.1 in cycle 3, since that would lead to an impure build: DB 1.0 uses FS 1.0 whereas App 1.1 would have used FS 1.1. This means there are two versions (1.0 and 1.1) of the same component (FS) in the closure of App 1.1.

Simple backtracking shows its value in the fourth cycle: there is a change in DB, and there is a successful most recent build, the build of DB 1.0 in the first cycle. Using simple backtracking, at least DB 1.2 can be built. We do not get feedback on the integration of DB 1.2 and FS 1.1 but it is better than nothing at all. Although in this case, it seems trivial to just use the build of FS 1.1 for building DB 1.2, this is deceiving. When the dependency graph is more complex one cannot just take the most recent successful builds of dependencies without ensuring the result will be pure. This is exactly what true backtracking achieves, which I will discuss next.

5.3.4 True Backtracking

Simple backtracking involves searching for earlier successful builds of the component that should be built now. True backtracking adapts this search by search for sets of successfully built required components such that a purity of integration is ensured. Figure 5.3 shows the example scenario with true backtracking enabled.

The figure only differs from Figure 5.2 in the fourth cycle. Cycle 3 remains the same because using the most recent set of successful dependency builds that maintain

purity (DB 1.0 and FS 1.0) again would entail duplicating the App build of cycle 2. It is still impossible to use DB 1.0 and FS 1.1 because it would violate purity.

In the cycle 4 however, the new version DB (1.2) can now be built. The set of most recent successful dependency builds is {FS 1.1} and this set does not violate purity from the perspective of DB 1.2. Furthermore, App 1.1 can now also be built: {DB 1.2, FS 1.1} maintains purity.

Note that in both Figure 5.2 and Figure 5.3 all "not trieds" have disappeared. However, true backtracking presents the following advantages over simple backtracking:

- We were able to build DB 1.2 against FS 1.1 instead of FS 1.0, hence with true backtracking the build of DB 1.2 is more on the bleeding edge.
- It was possible to build App 1.1 against DB 1.2 and FS 1.1, hence we obtain one additional release opportunity for component App.

In addition, the simple backtracking suffers from the fact that it only works if dependencies have not changed in between integrations. In that case, the probability of finding an earlier build with the same set of requirements is rather low.

In the following section I will present a light-weight formalization of incremental continuous integration and the two forms of backtracking. The formalization extends earlier work reported in Van der Storm [100].

5.4 Formalization

5.4.1 Preliminaries

In order to reason about integration I introduce a lightweight formal model of components, revisions and builds in this section. It is instructive to see the relations and sets that make up the model as a *software knowledge base* (SKB) [73] that can be queried and updated. The SKB is required for implementing build sharing, backtracking and to automatically derive pure releases.

To be able to build a component, a source tree is needed for every component. To reflect this relation explicitly I introduce the relation *state* that bijectively maps components (names) to source trees (revisions) according to some criterion (for instance, by taking the *current* revision of each component). It has the following type:

$$State \subseteq Components \times Revisions$$

In practice, a revision $r \in Revisions$ is often represented as a tuple of a source location (e.g. a path or URL) together with a version identifier.

Source trees may declare dependencies on components. This is modeled by the relation *Requires*:

$$Requires \subseteq Revisions \times Components$$

Note that the domain and range of this relation are not the same. The idea is that dependencies on components may change inbetween revisions. For instance, source tree T may require the components A and B, but the following revision T' might only

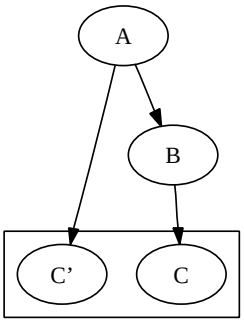


Figure 5.4: Impure integration of component *A*

require *A*. Moreover, the range of the relation is unversioned for another reason. If *Requires* would have been a relation from *Revisions* to *Revisions* this would mean that it would have to be updated on every change to a component that is depended by T. In the current situation, the requires relation can be maintained *within* the source trees themselves.

A snapshot maps every path to a tree according to some criterion, that is, it fixates a certain version for each component. The definition makes use of the *State* relation introduced above.

$$\mathit{Snapshot} = \mathit{Requires} \circ \mathit{State}$$

Snapshot is a dependency graph between *Revisions* and is the starting point for integration. This means that subsets of carrier(*Snapshot*) will be built in an appropriate order. The results are stored in a relation *Builds*:

$$\mathit{Builds} \subseteq \mathit{State} \times \mathbb{N}$$

This set is partitioned in two subsets *Success* and *Failure*, resp. containing the successful builds and the failed ones. A build relates a tree to a build number, because a single tree can be built many times, possibly using different dependencies. Which builds of dependencies actually were used is recorded in the *Integration* relation:

$$\mathit{Integration} \subseteq \mathit{Builds} \times \mathit{Builds}$$

Again, *Integration* is a dependency graph but this time between builds.

For any successful integration we require that the set of builds that participated in the integration is *pure*. Purity of a set of builds *B* is defined as:

$$\text{pure?}(B) \equiv |B| = |\text{domain}(\text{domain}(B))|$$

Note that a set *B* contains tuples of the form $\langle\langle c, r\rangle, i\rangle$, so that $\text{domain}(B) \subseteq \mathit{State}$, and hence $\text{domain}(\text{domain}(B)) \subseteq \mathit{Components}$. In other words, a set of builds *B* is pure if there are no two builds and/or revisions for the same component contained in *B*.

Algorithm 3 Template for component-based integration

```
 1: procedure INTEGRATE(i, Snapshot)
 2:     order ← topological-sort(Snapshot)
 3:     for t ∈ order do
 4:         w ← workingset(t)
 5:         if w is undefined then                          ▷ "Not tried"
 6:             Failed ← Failed ∪ {⟨t, i⟩}
 7:             continue
 8:         end if
 9:         if build?(t, w) then
10:             b ← execute-build(i, t, w)
11:             Builds ← Builds ∪ {b}
12:             Integration ← Integration ∪ ({b} × w)
13:         end if
14:     end for
15: end procedure
```

As an example of impurity of integration, consider Figure 5.4. The figure shows the integration of a component *A* against the integrations of its dependencies. During the integration of *B*, however, a different version of component *C* was used: two versions of *C* are reachable from *A*, hence, this constitutes an impure integration. The algorithms presented here ensure that *Integration* is pure for every build. Formally this means:

$$\forall b \in \mathit{Builds} : \text{pure?}(\mathit{Integration}^*[b])$$

This invariant was introduced in [100]. It ensures that the *Integration* relation can be used to derive *compositions* for every build which is what is delivered to users. If *Integration* would not have been pure composition would be ambiguous: it could occur that two builds used different versions for the same dependency,—which one should be in the composition? This invariant is used in Subsection 5.4.4 where backtracking is added to integration.

5.4.2 Schematic Integration Algorithm

Now that I have introduced the preliminary definitions, I present a schematic version of an algorithm for continuous integration in component-based development setting; it is shown in pseudo-code in Algorithm 3. The input to the build algorithm is the *Snapshot*, i.e. a relation between *Revisions* and a number that identifies the build cycle. Since snapshots are directed, acyclic graphs (DAGs) they have a topological order. The topological order consist of a list of vertices in the DAG, such that every dependency of vertex *N* comes *before N*. The topological order of the snapshot is stored in variable *order* on line 2.

Then, for each revision/source tree *t* in *order* (line 3) we obtain a *workingset* for *t* (line 4). Workingsets consist of builds ($\in$ *Success*) that will be used to satisfy the requirements of *t* (i.e. *Requires*$[t]$). The definition of workingset is a parameter of

this algorithm, as it captures the nature of backtracking. For now, we just assume that it returns a valid (i.e. pure) set of builds compatible to the requirements of t, if any. Below I will present three versions of the function, corresponding to the cases of no backtracking, simple backtracking and finally true backtracking.

If the function workingset is undefined (i.e. there are no valid workingsets) the algorithm continues with the next source tree in *order*. In this case the build of t is "not tried". Otherwise, w will be used in build i of t.

As Figure 5.2 and 5.3 showed, rebuilding a component using earlier dependencies occasionally amounts to duplicating earlier builds of that same component. The *build criterion* prevents this from occurring:

$$\text{build?}(t,w) \equiv \neg\exists b \in \mathit{Builds} : \text{tree}(b) = t \wedge \mathit{Integration}^{+}[b] = \mathit{Integration}^{*}[w]$$

This function takes a tree t and a workingset w and searches *Builds* for an earlier build of t. If such a build is found, the *Integration* relation is used to check whether the same (transitive) dependencies were used the algorithm is about to use now via w. If the two closures are the same, building t against w would mean duplicating an earlier build, and no build is required.

If, on the other hand, a build *is* required according to the build criterion, t is built against workingset w by the function execute-build on line 10. This function returns a new build entity b which is either failed or successful. The following lines update the software knowledge-base. First, b is added to *Builds*[2], then *Integration* is extended with tuples linking b to each build in w since this captures how t has been built.

5.4.3 Incremental Continuous Integration

In this subsection I explain how the model just introduced, can be used to do continuous integration in an incremental fashion. This is done by presenting an implementation of the workingset function referenced in Algorithm 3. Without backtracking, this function can specified as follows:

$$\begin{aligned}&\text{workingset}(t) = w\\ &\text{where}\\ &\quad w = \{\langle t', i\rangle \in \mathit{Builds} \mid t' \in T, \neg\exists\langle t', j\rangle \in \mathit{Builds} : j > i\}\\ &\quad w \subseteq \mathit{Success}\end{aligned}$$

For every revision (required by t) in the current snapshot, the working set contains the most recent build that has been successful. So, the set of workingsets is defined as the edge of the set of dependencies of t in the current snapshot if all builds are successful.

A valid workingset should contain successful builds for *every* component in $\mathit{Requires}[t]$. Because of topological order, every dependency of t has an element in *Builds*. This means, in turn, that the workingset contains the (globally) most recent build for those dependencies. However, if it contains a failed build, it makes no sense to proceed with building t. In that case t is added to the *Failure* part of *Builds* (see line 6 of Algorithm 3). This kind of failure—failure because dependencies have failed—is labeled

[2]This entails that b is either added to *Success* or *Failed*.

with "not tried". It is precisely these kinds of failures that backtracking is designed to mitigate.

If we turn our attention to Figure 5.1, we observe that the only valid working sets in each cycle (indicated by subscripts) are as follows:

$$\begin{array}{lcl}
\text{workingset}_1(\langle \text{FS}, 1.0\rangle) & = & \{\} \\
\text{workingset}_1(\langle \text{DB}, 1.0\rangle) & = & \{\langle\langle \text{FS}, 1.0\rangle, 1\rangle\} \\
\text{workingset}_1(\langle \text{App}, 1.0\rangle) & = & \{\langle\langle \text{DB}, 1.0\rangle, 1\rangle, \langle\langle \text{FS}, 1.0\rangle, 1\rangle\} \\
\text{workingset}_2(\langle \text{App}, 1.1\rangle) & = & \{\langle\langle \text{DB}, 1.0\rangle, 1\rangle, \langle\langle \text{FS}, 1.0\rangle, 1\rangle\} \\
\text{workingset}_3(\langle \text{FS}, 1.1\rangle) & = & \{\} \\
\text{workingset}_3(\langle \text{DB}, 1.1\rangle) & = & \{\langle\langle \text{FS}, 1.1\rangle, 2\rangle\}
\end{array}$$

The working sets are presented in the order of building, as follows from the the topological order between component revisions and the integration cycles.

5.4.4 Backtracking Incremental Continuous Integration

In the previous section dependencies were resolved by taken the latests builds out of *Builds* whether they had failed or not. In this section I change the dependency resolution algorithm in order to find the latest *successful* set of dependencies that lead to consistent (i.e. pure) integration. In the following I discuss two ways of backtracking: simple backtracking and true backtracking.

Formalization of Simple Backtracking

The simplest approach to find such a set is to look at earlier builds of the same component we are resolving the dependencies for. If an earlier successful build exists, then that build used successful dependencies. Since all built artifacts can be reproduced at all times, the dependencies of that earlier build could be used.

In this case the workingset is computed as follows:

$$\begin{array}{l}
\text{workingset}(t) = w \\
\text{where} \\
\quad t = \langle c, v\rangle, t' = \langle c, v'\rangle, \langle t', i\rangle \in \mathit{Success}, \\
\quad \neg\exists\langle\langle c, v''\rangle, j\rangle \in \mathit{Success} : j > i, \\
\quad w = \mathit{Integration}[t'], \mathit{Requires}[t'] = \mathit{Requires}[t]
\end{array}$$

In other words, the set of successful builds is searched for the most recent build t' of the component of t (i.e. c). For this build the working set is retrieved from the *Integration* relation. Because the requirements of t may have changed since t'—requirements declarations are part of the source tree—we explicitly require that t and t' have the same requirements.

By induction on the sequencing of builds (i.e. in time and topological ordering of build) we know that the workingset w is pure because build $\langle t', i\rangle$ is, and therefore w can be used to build t. As a consequence purity of *Integration* is maintained. If no workingset w is found, the build of t still fails with "not tried".

Simple backtracking has been implemented as part of the Sisyphus continuous integration system [103]. How actual continuous integration performance is affected by this strategy is discussed in Section 5.5.

Formalization of True Backtracking

Let's state the problem more precisely. Assume we are building a source tree t. The objective is to find the most recent set of successful builds D for resolving the declared dependencies of t. Normally the dependencies used will be the builds for $Snapshot[t]$, as they have been built already due to the topological order. But since these builds may have failed this requirement is weakened, that is, we are looking for *any* most recent set of successful builds for each component in $Requires[t]$ such that building t against D is pure.

If the builds for the dependent trees in the current snapshot *did* actually succeed, the following algorithm will select these builds as D nevertheless. Thus, if all builds succeed, no currency is lost with respect to the normal dependency resolution algorithm.

Next I will present a formal version of selecting the most recent set D that can be used to build a tree t. It operates by computing all combinations of successful builds for each of the components in $Requires[t]$ and then selecting the newest combination. Formally, this reads:

$$\text{workingsets}(t) = \prod_{c \in Requires[t]} \{\langle\langle c, v\rangle, i\rangle \in Success\}$$

The function workingsets returns all workingset candidates that *could* be used for building source tree t. However, this could contain invalid permutations that would cause the build of t to become impure.

If we consider Figure 5.3 again, it can be seen that this algorithm returns the following sets of workingsets in the fourth integration cycle for component revision App 1.1:

$$\begin{aligned}\text{workingsets}_4(\langle\text{App},1.1\rangle) = \{ & \\ & \{\langle\langle\text{DB},1.0\rangle,1\rangle,\langle\langle\text{FS},1.0\rangle,1\rangle\}, \\ & \{\langle\langle\text{DB},1.0\rangle,1\rangle,\langle\langle\text{FS},1.1\rangle,3\rangle\}, \\ & \{\langle\langle\text{DB},1.2\rangle,4\rangle,\langle\langle\text{FS},1.0\rangle,1\rangle\}, \\ & \{\langle\langle\text{DB},1.2\rangle,4\rangle,\langle\langle\text{FS},1.1\rangle,3\rangle\} \\ \}\end{aligned}$$

The second and third workingsets lead to impure integrations of App 1.1. This is a consequence of the fact that the FS version in those workingets (resp. 1.1 and 1.0) are not the versions that have been used in the builds of DB. Therefore, App 1.1 cannot be built using those workingsets. To fix the problem, the *Integration* relation is used to filter out the workingsets leading to impurity. This leads to the final version of

workingset which implements true backtracking:

$$\begin{aligned}&\text{workingset}(t) = w\\&\text{where}\\&\quad w \in \text{workingsets}(t),\\&\quad \text{pure?}(\mathit{Integration}^*[w]),\\&\quad w \text{ is most recent}\end{aligned}$$

Whether one workingset is more recent than another can be determined as follows. Since builds in *Builds* are totally ordered (in time), subsets $w \subseteq \mathit{Builds}$ can be sorted such that builds that are more recent come up front. Whether one working set is newer than another is determined by defining a lexicographic order on the sorted workingsets. In the example above it then follows that workingset $\{\langle\langle \text{DB}, 1.2\rangle, 4\rangle, \langle\langle \text{FS}, 1.1\rangle, 3\rangle\}$ is the most recent one.

Efficient Implementation

The generalized product used to find all permutations that could serve as a working set is very expensive. The number of workingsets increases very fast so this is no feasible way of implementation. In this subsection I describe an algorithm to generate all workingsets incrementally. By ordering builds in decreasing temporal order, only the workingsets have to be generated that come before the one that will be used.

The algorithm is displayed in Algorithm 4. The function WORKINGSET takes a source tree *t* and incrementally searches for a valid working set in order to build it. It does this by maintaining a *cursor* that indicates the start of a search window over the set of successful builds (*Success*). Builds are ordered in time so that $\mathit{Success}_0$ is the first build maintained by the system, and $\mathit{Success}_{|\mathit{Success}|-1}$ is the last build. The cursor is moved from the last build downwards towards the first one in the outer loop (line 3) of the algorithm.

In the body of the loop the variable *todo* is initialized with the components we have to find builds for, i.e., the components required by *t* (line 4). Additionally, the current workingset *w* is initialized to be empty and a cursor *within* the window will be 0 (*i*).

The inner loop (line 5) iterates over every successful build in the current search window as long as *todo* is non-empty. A successful build is retrieved by indexing *Success* on $\mathit{cursor} - i$. If this index is below zero, however, we have exhaustively searched through *Success* without finding a suitable workingset, so we fail by returning the empty set (line 6). Otherwise, *b* will contain the *i*th build in the current search window (starting at *cursor*). If the component of this build *b* (*c*) is in *todo*, it is added to the current workingset *w* and *c* is removed from *todo*. Upon normal loop exit, *todo* is empty and *w* represents a workingset candidate. If the extent of *w* through *Integration* is pure, the workingset candidate is valid and *w* is returned as the result of WORKINGSET (line 17). Because the search window is moved downwards, we postulate that *if* a workingset is found, it will be the most recent one. Otherwise, if *w* is not valid, the search window is moved down one position and the outer loop starts anew.

Algorithm 4 Incremental search for the latest working set

```
 1: function WORKINGSET(t)
 2:     cursor ← |Success| − 1
 3:     loop
 4:         w ← ∅; i ← 0; todo ← Requires[t]
 5:         while todo ≠ ∅ do
 6:             if cursor − i < 0 then return nil
 7:             end if
 8:             b ← Success_{cursor−i}
 9:             c ← component(tree(b))
10:             if c ∈ todo then
11:                 w ← w ∪ {b}
12:                 todo ← todo \ {c}
13:             end if
14:             i ← i + 1
15:         end while
16:         if pure?(Integration*[w]) then
17:             return w
18:         end if
19:         cursor ← cursor − 1
20:     end loop
21: end function
```

5.5 Evaluation

I have validated the simple form of backtracking in the context of the Sisyphus continuous integration and release system [103]. True backtracking has been implemented just recently and as a consequence no interesting data is available yet. However, the implementation of simple backtracking has delivered fruitful results.

To evaluate simple backtracking, I have compared build statistics derived from the database maintained by Sisyphus over two consecutive periods of 32 weeks. During this period Sisyphus continuously integrated the ASF+SDF Meta-Environment [92]. The Meta-Environment is an integrated development environment for developing source code analysis and transformation tools. It consists of around 60 components and is implemented in Java, C, and several domain specific languages. Figure 5.5 shows an example integration graph of a subsystem of the Meta-Environment, called `pandora`. The nodes represent successful builds, and the edges represent dependencies between those builds. The clusterering of nodes indicate build cycles. The figure shows that certain builds are used across build cycles.

General statistics about the two periods of time are collected in Table 5.1. In this table I have counted the number of revisions, the number of successful builds, the number of failed build and the number "not tried" builds. The total number of builds is shown as well. Although in the period that simple backtracking was enabled, the number of component revisions was one third fewer than in the previous period, the number of failed builds has decreased by roughly 43% and the number of "not tried"

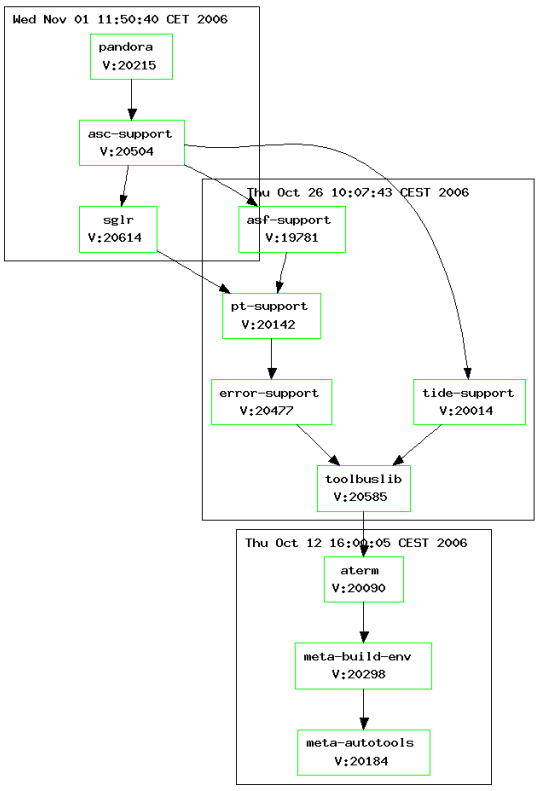


Figure 5.5: Integration graph of the `pandora` subsystem of the ASF+SDF Meta-Environment

	No backtracking	Simple backtracking
#Revisions	1497	1025
#Success	11565	9391
#Failure	1074	507
#“Not tried”	4499	1163
#Builds	17138	11061

Table 5.1: Build statistics over two consecutive periods of time of 32 weeks

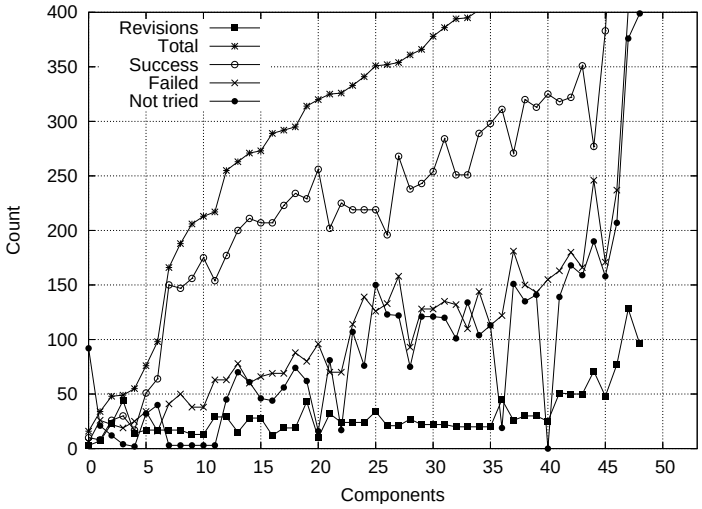


Figure 5.6: Build results without backtracking

builds has decreased even stronger, by around 74%. These absolute numbers suggest that on average the build feedback has improved considerably.

The amount of not tried builds per component is dependent on the component architecture of the system and how the architecture evolves. For instance, a component that has no dependencies will never be “not tried”. The consequence of this is that “not tried”-ness has a cumulative effect if there is no backtracking. If a component build fails, *every* component requiring that build, transitively, will be assigned the status of “not tried”. Clearly there is considerable gain if this can be avoided.

To show the propagating effect of build failures I have plotted the number of revisions, successes, failures, “not trieds” and builds per component to see how these numbers relate to the position of a component in the dependency graph. The plot for the first period—no backtracking—is shown in Figure 5.6. The X-axis of this plot represents the different components of the Meta-Environment. They are sorted according to the total number of builds. This way of ordering components is a raw estimate of position in the dependency graph. If this dependency graph would have been stable, this ordering corresponds to the topological sort of the graph. However, components are added and removed, and dependency relations may change in between integration cycles. It would therefore make no sense to use the topological sort. A similar plot for the second period of time, with simple backtracking enabled, is displayed in Figure 5.7.

If we compare the two figures, what stands out the most is that the number of builds (total, success, failed and “not tried”) in the period without backtracking grows much steeper than in the period of backtracking if the components are higher up in the dependency graph. A second observation is the relative large *and growing* distance between the number of total and successful builds. This is precisely the cumulative effect of build failures.

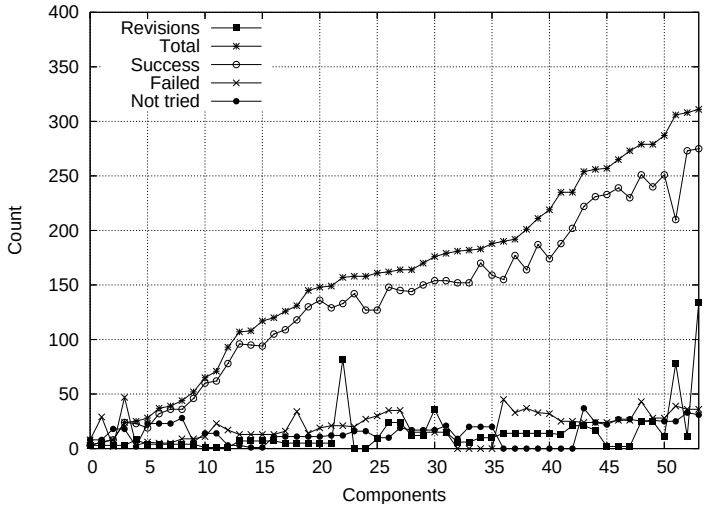


Figure 5.7: Build results with simple backtracking

In the period with simple backtracking, however, as displayed in Figure 5.7, the build figures grow much slower and the distance between the total number of builds and the number of successful builds is more or less constant. In addition, the line indicated "not tried" builds is almost flat. This means that even the simple form of backtracking almost completely eliminates the problem of build failure as an impediment to delivery. These results do not say anything about whether the actual integrations are optimal with respect to up-to-dateness. Still, changes could be missed in an integration. Unfortunately it is not possible to use the currently available revision history to simulate the operation of true backtracking because development activity itself is dependent on the results published by the continuous integration. In other words, one would need a model of how the software would evolve if true backtracking had been enabled in the continuous integration system. The construction of such a model is complicated by the fact that it is very hard to quantify how the behaviour of software developers is affected by backtracking continuous integration. Future work will have to show whether true backtracking is optimal in this respect.

5.6 Related Work & Conclusion

5.6.1 Related Work

Continuous integration has received very little attention from the research community; we only know of Dolstra [30], who describes the use of the deployment system Nix as a continuous integration system with similar goals as Sisyphus. Additionally, Lippert *et al.* [67] describe the implementation of a continuous integration system as means for realizing the *practice* of continuous integration. The lack of attention is surprising

since there exists a host of continuous integration systems, both commercial and freely available. For an overview the interested reader is referred to:

```
http://damagecontrol.codehaus.org/
    Continuous+Integration+Server+Feature+Matrix
```

Unfortunately no empirical data on the assumed merit of continuous integration seems to be available as of today although it is widely considered to be a best practice [9, 39, 71].

Incremental building, or selective recompilation, goes back to Make [37] and has been researched quite vigorously; see e.g. [15, 44, 49, 65]. This work, however, mostly considers dependencies on the level of files. Determining whether a file requires recompilation mostly involves checking timestamps of cryptographic hashes. In this work, however, we compare actual revisions of version control system (VCS) to a database storing accurate bills of materials (BOMs) [70] of all past builds.

Build optimization is another area of related work. Caching build [89], distributing builds [78] and build paralellization [3], header restructuring [25, 110] and precompilation [111] mostly optimize towards minimizing resource consumption. In this chapter I try to optimize towards improved feedback and maximum release opportunity. Of course, both goals are not mutually exclusive.

5.6.2 Conclusion

The subject of this chapter is to improve automatic continuous integration in component-based development settings in order to maximize feedback and maximize release opportunity. I introduced an algorithm for incremental continuous integration and subsequently extended it with “simple backtracking” and “true backtracking” to make the integration process more resilient with respect to build failures. Finally I discussed some empirical results that were obtained from a running implementation of simple backtracking. These results show that even the simple backtracking algorithm almost completely neutralizes the cumulative effect of build failures. Future work will have to show how true backtracking improves this situation. The true backtracking algorithm is still highly experimental and will require further study in order to positively claim that it behaves as expected. Additionally, it is not clear what the worst-case complexity of the algorithm is. Finally, we will generalize the binding of component requirements to source trees. Currently, this binding was implicit: every component has a single designated source location. However, if components have multiple development branches, this means that builds are executed for a single branch per component only. By making binding of components to source locations a first-class concept, the continuous integration system could integrate different compositions with different branch organizations.

Chapter 6

Techniques for Incremental System Integration

Abstract Decomposing a system in smaller subsystems that have independent evolution histories, while beneficial for time-to-market and complexity control, complicates incremental system integration. Subsystem teams may use different versions of interfaces of required dependencies for maintaining, building and testing a subsystem; these differences in assumptions surface during integration as version conflicts. We present techniques for understanding such conflicts in and describe how they can be used to prevent additional effort to resolve them. Our techniques represent first steps towards explicitly managing subsystem dependency and interface compatibility as first-class software configuration management concepts.

This chapter is joint work with René Krikhaar and Frank Schophuizen and has been submitted to *14th Conference on Reverse Engineering (WCRE'07)*, 2007

6.1 Introduction

Developing a large complex system by decomposing the system in smaller parts such as components or subsystems is beneficial both for complexity control and time to market. Different development teams work on different parts, so that changes are localized. The gain in productivity comes from an increase in parallelism between teams.

There is, however, a downside to this mode of development. Each subsystem evolves more or less isolated from the rest of the system, but often has dependencies on different parts of the system. For instance, to build and test the subsystem, at least the interfaces of its dependencies are required.

The independence of subsystem development activities, while beneficial for time-to-market, has an impact on the effort of integration. Putting all subsystems together

to form the final system is not straightforward because of the different assumptions subsystems have been built with. One subsystem could have been built using a different version of a dependency than the version used by another subsystem. Such version conflicts are common and are an impediment to frequent and efficient integration.

In this chapter we investigate the costs of incremental system integration for C/C++ based systems in the setting of component-based Software Configuration Management (SCM). We assume that subsystem builds are executed by individual teams and that system integration takes these builds as a starting point to compose the whole system. Conflicts are resolved by rebuilding subsystems in order to normalize the set of interfaces that is used. By analyzing the build results of various versions of subsystems and the process of incremental integration, we derive the space of possibilities of resolving conflicts this way. Based on this solution space we can assess the relative cost of different rebuild strategies using the *build penalty* metric. This metric represents the impact of each of the elements in the solution space, expressed in the number of subsystem rebuilds that are required.

The build penalty is then detailed using another metric, the *parsed lines of code* (PLOC). The PLOC metric can be correlated to the actual time a subsystem takes to build. This way a more accurate estimation of the cost of a particular resolution strategy is obtained.

Finally we discuss how interface compatibility can be exploited to achieve even less required rebuilds in case of conflict.

Contributions The contributions of this chapter can be summarized as follows:

- We define the notion of *Bill of Materials* (BOM) [70] in the field of software engineering. We show how this notion plays a steering role during the integration phase of the software construction process. This concept proves crucial in the formalization of the build penalty and PLOC metrics.
- The build penalty metric is applied in a case-study at Philips Medical Systems. The results show that for a number of past integrations the effort involved has been less than optimal.
- We speculate on how SCM systems could be extended to support incremental integration by explicitly managing subsystem dependency and interface compatibility.

Organization of this chapter This chapter is organized as follows. In Section 6.2 we introduce preliminary terminology derived from the Unified Change Management methodology [10]. We also describe the incremental integration process at our industrial partner Philips Medical Systems and how integration conflicts are dealt with there.

Then, in Section 6.3 the concepts of integration and integration conflict are formalized using with the BOM concept. Using the definition of BOM we define the build penalty metric in and We explain how this metric is used to find an efficient integration strategy.

In Section 6.4 we discuss the interface compatibility and subsystem dependency as a first-class SCM citizen. Furthermore, we refine the definition of build penalty to

take interface compatibility into account in and conjecture that this is beneficial for incremental system integration without inducing extra costs.

Our work is strongly positioned in the field of build management and SCM. We discuss related work in more detail in Section 6.5. Finally we present a conclusion and some directions for future work.

6.2 Component-Based Integration

6.2.1 Introduction

The defining characteristics of software components proper is that they are units of composition, versioning and replacement [87]. Examples of components are shared libraries (DLLs), .NET assemblies [72], COM components [84] or even executables. Many of the traditional arguments (e.g. reuse and variation, complexity control, third-party acquisition and deployment) for component-orientation apply equally well in the SCM context. The SCM perspective, however, emphasizes an additional argument: factoring a system in separate components creates opportunities for increased parallelism in software development activities. Teams can be more independent and changes are more localized, which in turn may prevent complicated merge processes.

In the following we highlight the software configuration management aspect of components. We will use terminology adapted from the Unified Change Management (UCM) methodology as used in IBM Rational ClearCase [10]. The following definitions are relevant to this chapter: subsystem, component, interface and body.

A *subsystem* is an architectural building block of a software system. Subsystems are implicitly versioned in their constituent parts, its components. A subsystem is thus a *logical* clustering of components. A *component* is versioned group of files. In UCM a component corresponds to a directory in a versioned object base (VOB). Components have *baselines* identifying particular versions of the component. Note that this definition differs from the common notion of component as in Szyperski *et al.* [87]. Physically, however, a subsystem may represent a component in that sense (e.g., a COM component or DLL). There are two kinds of component: bodies and interfaces. An *interface* is a component consisting of only interface files (e.g. IDL headers). *Bodies*, on the other hand, are components containing implementation sources of a subsystem. The elements defined in a body are exported through the interface. In our context, a subsystem consists of a body component and an interface component, and is versioned through their respective baselines.

Although these definitions are taken from the UCM approach to SCM, there are other manifestations of the same concepts. For instance, a single C source file could also be seen as an instance of a Body with the corresponding header file as Interface. Therefore, these definitions provide a more or less generic framework for thinking about system integration. Next, we describe the integration process at Philips Medical Systems; this can be seen as a template for large-scale component-based integration of C/C++ systems.

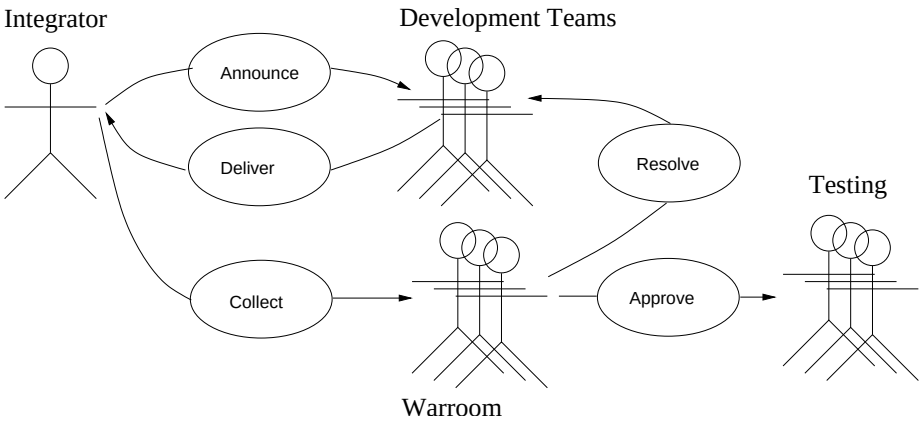


Figure 6.1: Integration work flow

6.2.2 Integration of Subsystems in Practice

Integration is the process of putting all subsystems together to test the complete system. That subsystems have been locally built using different configurations of body and interface baselines becomes visible during this process: some subsystems should be rebuilt to make sure that each subsystem uses the same versions of interfaces every other subsystem uses.

Rebuilding a subsystem induces a (time) cost. Build time in general may not be an issue. Before a product release is put out, in most cases a complete rebuild of all subsystems is required anyway. But doing integration *frequently* within a project is widely considered to be a best practice [39]. Integration should be executed as often as possible, in order to find bugs originating from the interaction between subsystems as quickly as possible. Postponing integration may result in those bugs to be very hard to track down and fix. Therefore, to increase the frequency of integration, it is important that the costs be minimal.

At Philips Medical Systems subsystems are built *locally* by individual teams and the resulting binaries are input to the integration process. The integration process, displayed in Figure 6.1, then unfolds as follows:

Announce The system integrator announces (e.g., by email) to the teams which interface baselines are "recommended". Teams then build their subsystem against those set of baselines.

Deliver The results of subsystem builds are delivered to a shared location. These include a designated file (the BOM) that records how subsystems have been built and the binaries that have been produced. In our case, the binaries correspond to an MSI file that is used by the Microsoft tool "Windows Installer" [74].

Collect The integrator collects the MSIs and associated BOMs. The BOMs together form a SYSTEM BOM. Note that no *system build* is performed during this phase.

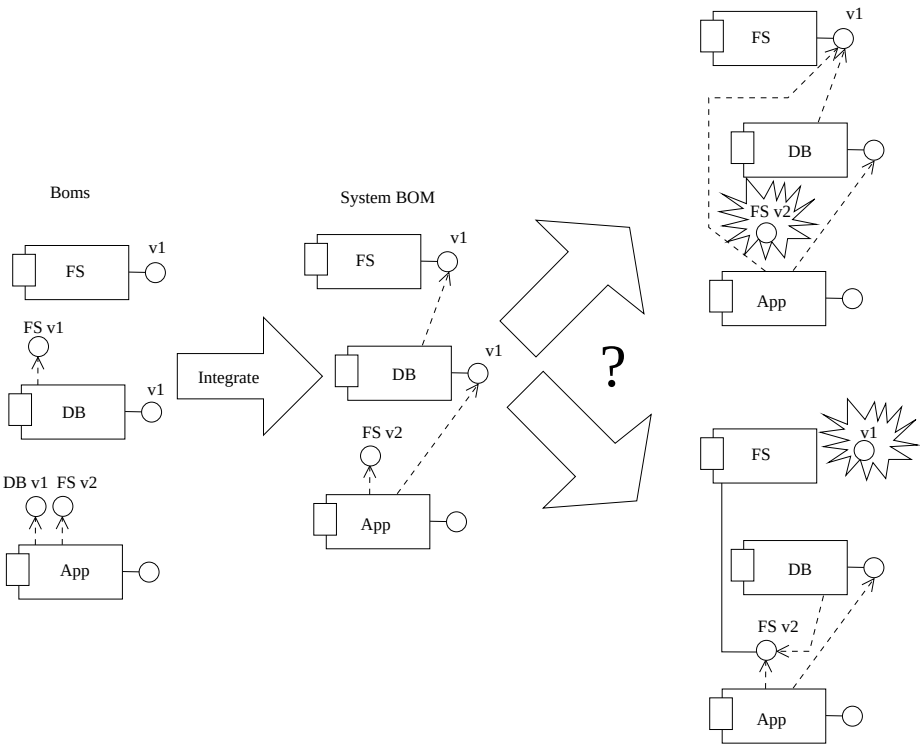


Figure 6.2: Integration conflicts

The MSI files represent components that can be independently deployed; linking occurs at runtime.

Resolve SYSTEM BOMs may contain conflicts; these are reviewed by a (weekly) meeting in the "war room". There, it can be decided that a conflict is not a problem (due to compatibility of interfaces), or that some subsystems have to be rebuilt.

Approve After conflicts have been resolved, the set of (possibly new) MSI files is approved and is ready to go to testing.

In this chapter we assume that integration is *incremental*. This means that if a team does not deliver in the current integration cycle, the build results (BOM and MSI) of the previous integration cycle are used. In the following we will take a closer look at the "Resolve" step and describe the nature of version conflict.

6.2.3 Integration Conflicts

Figure 6.2 visually depicts the "Collect" and "Resolve" phases in the integration process. On the left three BOMs are depicted (in UML-like notation) for three subsystems,

App, DB and FS. For instance, on the bottom left-hand side of the figure, a BOM of subsystem App is shown: it has been built against interfaces v1 and v2 of subsystem DB and FS respectively. The BOM relation is indicated using dashed arrows. For clarity, the versions of the body baselines are not shown.

Each BOM consists of a body baseline coupled with a number of interface baselines that were used in the subsystem build in this cycle. The integrator collects these BOMs to form a SYSTEM BOM, displayed in the middle of the figure. Some used interface baselines become bound to implementations that were built using the same (exported) interface baseline. For instance, the App subsystem used baseline v1 of the DB interface, which was also used during the build of the body of DB. So the requirements of App are matched to the current implementation body of DB via interface baseline DB v1. Similarly the dependency of DB on FS is bound via interface baseline FS v1.

However, both App and DB used different baselines for the FS interface, v2 and v1 respectively and no implementation is found for the dependency of App because FS has been built against v1 of its interface whereas App used v2. This constitutes a baseline conflict that has to be resolved before the system can go into testing.

The right-hand part of the figure will be discussed in Subsection 6.2.5.

6.2.4 Causes of Integration Conflicts

Before describing the possibilities of dealing with integration conflicts in detail, it is instructive to look at the causes of these conflicts: parallel development and local builds.

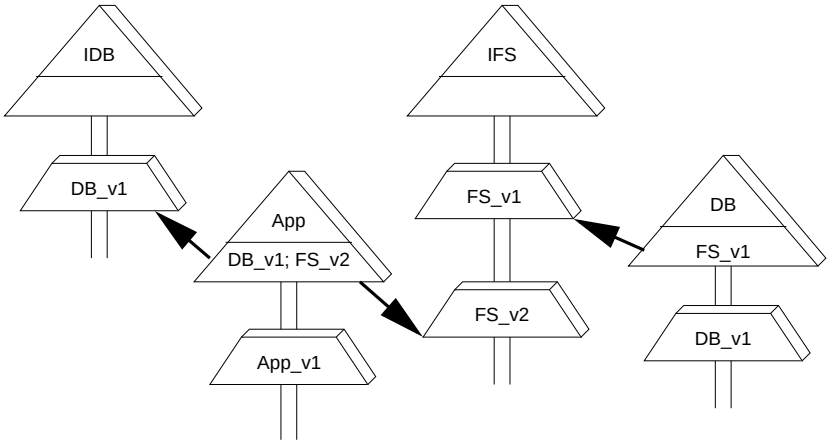


Figure 6.3: Causes of integration conflicts

To take inspiration from the UCM model once more, consider Figure 6.3. The figure shows four development lines (called streams) for two interface components and two body components from the subsystems of Figure 6.2. The top-level triangles are called the *foundation* of the stream. The foundation is the root of a code line, although it may derive from another code line (branching). The trapezoids below the foundation

indicate baselines for the component in question.

In UCM the foundation can contain references to read-only or *shared* baselines. In the figure we see that both App and DB have references to interface baselines in their foundation. For instance, the body baseline App_v1 is configured with interface baselines of DB and FS, and DB_v1 is configured with an interface baseline of FS. The configuration of streams using baselines in the foundation follows the dependencies between subsystems.

From the figure it becomes clear how integration conflicts come about: streams are configured with different baselines for the same interface. In the example we have App being configured with v2 of the FS interface, whereas DB uses v1. The existence of these different configurations is a direct consequence of increased parallelism and independence among subsystem development activities.

Streams can be reconfigured (called *rebasing*) to start using newer interface baselines. This is exactly what is supposed to happen during the announcement phase of the integration process. The system integrator broadcasts the set of recommended baselines, and all teams are supposed to rebase their streams accordingly.

So, how is it possible that integration conflicts surface in the collection phase of the integration process? There are two simple reasons for this:

- The announcement of recommended baselines is a manual process. The recommended baselines are announced by email or personally. The rebasing to this new set of baselines is also manual. So there is some risk of human error.
- Even if all teams rebase to the right set of baselines, a team might not deliver build results during the current integration cycle. In this case build results from the previous cycle are used. These builds often have been executed against a different set of baselines.

It is tempting to solve these problems by automating the rebasing of streams and forcing every team to always deliver new builds. However, this goes against the spirit of the relative autonomy each team has, and may have consequences the team itself is in the position to assess best. Imposing the stream configuration “from above” is a move away from decentralized component-based development. Moreover, forcing subsystem builds every cycle, even when there are no interesting changes, will consume valuable time of the team members and constitutes a move from incremental integration to big bang integration. Precisely the ability to reuse previous build results saves a lot of time. We therefore take conflicts as a given and then try to minimize their impact.

6.2.5 Resolving Conflicts

To illustrate the ways of resolving a conflict, let’s again turn our attention to Figure 6.2. The right-hand side of the figure shows the possibilities for resolving the conflict. There are two options to choose from, as highlighted by the question mark. First, the upper resolution shows that baseline v1 is chosen of baseline v2 of the FS interface. This means that possibly App should be rebuilt, this time using interface baseline v1. The second (lower) option shows the alternative choice: v1 is dropped and v2, the newer baseline, is chosen. Although it seems natural to always choose the newer alternative

in a conflict, there may be external considerations that make this choice less desirable. For instance, when choosing v2 in the example, both App and DB should be rebuilt.

Both choices are valid, depending on the circumstances. If the changes between v1 and v2 of the FS interface are important or if they are compatible – we come back to this in Section 6.4 –, one would probably choose v2 and request a rebuild of both App and DB. However, if the changes are minor, and successful integration is more important, it might be wise to minimize build time and choose v1.

We have experienced that the way such conflicts are resolved are implicit, ad hoc and inefficient. Basically, two strategies are applied. The first, and most obvious, solution is to rebuild every subsystem that is in conflict against the newest baselines in the SYSTEM BOM. This boils down to reinitiating the integration cycle, by again requesting rebuilds of certain systems with certain baselines. Once again, MSIs and BOMs are collected, and the SYSTEM BOM is reviewed.

The second strategy may be followed during the review meeting in the war room. During that meeting, the participants may decide that a conflict is not a real problem because interface changes are backwards compatible. For instance, referring to the example of Figure 6.2, this might mean that v2 of the FS interface is backwards compatible with v1, so that, as a consequence, neither FS nor DB have to be rebuilt.

Both strategies have a negative impact on incremental system integration. Just rebuilding all bodies in conflict is a waste of resources that can be prevented. On the other hand, the war room decision that two baselines are compatible is a violation of basic SCM principles: the decision is informal, untraced and resides solely in the heads of the employees involved. There should be a formal record of the rationale(s) for such decisions. Automation provides such rationale.

6.2.6 Solution Overview

Frequent incremental integration is an important practice, but so is parallel development. In our approach, we take conflicts for granted, but attempt to minimize their impact by finding the minimal cost resolution strategy.

Our solution is twofold. The first component of our approach consists of formalizing the build cost (in number of rebuilds and approximate time) in order to assess the impact of conflict resolution. This gives use the “cheapest” way of proceeding with the integration process in case of conflict, by computing the minimum set of rebuilds. This is discussed in detail in Subsection 6.3.4.

Secondly, the notion of compatibility can be used to prevent rebuilds entirely. This notion, however, should be an explicit SCM citizen in order to leave no ambiguity with respect to what system configurations enter testing. Interface compatibility is explored in Section 6.4 where we tentatively refine the notion of build penalty to take interface compatibility into account.

6.3 Towards Automating Integration

6.3.1 Introduction

In this section we will formalize the notion of *Bill of Materials* [70] in the field of software engineering. We show how this notion plays a steering role during the integration phase of the software construction process. We will use a relational approach to formalization, which entails that the entities of interest are modeled by sets and relations between those sets. Common operations on binary relations include, domain, range and right image, where the right image of a relation R for a set X – denoted by $R[X]$ – gives all elements of the range of R that have a corresponding element $x \in X$ in the domain of R. Finally, some standard set operations we use include cardinality ($|X|$) and set product ($X \times Y$). The formalization enables automated reasoning and computation with the concepts involved.

6.3.2 Bill of Materials

First, some preliminary concepts have to be formalized. A subsystem is identified by name and has an interface and a body. The set of all interfaces is called I and the set of bodies is designated by B. Interfaces and bodies are versioned in baselines i_v, b_v which are contained in the baseline sets I_V, B_V respectively. Two possibly different baselines for the same interface (body) are denoted by i_v (b_v) and i_w (b_w).

Subsystem bodies are built in a context that consists of interface baselines for each subsystem that is required including the interface exported by the body itself. This set of interfaces will be used to satisfy dependencies during builds. If such a build is successful, this results in a BOM; it records the version of the subsystem body and which versions of imported interfaces were used. Formally BOMs are defined as follows:

$$bom \subseteq B_V \times I_V \text{ such that } |\text{domain}(bom)| = 1 \tag{6.1}$$

Thus, a BOM can be seen as a relation that relates a *single* body baseline of a certain subsystem to a number of interface baselines, hence the side condition. The set range(bom) are the imported interfaces used in that particular build of b_v; it includes a baseline for the interface exported by b_v.

BOMs capture closures. That is, all *transitively* imported interfaces should be listed in a BOM. For BOMs to be a proper identification mechanism for builds, *all* inputs to a build must be part of it. If an interface file (e.g., a header) itself includes another interface, this second file also contributes to the end-result of the build. A BOM thus should identify both the version of the sources of the body, as well as the versions of transitively imported interfaces.

In the example of Figure 6.2, there are three BOMs represented by the following relations:

$$\begin{aligned} bom_{\text{FS}} &= \{\langle \text{FS}, \text{IFS}_{v1} \rangle\} \\ bom_{\text{DB}} &= \{\langle \text{DB}, \text{IFS}_{v1} \rangle, \langle \text{DB}, \text{IDB}_{v1} \rangle\} \\ bom_{\text{App}} &= \{\langle \text{App}, \text{IFS}_{v2} \rangle, \langle \text{App}, \text{IDB}_{v1} \rangle, \langle \text{App}, \text{IApp} \rangle\} \end{aligned}$$

The BOMs can be used to identify certain builds: if two BOMs are equal, the corresponding builds are equal, and consequently the corresponding binaries are equal. This can be seen as an additional level of versioning on top of the versioning facilities offered by the SCM system.

6.3.3 Integration

The dependencies of a subsystem are *parameterized dependencies* [87]. This means that a subsystem body is only allowed to depend on interfaces; the implementation of these interfaces is a parameter. There may be multiple implementations for the same interface. Putting a system together thus means that every interface baseline used in a BOM must be matched to some body that exports that interface. This process is *integration* proper, which we describe next.

The starting point for integration is a set of BOMs, since only subsystems that have successfully been built can be integrated. The objective is to select a set of BOMs such that there is a BOM for every subsystem, and that every dependency is satisfied. Such a selection is called a SYSTEM BOM. In the ideal situation any referenced interface baseline in the dependency section of a BOM is bound to an implementation via another BOM in the system, but because of independent development of subsystems this may not always be the case. For instance, in Figure 6.2, the dependency of App on FS cannot be bound, because the body of FS has been built using a different version of the interface of FS.

The SYSTEM BOM is defined as follows:

$$sysBom \subseteq B_V \times I_V \text{ such that } |\text{domain}(sysBom)| = |B| \tag{6.2}$$

Thus, a SYSTEM BOM is created by taking the union of all the BOMs under consideration. Every body in the system has a baseline representative in the SYSTEM BOM. Note that it *is* possible that $|\text{range}(sysBom)| \neq |I|$. In other words, some interfaces have multiple baseline representatives in the SYSTEM BOM. If this is the case then the SYSTEM BOM contains conflicts that should be resolved. In the following we assume *sysBom* to designate the SYSTEM BOM under consideration.

In the context of the example $sysBom = bom_{\text{App}} \cup bom_{\text{DB}} \cup bom_{\text{FS}}$. Note that the domain of this relation has the same size as the set of subsystems (App, DB, and FS) but that the range is larger than the set of interfaces. Therefore, this SYSTEM BOM contains a conflict. As mentioned before, there is more than one way to resolve it, so we will now describe how to select a single solution that has minimal cost in terms of build resources.

6.3.4 Resolving Conflicts

We introduce the build penalty metric to measure the cost of conflict resolution. Conflict resolution consists of selecting a *single* interface baseline for each interface, so that every imported interface can be bound to a body that exports that interface. This is formalized below.

Formally, we define a conflict C_i to be a set of baselines for the same interface i present in the SYSTEM BOM. We defer the discussion of the influence of interface

compatibility to Section 6.4 and for now assume that the interface baselines in a conflict are incompatible. A conflict can then be resolved by choosing a single baseline from the alternatives in the conflict. A global solution amounts to choosing a single baseline for every conflict in the SYSTEM BOM. There may be more than one global solution.

Derive the set of conflicts from a SYSTEM BOM as follows:

$$\forall i \in I : C_i = \{i_v \in \text{range}(\mathit{sysBom})\} \tag{6.3}$$

If interface i is not in conflict, then $|C_i| = 1$. If the set of conflicts is not a singleton, then the elements in it are alternatives for resolving all conflicts at once. In our example there is only one conflict, between the two interface baselines of the FS subsystem:

$$C_{\text{IFS}} = \{\text{IFS}_{v1}, \text{IFS}_{v2}\}$$

Conflicts introduce choice between interface baselines. The set of all possible solutions can be computed by:

$$\mathit{Solutions} = \prod_{i \in I} C_i \tag{6.4}$$

A solution $S \in \mathit{Solutions}$ is a n-tuple ($n = |I|$) which corresponds to a selection of interfaces where each interface in the selection belongs to exactly one C_i; below we interpret a solution S as a set. The number of alternatives equals $\prod_{i \in I} |C_i|$. The solution space in the example is the following ternary relation:

$$\{\langle \text{IFS}_{v1}, \text{IDB}_{v1}, \text{IApp}\rangle, \langle \text{IFS}_{v2}, \text{IDB}_{v1}, \text{IApp}\rangle\}$$

The elements of such solution spaces can be sorted according to build penalty which yields the configuration(s) that costs least to construct. This is described next.

6.3.5 Minimum Build Penalty

Take an arbitrary solution S which contains baselines for every interface in the system. Build penalty corresponds to what has to be rebuilt as a consequence of choosing a certain solution. The rebuild criterion for a body b_v and solution S is defined as:

$$\text{rebuild?}(b_v, S) = \mathit{sysBom}[b_v] \not\subseteq S \tag{6.5}$$

That is, the used interfaces in the build of b_v should be part of the solution, and otherwise it should be rebuilt. Using the rebuild criterion we can now the define the set of bodies that require a rebuild:

$$\text{rebuilds}(S) = \{b_v \in \text{domain}(\mathit{sysBom}) \mid \text{rebuild?}(b_v, S)\} \tag{6.6}$$

Then the build penalty is defined as the number of rebuilds required:

$$\text{build-penalty}(S) = |\text{rebuilds}(S)| \tag{6.7}$$

The set of solutions can now be sorted in ascending order according the associated build penalty. At the top of the resulting list will be the optimal choices for resolving

the integration conflicts. If there is more than one optimal solution, one can reduce this set to a singleton, by taking the "newest" solution. This is done by summing the versions and/or timestamps of the baselines in the solutions and then comparing them.

To turn our attention once more to the example of Figure 6.2, we can assign a build penalty to all elements of the solution space. In this case, the first solution, the one containing IFS v1 has a build penalty of 1; only App has to be rebuilt. The second solution, where both DB and FS have to be rebuilt has a penalty of 2. Thus, the latter is less optimal with respect to integration effort.

Note however that the second solution may be preferable because it is more "on the bleeding edge" that the first solution, since it includes changes between v1 and v2 of IFS. If these changes turn out to be compatible the build penalty metric should take this into account and select the second solution in favor of the first one. We elaborate on this in Section 6.4.

6.3.6 Evaluation

In order the evaluate the concept of minimal build penalty, we have computed it for 86 system integrations at a business unit of Philips Medical Systems and compared it to the build penalty induced by taking the newest baseline for every interface in conflict, the "newest" strategy.

The results are shown Figure 6.4. The plot shows that the minimum build penalty is consistently below or equal to the "newest" build penalty. The difference between the penalty of the "newest" strategy and the minimal strategy equals the number of builds that would have been saved if the minimal strategy had been chosen.

6.3.7 Parsed Lines of Code (PLOC)

Build penalty abstracts over the real, wall-clock time that a subsystem build takes. However, it could occur that rebuilding three subsystems actually takes less time that rebuilding one or two. It is therefore beneficial to also know the actual time it takes to build a subsystem in order to make a better decision when resolving integration conflicts.

For C-like languages, we define the Parsed Lines Of Code (PLOC) metric to capture the number of lines to be processed when all include files are textually expanded in the source file without counting duplicate includes. The total number of lines is calculated by considering all files needed to compile a unit, i.e. both source file and included files (whether flagged or not by preprocessor statements). This is a worst case estimate of lines of code to be interpreted by the compiler after preprocessing the source file.

It is clear that the PLOC count can be an order of magnitude larger than the normal LOC count. Also, slight changes in the import structure of a subsystem can have major consequences for PLOC whereas they are almost invisible in a simple LOC count. The order of magnitude in which the PLOC is larger than the LOC is more or less constant in time. Experience at Philips Medical Systems shows that the PLOC metric is a better measure to estimate build time and related research corroborates this [110].

In the context of this chapter, a compilation unit corresponds to a BOM. Recall that BOMs link a certain subsystem body with the set of interface baselines that were

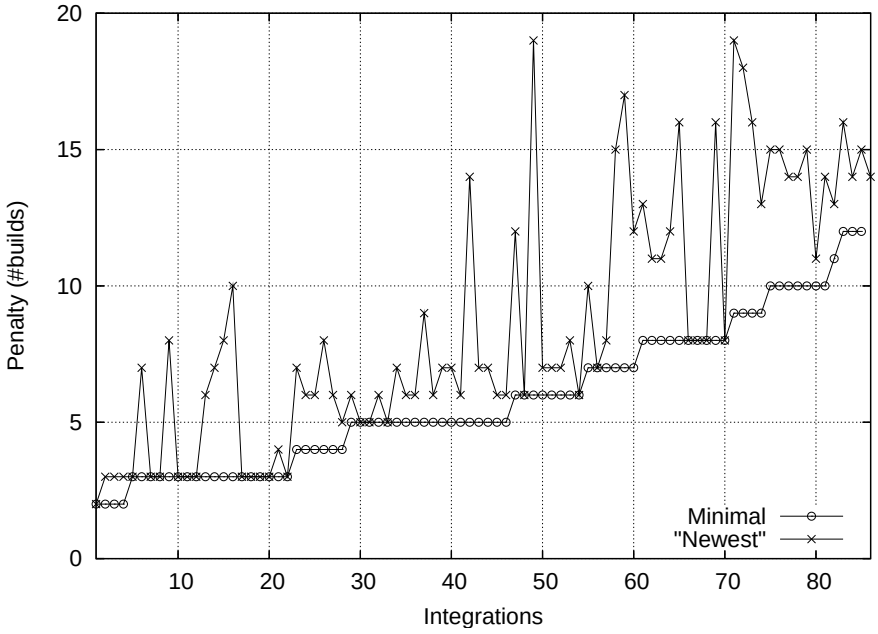


Figure 6.4: Minimal build penalty compared to “newest” build penalty

(transitively) imported during a build (Definition 6.1). The PLOC metric can now be defined as:

$$\text{PLOC}(b_v) = \text{LOC}(b_v) + \sum_{i_v \in sysBom[b_v]} \text{LOC}(i_v) \tag{6.8}$$

For a certain body it estimates the number of lines that have to be recompiled should this body have to be rebuilt. Note that in case of a rebuild, different baselines of the interface dependencies (different from $sysBom[b_v]$) will be used. However, since BOMs record transitive imports, we can be sure that the PLOC count based on the old situation will not deviate much from the situation during the rebuild. To be completely accurate the PLOC count could be computed from build context of the *re*build of the body corresponding to the chosen solution.

6.3.8 PLOC Build Penalty

The build penalty, in terms of PLOC to be recompiled, after changing a single file is the sum of the PLOC of the compilation units in which the file participates. Per file that should be rebuild we can calculate the build penalty in terms of PLOC. This gives us the combination of build-penalty and PLOC, the ploc-build-penalty:

$$\text{ploc-build-penalty}(S) = \sum_{b_v \in \text{rebuilds}(S)} \text{PLOC}(b_v) \tag{6.9}$$

This function sums the PLOC counts for every subsystem body that should be rebuilt according to S. It can now be used to rank the solutions in the solution space, in a similar way that build-penalty was used to rank solutions. We have, however, not validated this in practice.

6.3.9 Evolutionary PLOC

Another interesting aspect of PLOC is to calculate the effect of the system's total PLOC on a per file basis. In case a file is changed in the version management system, it will result in a number of files (client code) that will have to be rebuilt to accommodate and test the changes.Each such file can be ranked according to the build penalty. A header file with a high build penalty (in terms of PLOC to be recompiled) is interesting to analyze. In case the file is often changed, as can be derived from the SCM system, one may consider to reduce the PLOC of this file, by splitting up.

As an example, at Philips Medical System there used to be a file in the software of a medical system containing many data definitions that had remained there just because of historical reasons. Inspection lead us to conclude that this file could be easily split up in seven parts. The generic part did not change that often, all other parts changed now and then but had a close relationship with the related subsystem. It resulted in a much lower PLOC count which on average turned out to be beneficial for the build time.

PLOC could be used to spot such refactoring opportunities. Especially if the PLOC metric is multiplied with the frequency of change of a certain subsystem. Then, we conjecture, this is probably a subsystem that has high coupling and weak cohesion.

This is another reason that understanding the dependencies and the impact of build time is important.

6.4 Discussion: Compatibility

6.4.1 Introduction

In the previous section we introduced metrics to better assess the cost of resolving baseline conflicts during integration. The cost of integration is reduced because the metrics indicated when the additional work involved in resolving conflicts was minimal. However, building and testing still remains expensive in terms of time and man-power. Therefore, we would like to minimize the build penalty even more, by preventing builds altogether. This can be achieved by considering *interface compatibility* as a first-class SCM citizen. Explicit knowledge of the stability of interfaces can be used to make integration more efficient.

We can allow some flexibility by establishing interface compatibility. For instance, in the case of COM [84] IDL-interfaces, the notion of binary compatibility of two interface baselines i_v and i_w entails that a body built against the required interface i_v, will work correctly at runtime with a body providing i_w. So, if we can syntactically derive this compatibility relation, it can be used to weaken the rebuild criterion, thereby requiring fewer rebuilds as a result of conflicts. We consider the construction of a tool that derives the binary compatibility relation as future work. However, the consequences for build penalty are discussed below.

6.4.2 Compatibility as SCM Relation

In traditional version models for SCM a distinction is made between version space and product space [20]. These two spaces correspond to the temporal versioning (successor relation between revisions) and spatial versioning (branch and merge relations between code lines). Component orientation in SCM adds another relation, the *dependency* or *use* relation. In the presence of first-class interfaces this relation has two aspects: import relations and export relations, which both can be derived from the BOMs discussed in Section 6.3. Normally, import and export relations are unversioned. For instance, in the case of imports, `#include` directives normally do not specify the precise version of the included header file. Which version is used is a function of the build process.

The notion of compatibility adds a level of flexibility to the integration process. To fully appreciate this, consider Figure 6.5. It shows three code lines, one for an interface X, one for the corresponding body, and one for a body Y that uses the interface X. The circles represent body and interface baselines.

The figure shows the normal product space and version space relations commonly found in any SCM system. However, the import, export and compatibility relations are shown as well. The solid arrows depict import relations and the dashed arrows are export relations, both derived from BOMs. The dotted clusterings indicate compatibility classes. For instance, the clustering of baseline 1.1 and 1.2 of interface X means that 1.2 (the successor of 1.1) is (backwards) compatible with 1.1.

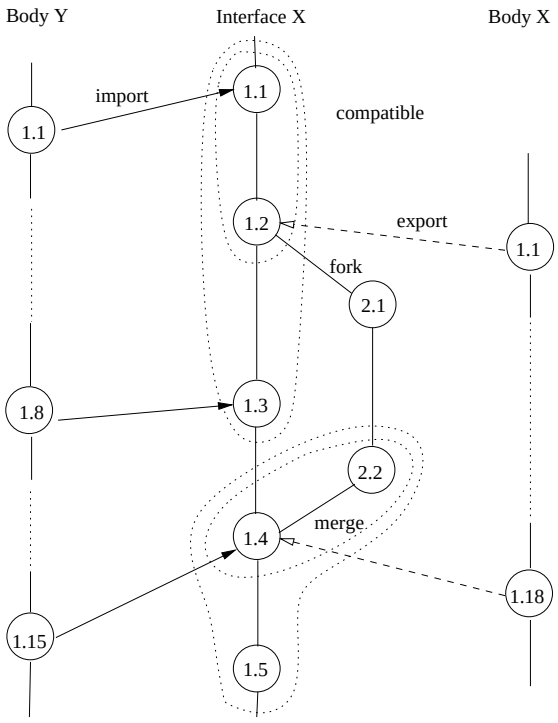


Figure 6.5: Compatibility as part of SCM

Managing the compatibility relation explicitly as part of the general SCM framework, weakens the constraints on integration. Not only interfaces occurring in a BOM of a body are allowed to contribute to the integrated system, but also the ones that are compatible with them. Visually this means that one is allowed to redirect import arcs forward in time within its compatibility class, and redirect export arcs backwards in time within its compatibility class.

For instance, in Figure 6.5, body *Y* with version 1.1 has been built against baseline 1.1 of interface *X*. But the nearest body of *X* (1.1) has been built using baseline 1.2 (of interface *X*). Because version 1.2 of interface *X* is backwards compatible to version 1.1, the system can be composed using body *X* version 1.1 without rebuilding body *Y*. Note that both version 1.2 and 1.1 of interface *X* are eligible, because also the export arc from body *X* 1.1 can be moved backwards.

On the other hand, version 1.8 of body *Y* uses version 1.3 of interface *X* but there is no possibility of moving the import arc forward in time within the same compatibility class. Also, no export arc from body *X* can be moved backwards to arrive at the same baseline. This probably means that either body *Y* will have to be rebuilt against version 1.2 of interface *X*, or that body *X* 1.1 has to be rebuilt using 1.3 of interface *X*.

The figure also depicts a temporary branch. As can be seen from clustering of baselines, the branch introduces an incompatible change in interface *X*. However, after the branch is merged to the main line again, the new baseline on the mainline is backwards compatible with baseline 2.2 of interface *X* on the branch. Although both body *Y* 1.15 and body *X* 1.18 used version 1.4 of interface *X* during their builds, the import arc originating from 1.15 can be moved forward, and the export arc from 1.18 can be moved backward. This may be of value from a larger perspective when more subsystems play a role. From what the figure shows, the build penalty in this case is zero. The BOMs of other subsystems still may require one of the arrows to be redirected, and, consequently either body *Y* 1.15 or body *X* 1.18 may have to be rebuilt.

6.4.3 Requirements of SCM Systems

In order to automate much of the reasoning described in the discussion of the example of Figure 6.5, SCM systems should be extended to record dependency and compatibility information in addition to the traditional historic version information. We briefly list the requirements here:

- Interfaces and bodies must be first-class and distinguished entities in the version control system. Without the distinction the notion of parametric dependencies breaks down.

- It should be possible to configure the system with a compatibility criterion between interface baselines for the language of the domain (e.g. COM IDL). This serves as an oracle for the system to derive the compatibility classes of Figure 6.5.

- Systems should record which bodies export and/or import which interfaces as a result of successful builds and/or a syntactic analysis (that is through a use/define analysis and applying). This way the solid and dashed arrows of the figure are

maintained as part of the repository and can be subject to (historic) automated analysis.

If these requirements are satisfied and the required dependency and compatibility relations are readily available from the SCM system, this knowledge can be used to automatically reduce the set of required rebuilds if conflicts should occur. Below we tentatively describe how the build penalty approach is adapted to support this.

6.4.4 Weakening the Rebuild Criterion

We define a relation between interface baselines belonging to the same interface $i_v \sqsubseteq i_w$ ("baseline w of interface i is compatible with baseline v") that captures compatibility between two concrete interfaces such that i_w can be substituted wherever i_v is used. This relation is reflexive and transitive. The definition of $\sqsubseteq$ can be extended to a relation between sets of interfaces as follows:

$$I_V \sqsubseteq I'_V \equiv \forall i_v \in I_V : \exists i'_w \in I'_V : i_v \sqsubseteq i'_w \tag{6.10}$$

Note that $I_V \subseteq I'_V$ implies $I_V \sqsubseteq I'_V$. This definition of compatibility is then used to adapt the rebuild criterion of Definition 6.5 as follows:

$$\text{rebuild?}(b_v, S) = \mathit{sysBom}[b_v] \not\sqsubseteq S \tag{6.11}$$

A baseline b_v is only rebuilt if there are no interface baselines in the current solution that are compatible with the ones it has been built against.

6.5 Conclusions

6.5.1 Related Work

Our work belongs to the area of build-management, as part of SCM. SCM in component-based settings has received some attention (see e.g., [21,107]), however, the perspective of incremental system integration is largely missing.

The notion of build-level components is discussed in [28]. There, a lightweight process for improving the build architecture of component-based software systems is described. This kind of refactorings improve the process of incremental system integration, but leaves out the versioning and traceability aspect.

In [110] the authors discuss how to improve build performance by removing false code dependencies in large industrial C/C++ projects. The authors perform just-in-time header restructurings before compilation starts in order to reduce the LOC that needs to be processed by the compiler. The PLOC metric was used to spot opportunities for similar refactorings. Again, the traceability and version aspect are not discussed, whereas we think these are essential for system integration.

The notion of Bills of Materials is well-known in manufacturing [70] but we are aware of only few references in the context of software engineering [34]. Although software production has been viewed as a branch of manufacturing [15], the notion

of a software BOM remains implicit in many discussions of software configuration management [4].

Our approach of formalizing the concept of Bill of Materials is inspired by relational approaches to build formalization [65] and architecture analysis [63].

Such an approach is also taken in the incremental continuous integration system Sisyphus [100,103]. The Sisyphus system maintains a database recording the transitive dependencies used during component builds. Integration results are thus fully traceable to sets of component baselines.

That the build process requires a separate architectural view for understanding the system in itself and how the system is constructed was recognized in [91]. How a system is constructed follows from the build process. In this chapter we used the BOM concept as a formal means for understanding and improving this process. We discussed how our build architecture view could benefit from explicitly managing interface compatibility, which in turn is related to architecture-aware SCM [76].

6.5.2 Conclusion

The ever-increasing complexity of software and the pressure for time-to-market require a divide-and-conquer approach to software development which is exemplified in component-based software development. However, the consequences for incremental system integration are not well understood. In this chapter we showed the importance of the concept of BOM, both for understanding and improving the integration process. The formalization enables a rigorous impact analysis of version conflicts among BOMs. We analyzed the build process in an industrial environment using the build penalty metric and the more fine-grained PLOC metric.

On the other hand, explicit knowledge of the architecture of a system can improve the integration process. We proposed how interface compatibility can be managed as a first-class SCM concept in order to fully exploit such architectural knowledge.

In recent years, both industry and research have put much effort in defining methods for developing software product lines. The relation to SCM, however, has not received enough attention in this research. In our opinion SCM can empower software product line development by providing the right means to support it, both conceptual and technical (see [64]). We consider the closer investigation of dependency and compatibility relations in the context of SCM as future work. New means or extensions of traditional versioning and branching relations could, we conjecture, significantly improve SCM practice in the context of multi-project, multi-component product lines.

Chapter 7

Binary Change Set Composition

Abstract Continuous delivery requires efficient and safe transfer of updates to users. However, in the context of component-based software, updating user configurations in lightweight, efficient, safe and platform independent manner still remain a challenge. Most existing deployment systems that achieve this goal have to control the complete software environment of the user which is a barrier to adoption for both software consumers and producers. Binary change set composition is a technique to deliver incremental, binary updates for component-based software systems in an efficient and non-intrusive way. This way application updates can be delivered more frequently, with minimal additional overhead for users and without sacrificing the benefits of component-based software development.

This chapter has been previously published as: T. van der Storm, Binary Change Set Composition, in *Proceedings of the 10th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'07)*, LNCS, Springer, 2007 [101].

7.1 Introduction

An important goal in software engineering is to deliver quality to users frequently and efficiently. Allowing users of your software to easily take advantage of new functionality or quality improvements can be a serious competitive advantage. This insight seems to be widely accepted [43]. Software vendors are enhancing their software products with an automatic update feature to allow customers to upgrade their installation with a single push of a button. This prevents customers from having to engage in the error-prone and time consuming task of deploying new versions of a software product. However, such functionality is often proprietary and specific to a certain vendor or product, thereby limiting understanding and broader adoption of this important part of the software process.

The aim of this chapter is to maximize the agility of software delivery without sacrificing the requirement that applications are developed as part of a component-based product line. While it may not be beneficial to force the user environment to be component-based, it certainly can be for the development environment. One would like to develop software in a component-based fashion, and at the same time allow users to transparently deploy an application as a whole.

If certain actions are tedious, error-prone or just too expensive, they tend to be performed less frequently. If the effort to package a software product in such a way that it is ready for deployment is too high, releases will be put out less frequently. Similarly, if deploying a new release is a time consuming activity with a high risk of failure, the user probably will not upgrade every day. Therefore, if we want to optimize software delivery this can be achieved by, on the one hand, reducing the cost of release, and on the other hand, by reducing the cost of deployment.

How would one optimize both release and deployment in a platform and programming language independent way, when many products composed of multiple shared components have to be released and deployed efficiently? In this chapter I present a technique, called *binary change set composition*, which provides an answer to this question. Using this technique, applications are updated by transferring *binary* change sets (patches). These change sets are computed from the compositional structure of application releases. It can be used to implement lightweight incremental application upgrade in a fully generic and platform independent way. The resulting binary upgrades are incremental, making the upgrade process highly efficient.

Contributions The contributions of this chapter are summarized as follows:

1. A formal analysis of automatic component-based release and delivery.
2. The design of a lightweight, efficient, safe and platform independent method for application upgrade.
3. The implementation of this method on top of Subversion.

Organization This chapter is organized as follows. Section 7.2 provides some background to the problem of application upgrade by identifying the requirements and discussing related work. Section 7.3 forms the technical heart of this chapter. I describe how to automatically produce releases and deliver updates in an incremental fashion. The implementation of the resulting concepts is then discussed in Section 7.4. Then, in Section 7.5, I evaluate the approach by setting it out against the requirements identified in Section 7.2. Finally, I present a conclusion and list opportunities for future work.

7.2 Background

7.2.1 Requirements for Application Upgrade

Application upgrade consists of replacing a piece of software that has previously been installed by a user. The aim of an upgrade for the user is to be able to take advantage

of repaired defects, increased quality or new functionality. The business motivation for this is that customer satisfaction is increased. To achieve this goal, the primary requirement is that upgrades *succeed*. Nevertheless, there are additional requirements for application upgrade. In the paragraphs below I discuss four requirements: *lightweightness*, *efficiency*, *genericity* and *safety*.

For an software deployment method to be lightweight, means that (future) users of a software product should not be required to change their environment to accomodate the method of deployment of the product. Reasoning along the same lines, the method of creating deployable release should not force a development organization to completely change their development processes. Furthermore, the effort to create a release on the one hand, and the effort to apply an upgrade on the other hand, should require minimum effort.

Efficiency is the second requirement. If the aim is to optimize software delivery, both release and upgrade should be implemented efficiently. If deploying an upgrade takes too much time or consumes too much bandwidth, users will tend to postpone the possibly crucial update. Again, also the development side gains by efficiency: the storage requirements for maintaining releases may soon become unwieldy, if they are put out frequently.

To ease the adoption of a release and deployment method, it should not be constrained by choice of programming language, operating system or any other platform dependency. In other words, the third requirements is *genericity*. It mostly serves the development side, but obviously has consequences for users: if they are on the wrong platform they cannot deploy the application they might desire.

The final and fourth requirement serves primarily users: safety of upgrades. Deployment is hard. If it should occur that an upgrade fails, the user must be able to undo the consequences quickly and safely. Or at least the consequences of failure should be local.

7.2.2 Related Work

Related work exists in two areas: update management and release management,—both areas belong to the wide ranging field of software deployment. In this field, update management has a more user oriented perspective and concerns itself with the question how new releases are correctly and efficiently consumed by users. Release management, on the other hand, takes a more development-oriented viewpoint. It addresses the question of how to prepare software that is to be delivered to the user.

In the following I will discuss how existing update and release tools for component-based software deployment live up to the requirements identified in Section 7.2.1.

Research on software deployment has mostly focused on combining both the user and development perspectives. One example is the Software Dock [45], which is a distributed architecture that supports the full software deployment life cycle. Field docks provide an interface to the user's site. These docks connect to release docks at producer sites using a wide area event service. While the software dock can be used to deploy any kind of software system, and thus satisfies the genericity requirement, the description of each release in the Deployable Software Description (DSD) language presents significant overhead. Moreover, the Software Dock is particularly good at

deploying components from different, possibly distributed origins, which is outside the scope of this chapter. The same can be said of the Software Release Manager (SRM) [98].

Deployment tools that primarily address the user perspective fall in the category of software product updaters [53]. This category can be further subdivided into monolithic product updaters and component-based product updaters. Whereas product updaters in general do not make assumptions on the structure of the software product they are updating, component (or package) deployment tools are explicitly component-based.

JPloy [68] is a tool that gives users more control over which components are deployed. The question is, however, whether users are actually interested in how applications are composed. In that sense, JPloy may not be a good match for application deployment in the strict sense.

Package deployment tools can be further categorized as based on source packages or binary packages. A typical example of source-based package deployment tools is the FreeBSD ports system [79]. Such systems require users to download source archives that are subsequently built on the user's machine. Source tree composition [26] is another approach that works by composing component source distributions into a so-called *bundle*. The tool performing this task, called AutoBundle, constructs a composite build interface that allows users to transparently build the composition. Source-based deployment, however, is relatively time-consuming and thus fails to satisfy the efficiency requirement.

Binary package deployment tools do, however, satisfy the efficiency requirement. They include Debian's Advanced Package Tool (APT) [85], the Redhat Package Manager (RPM) [5], and more recently AutoPackage [2]. These tools download binary packages that are precompiled for the user's platform. Both APT and RPM are tied to specific Linux distributions (Debian/Ubuntu and Redhat/SuSe respectively) whereas autopackage can be used across distributions. Nevertheless AutoPackage only works under Linux. Although these deployment tools are independent of programming language, they are not generic with respect to the operating system.

The deployment system Nix [31] supports both source and binary deployment of packages in such a way that it is transparent to the user. If no binary package is found it falls back to source deployment. It features a store for non-destructively installing packages that are identified by unique hashes. This allows side-by-side installation of different versions of the same package. Nix is the only deployment tool that is completely safe because its non-destructive deployment model guarantees that existing dependencies are never broken because of an update. Furthermore, it is portable across different flavors of Unix and does not require root access (which is the case for all package deployment tools except AutoPackage).

One problem in general with package deployment tools is that they are invasive with respect to the environment of the user. For instance, the value of these tools is maximum when *all* software is managed by it. This explains why most such tools are so intertwined with operating system distributions, but it is a clear violation of the lightweightness requirement.

While some systems, such as Nix, AutoPackage and JPloy, can be used next to the 'native' deployment system, they still have to be able to manage all dependencies in addition to the component that the user actually wants to install. In the worst case

this means that a complete dependency tree of packages is duplicated, because the user deployed her application with a deployment tool different from the standard one. Note that this is actually unavoidable if the user has no root access. Note also that the user is at least required to install the deployment system itself, which in turn may not be an easy task.

7.2.3 Overview of the Approach

The motivations for component-based development are manyfold and well-known. Factoring the functionality of an application in separate components, creates opportunities for reuse,—both within a single product or across multiple products [87]. A distinguising feature of component-based development is the fact that components have their own life-cycle, both within a product and across products. This means that components are evolved, released, acquired and deployed independently, by different parties and at different moments in time.

In this chapter components are interpreted as groupings of files that can be versioned as a whole. Components, however, often are not stand-alone applications. This means that a component may require the presence of other components to function correctly. Such dependencies may be bound either at build-time or at runtime. Applications are then derived by binding these dependencies to implementation components, either at build-time, load-time or even runtime.

In the following I assume a very liberal notion of dependency, and consequently of composition. When one component requires another component it is left unspecified what the concrete relation between the two components amounts to. Abstract dependencies thus cover both build-time and runtime dependencies. Under this interpretation, composition is loosely defined as merging all files of all related components into a single directory or archive.

When a component has been built, some of the resulting object files will contribute to the composed application. This set of files is called the (component) distribution. To distribute an application to users, the relevant component distributions are composed before release, resulting in a single application distribution. Thus, an application is identified with a certain root node in the component dependency graph and its distribution consists of the transitive-reflexive closure of the dependencies below the root.

In the next section I will present a technique to efficiently create and deliver such application releases, called *binary change set composition*. We will see that continuous integration of component-based software extends naturally to a process of automatic continuous release. A component will only be built if it has changed or if one of its dependencies has changed. If a component has been built it is released automatically. The results of a build are stored persistently so that components higher up in the dependency graph may reuse previous builds from components lower in the dependency graph.

Apart from the files belonging to a single component, the composition of these sets of files is also stored. The space requirements for this can quickly become unwieldy, therefore these application distributions are stored differentially. Differential storage works by saving the changes between files. Instead of composing sets of files, one

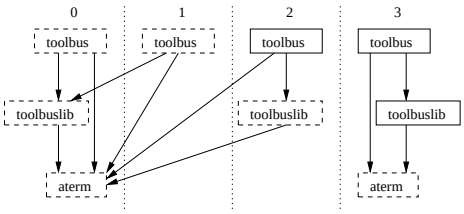


Figure 7.1: Incremental integration

can now compose sets of change sets. In addition to storing many releases efficiently, binary change set composition yields an efficient way of updating user installations.

7.3 Binary Change Set Composition

7.3.1 Incremental Integration

Tools like *make* optimize software builds because it only updates targets when they are out of date. It is possible to lift this paradigm from the level of files to the level of components. Hence, a component is only built if it is out of date with respect to some saved state, or when one of its dependencies is out of date. If built artifacts are stored persistently they can be reused. Sharing of builds is particularly valuable when a software product is continuously integrated [39]. Traditionally continuous integration is defined as a process where developers continuously integrate small changes to main development branch in the source control system. Then, after every change, the complete application is automatically built from scratch and automated tests are run. A naive apporach to building large systems from scratch, however, may not scale.

Consider an example that derives from three real-world components, *toolbus*, *toolbuslib* and *aterm*. The Toolbus is a middleware component that allows components ("tools") to communicate using a centralized software bus. Tools implemented in C use the *toolbuslib* component for this. Using the Toolbus, tools exchange data in a tree-like exchange format called Annotated Terms (ATerms) this datastructure is implemented by the *aterm* component. Obviously, *toolbus* requires both the connection and the exchange format libraries, whereas the connection library only requires the exchange format. All three components are used with the ASF+SDF Meta-Environment, a component-based application for language development [95].

Figure 7.1 shows four build iterations. The dashed boxes indicate changes in that particular component. In the first iteration every component has been built. At the time of the second iteration, however, only the top-level toolbus component has changed, so it is built again but this time reusing the previous builds of *toolbuslib* and *aterm*. Similarly, in the third iteration there has been a change in the *toolbuslib* component. Since *toolbus* depends on *toolbuslib* a new build is triggered for both *toolbuslib* and *toolbus*. Finally, in the last iteration changes have been committed to the *aterm* component and as a result all components are rebuilt.

An implementation of incremental continuous integration, called Sisyphus, has been described in [100]. This system works as follows. Every time a commit to the source control system occurs, Sisyphus checks out all components. It does this by starting with a root component, and reading a special file contained in the source tree that describes the dependencies of this component. This process is repeated for each of the dependencies. Meanwhile, if the current version of a component has not been built before, or one of its dependencies has been built in the current iteration, a build is triggered. Results are stored in a database that serves as saved state.

7.3.2 Build and Release Model

The build and release model presented in this section can be seen as the data model of a database for tracing change, build and release processes. Additional details can be found in [100]. The state of a component at a certain moment in time is identified with its version obtained from the source control system. Each version may have been built multiple times. The model records for every build of a component version which builds were used as dependencies. A set of built artifacts is associated to each build. Finally, a release is simply the labeling of a certain build; the set of releases is a subset of the set of builds.

In the context of this chapter two sets are important: *Build*, the set that represents component builds, and *Use* defined as a binary relation between builds (i.e. $Use \subseteq Build \times Build$). This dependency relation derives from explicitly specified requires interface *within* the source tree of each component. At build-time the required components are bound the source trees of those components, *at that moment in time*. Thus, the integration process takes the *latest* revision of each component. Building a component then results in a set of built artifacts (libraries, executables etc), given by the function $\text{files}(aBuild)$.

The extent of a build is defined as the set of builds that have participated in a build. It is computed by taking right image of a build b in the transitive-reflexive closure of the *Use* relation: $\text{extent}(b) = Use^*[b]$. The extent of a build thus contains all builds that will make up an application release. The set of files that will be part of a release is derived from the set of files that each component in the extent contributes. This is discussed in the next section.

7.3.3 Prefix Composition

When a component has been built some of the resulting object files will contribute to the composed application. The set of files that is distributed to the user is called the application distribution, and it is composed of component distributions.

Figure 7.2 shows how the files contributed by each component to the toolbus application are taken together to form a single application distribution. On the left is shown that all installable files of each component first end up in a component specific directory,—in the example this could have been the result of issuing *make install*. To release the *toolbus* as an application, these sets of files and directories are merged, resulting in a single application distribution, as shown on the right.

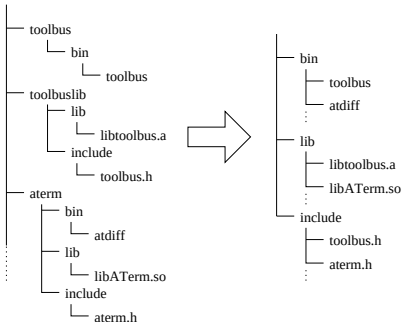


Figure 7.2: Prefix composition

I call this way of composing components "installation prefix composition" since the component directories on the left correspond to directory prefixes passed to *./configure* using the command line option *--prefix*. Such configuration scripts are generated by AutoConf [1], a tool to configure build processes that is widely used in open source projects. Among other things, it instructs *make install* to install files to a Unix directory hierarchy below the prefix. Prefix composition thus constitutes merging directories containing built artifacts.

Since components are composed by merging sets of files and directories we must ensure that no component overwrites files of another component. Formally, this reads:

$$\forall b \in \mathit{Builds} : \bigcap_{b' \in \text{extent}(b)} \text{files}(b') = \emptyset$$

In other words, this ensures that making a distribution is compositional. Instead of explicitly creating a global application distribution one can compose individual component distributions to achieve the same effect. What the property effectively states is that building a component, viewed as a function, distributes over composition.

There is one technicality which has to be taken care of: the distributed files should be relocatable. Because builds happen at the developer's site one must ensure that no (implicit) dependencies on the build environment are bound at build time. For instance, if a Unix executable is linked to a dynamic library that happens to be present at build time, then this library should also be present on the user's machine,—even on the same location. Since we do not want to require that users should reproduce the complete build environment, care must be taken to avoid such "imported" dependencies. I elaborate on this problem in Section 7.4.3.

7.3.4 Change Set Delivery

If the compositionality property holds the composition is defined by collecting all files that are in the extent of a build:

$$\text{files}^*(b) = \bigcup_{b' \in \text{extent}(b)} \text{files}(b')$$

Upgrade	Change set delivered to user
$0 \rightarrow 1$	$\{\Delta_1^0$bin/toolbus$\}$
$1 \rightarrow 2$	$\{\Delta_2^1$bin/toolbus, Δ_2^0lib/libtoolbus.a$\}$
$2 \rightarrow 3$	$\{-$bin/atdiff$\}$

Table 7.1: Change set delivery

The function files* computes the set of files that eventually has to be distributed to users. An update tool could transfer these files for every build that is released to the users of the application. If a user already has installed a certain release, the tool could just transfer the difference between the installed release and the new release. Let $F_{1,2} =$ files$^*(b_{1,2})$. Then, the change set between two releases b_1 and b_2 is defined as:

$$\{\Delta(F_1 \cap F_2), +(F_2 \backslash F_1), -(F_1 \backslash F_2)\}$$

Change sets have three parts. The first part, indicated by Δ contains binary patches to update files that are in both releases. The second and third part add and remove the files that are absent in the first or second release respectively.

If we turn our attention once again to Figure 7.2, we see on the right the composed prefix for the *toolbus* application. Let's assume that this is the initial release that a typical user has installed. In the meantime, development continues and the system goes through three more release cycles, as displayed in Figure 7.1. The sequence of change sets transferred to our user, assuming she upgrades to every release, is listed in Table 7.1.

The second iteration only contains changes to the *toolbus* component itself. Since the only installable file in this component is *bin/toolbus*, a patch is sent over updating this file at the user's site. In the next iteration there is a change in *toolbuslib* and as a consequence *toolbus* has been rebuilt. Updating to this release involves transferring patches for both *bin/toolbus* and *lib/libtoolbus.a*. There must have been a change in the *bin/toolbus* since *libtoolbus.a* is statically linked. In the final iteration the changes were in the *aterm* component. However, this time neither *toolbuslib* nor *toolbus* are affected by it—even though they have been rebuilt—because the change involved the removal of a target: the *bin/atdiff* program appears to be no longer needed. Neither *toolbus*, nor *toolbuslib* referenced this executable, hence there was no change in any of the built files with respect to the previous release. As a result, the change set only contains the delete action for *bin/atdiff*. Note that these change sets can be easily reverted in order to support downgrades.

7.3.5 Change Set Composition

Until now we have assumed that every application release was completely available and the change sets were only used to optimize the update process. From the use of change sets to update user installations, naturally follows the use of change sets for storing releases. Figure 7.3 shows how this can be accomplished.

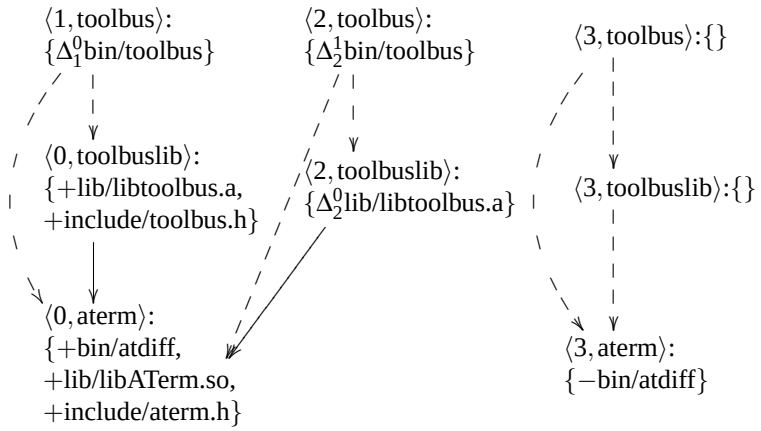


Figure 7.3: Change set composition

Once again, the three integration iterations are shown. In the first iteration, only the *toolbus* had changed and had to be rebuilt. This resulted in an updated file *bin/toolbus*. The figure shows that we only have to store the difference between the updated file and the file of the previous iteration. Note that initial builds of *aterm* and *toolbuslib* (from iteration 0) are stored as change sets that just add files.

The second iteration involves a change in *toolbuslib*; again, patches for *toolbus* and *toolbuslib* are stored. However, in the third iteration, the change in the *aterm* component did not affect any files in *toolbus* or *toolbuslib*, so no change sets need to be stored for these components. But if users should be able to update their installation of the toolbus application, still the toolbus should be released. So there really are four toolbus releases in total, but the last one only contains changes originating from *aterm*.

I will now describe how this scheme of binary change set composition can be implemented on top of Subversion.

7.4 Implementation using Subversion

7.4.1 Composition by Shallow Copying

Subversion [19] is a source control system that is gaining popularity over the widely used Concurrent Version System (CVS). Subversion adds many features that were missing in CVS, such as versioning of directories and a unified approach to branching and tagging. Precisely these features prove to be crucial in the implementation of binary change set composition on top of Subversion.

Next, I will describe how Subversion repositories can be used as release repositories that allow the incremental delivery of updates to users. The release process consists of committing the component distributions to a Subversion repository, and then use branching to identify component releases. Such component-release branches are the

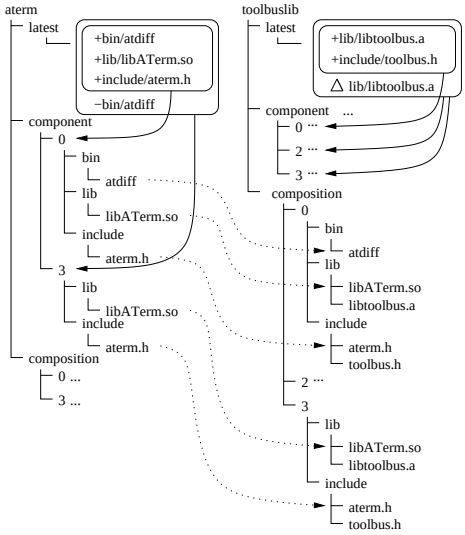


Figure 7.4: Composition by shallow copying

unit of composition, which is also implemented by branching.

The crucial feature of Subversion that makes this work efficiently, is that branching is implemented by shallow copying. So, for instance a branch is created for some repository location—file or directory—by copying the tree to another location. At the new location, Subversion records a *reference* to the source of the copy operation. The copy operation is a constant-space operation and therefore a very efficient way to implement sharing.

Figure 7.4 shows a snapshot of a Subversion repository containing *aterm* and *toolbuslib* releases based on the change set graph displayed in Figure 7.3. For the sake of presentation releases of the *toolbus* have been omitted. On the left we see the Subversion tree for *aterm*, and on the left the tree for *toolbuslib*. The trees have subtrees indicated *latest*, *component* and *composition*. The *latest* tree is where component distributions are stored. The rounded boxes contain the change sets from Figure 7.3. The *component* tree and the *composition* tree contain shallow copies of versions of the latest tree; these are the releases proper. Solid arrows indicate copy relations the context of a single component,—dotted arrows indicate cross component copying (i.e. composition relations).

After every build the changes in the distributions are commited to the *latest* tree. The state of the *latest* tree at that time is then copied to a branch identifying this particular build; such branches are created by copying the files from latest to a separate directory under *component*. Note that since the change set for *toolbuslib* in iteration 3 was empty, *toolbuslib* release 3 is created from the state of the latest tree at iteration 2.

The tree below *composition* contains releases for compositions. This works by, instead of just copying the files belonging to a single build, copying the files in the

extent of the build. In the example, this means that, next to the files contained in *toolbuslib* releases also the files in *aterm* releases are copied. If we compare *toolbuslib* composition 0 and 3, one can see in the figure that composition 0 is composed with release 0 of *aterm*, whereas composition 3 is composed with release 3 of *aterm*, exactly as in Figure 7.3.

7.4.2 Upgrade is Workspace Switch

Assuming the proper access rights are in place, the Subversion repository can be made publicly accessible for users. A user can now *check out* the desired subtree of *compositions*; this can easily be performed by a bootstrap script if it is the initial installation. She then obtains the composed prefix of the application.

Now that the user has installed the application by checking out a repository location, it is equally easy to down- or upgrade to a different version. Since the subtrees of the *composition* tree contain all subsequent releases of the application, and the user has checked out one of them, up- and downgrading is achieved by updating the user's local copy of the composed prefix to another release branch. Subversion provides the command *svn switch* for this. Subversion will take care of adding, removing or patching where necessary.

Note that the sharing achieved in the repository also has an effect on how local checkouts are updated. For instance, recall that the third release of *toolbus* in the example involved the removal of *bin/atdiff*. If we assume that the user has installed the second release, and decides to upgrade, the only action that takes place at the user site is the removal of *bin/atdiff*, since the third release of both *toolbus* and *toolbuslib* contain the same change sets as second release of both these components.

7.4.3 Techniques for Relocatability

Installed application releases are ready to use with the exception of one technicality that was mentioned before, which is: relocation. Since the released files may contain references to locations on the build server at the side of development, these references become stale as soon as the users installed them. We therefore require that applications distributed this way should be binary relocatable. There are a number of ways to ensure that distributions are relocatable. Some of these are briefly discussed below.

There are ways to discover dynamically what the locations are of libraries and/or executables that are required at runtime. For instance, AutoPackage [2] provides a (Linux-only) library that can be queried at runtime to obtain 'your' location at runtime. Since the files contributed by each component are composed into a single directory hierarchy, dependencies can be found relative to the obtained location.

Another approach is to use wrapper scripts. As part of the deployment of an application a script could be generated that invokes the deployed application. This script would then set appropriate environment variables (e.g. PATH or LD_LIBRARY_PATH on Unix) or pass the location of the composed prefix on the commandline.

Finally, we could use string rewriting to effectively relocate unrelocatable files just after deployment. This amounts to replacing build time paths with their runtime counter-parts in every file. Special care must be taken in the case of binary files, since

it is very easy to destroy their integrity. This technique, however, has been applied successfully.

7.5 Evaluation

7.5.1 Experimental Validation

A prototype implementation has been developed as part of the Sisyphus integration framework [103]. It has been used to deliver updates for a semi-large component-based system, consisting of around 30 components: the ASF+SDF Meta-Environment [92]. All built artifacts were put under Subversion, as described in the previous section. As expected, the repository did not grow exponentially, although all 40 component compositions were stored multiple times.

The ASF+SDF Meta-Environment is released and delivered using source tree composition [26]. This entails that every component has an abstract build interface based on AutConf. The prefixes passed using *--prefix* during build are known at the time of deployment so could be substituted quite safely. In order to keep binary files consistent, the prefixes passed to the build interface were supplanted with superfluous '/' characters to ensure enough space for the subtituted (user) path. This trick has not posed any problem as of yet, probably because package-based development requires that every dependency is always passed explicitly to the AutoConf generated *./configure* script.

A small Ruby script served as update tool. It queries the repository, listing all available releases. If you select one, the tree is checked out to a certain directory. After relocation the Meta-Environment is ready to use. Before any upgrade or downgrade however, the tool undoes the relocation to prevent Subversion from seeing them as "local modifications".

7.5.2 Release Management Requirements

The subject of lightweight application upgrade belongs to the field of software release management. In [98], the authors list a number of requirements for effective release management in the context of component-based software. I discuss each of them briefly here and show that our approach satisfies them appropriately.

Dependencies should be explicit and easily recorded Incremental continuous integration of components presumes that dependencies are declared as meta data within the source tree of the component. Thus, this requirement is satisfied.

Releases should be kept consistent This requirement entails that releases are immutable. The incremental continuous integration approach discussed in this chapter guarantees this.

The scope of the release should be controllable Scope determines who is allowed to obtain a software release. The release repository presented in this chapter enables the use of any access control mechanism that is provided by Subversion.

A history of retrievals should be kept Although I do not address this requirement directly, if the Subversion release repository is served over HTTP using Apache, it is easily implemented by consulting Apache's access logs.

With respect to release management the implementation of change set composition using Subversion has one apparent weakness. Since Subversion does not allow cross-repository branching it would be hard to compose application releases using third-party components. However, this can be circumvented by using the Subversion dump utility that exports sections of a repository on file. Such a file can then be transferred to a different repository.

7.5.3 Update Management Requirements

In Section 7.1 I listed the requirements for application upgrade from the user perspective. Let's discuss each of them in turn to evaluate whether application upgrade using Subversion satsifies them.

Lightweightness No invasive software deployment tool has to be installed to receive updates: only a Subversion client is required. Since, many language bindings exist for Subversion, self-updating functionality can be easily integrated within the application itself.

Genericity Change set composition works with files of any kind; there is no programming language dependency. Moreover, Subversion is portable across many platforms, thereby imposing no constraints on the development or user environment.

Safety The Subversion *switch* command is used for both upgrade and downgrade. A failed upgrade can thus be quickly rolled back. Another contribution to safety is the fact that Subversion repository modifications are atomic, meaning that the application user is shielded from inconsistent intermediate states, and that releases put out in parallel do not interfere.

Efficiency Efficiency is achieved on two accounts. First the use of Subversion as delivery protocol ensures that an upgrade involves the transfer of just the differences between the old version and the new version. Secondly, while the unit of delivery is a full application, only the files per component are effectively stored, and even these are stored differentially.

Although all requirements are fulfilled satisfactory, the primary weakness of binary change set composition remains the fact that distributed files have to be relocatable. Solving this problem is left as future work.

7.6 Conclusion and Future Work

In this chapter I have discussed the requirements that have to be fulfilled so that application upgrade is a burden neither for the development side, nor for the user side. Related work in the area of software release management did not live up to these requirements.

The binary change set composition technique does live up to these requirements, and can be used to deliver new application releases accurately, frequently and quickly. The implementation on top of Subversion shows that the approach is feasible and may serve as a low impact adoption path.

However, ample opportunities for future work remain. First of all, the relocatability requirement of distributed files should be investigated. For instance, so-called application bundles on Mac OS X are always relocatable and would be perfect candidates for being updated using the techniques of this chapter. Further research will have to point out if the notion of relocatable application bundles can be ported to other platforms. On the other had, I would like to investigate whether it is possible to make the binding of dependencies a first-class citizen in the model. For instance, one could envision a kind of service where components register themselves in order for them to be found by other components. This subject is closely related to the notion of dependency injection [38].

Another direction of future work concerns the integration of deployment functionality with the released application itself. Nowadays, many applications contain functionality to check for new updates. If they are available they are installed and the application is restarted. It would be interesting if using the approach of this chapter one could design such “update buttons” in a reusable and generic way. Similarly, it should be investigated how such self-updating applications could be enhanced with functionality for reporting bugs or other kinds of feedback.

Chapter 8

The Sisyphus Continuous Integration and Release System

Abstract Continuous integration and release requires special attention to the requirements of platform independence, configurability and traceability. This chapter describes how the implementation Sisyphus lives up to these requirements in the context of component-based development. We describe the high-level architecture of Sisyphus, how Sisyphus is configured and how traceability is maintained. Finally, we provide some details about the user interface of Sisyphus and about third-party software that has been used to keep the size of Sisyphus minimal.

8.1 Introduction

The Sisyphus continuous integration and release system can be used to automatically build and release component-based systems. As such it helps extending the practice of continuous integration [39] into a practice of continuous release. Sisyphus will monitor a set of source repositories and after every change to any of the sources it will build and release the complete system.

Whereas implementing a continuous integration system proper may not seem like a very big deal, the addition of automatic release facilities adds to the complexity of this task. Not only should the software product be built after every change, it should also be released. Apart from creating installable packages for each build and making them available to users, releasing entails that accurate links to the sources must be maintained. This calls for explicit management of meta-data describing exactly what went into a build and hence what is contained in a release package.

In this chapter we describe how such release facilities and the associated meta-data management has been implemented in Sisyphus. We highlight some of its distinguishing features and discuss design decisions that have lead to the current state of the system. This chapter is organized as follows. First, in Section 8.2 we will identify the technical requirements for a continuous release system, specifically in the context of

heterogeneous component-based systems. Second, in Section 8.3, we give an architectural overview of the system. This section will primarily focus on the distributed nature of the implementation of Sisyphus. One important feature of any build system is how it can be configured for use in a specific software development environment. This is elaborated upon in Section 8.4. Then, since Sisyphus is not only a build system but a release system too, it must ensure traceability of the released artifacts. For this Sisyphus maintains a database recording accurate meta data describing the contents of each release. The data model of the database is described in Section 8.5. Finally, we discuss details of the implementation Section 8.6 and present conclusions and future work in Section 8.8.

8.2 Functional Requirements

The functional requirements for a continuous integration and release system for component-based, heterogeneous software are discussed in this section. They can be summarized as follows:

- Component-based: components and dependencies are first-class. Every component is released together with all its transitive dependencies.
- Platform independent: the integration and release system should run on as many platforms as possible. As a consequence, no proprietary tools, libraries or protocols should be used in its implementation.
- Configurable: in order to be as widely applicable as possible, the system should be configurable. More specifically, this means that the system should be independent of the programming language(s) and build tools used in a project.
- Traceable: as mentioned above, it should at all times be possible to trace delivered releases back to the sources used to build the product.

Below we elaborate on each of the requirements and indicate how the implementation of Sisyphus realizes them.

8.2.1 Component-Based

One of the primary requirements for Sisyphus has been that it allows the integration and release of component-based systems. This means that components are represented by independently evolving sets of source files (called source trees), often residing in separate entries in the version control system (VCS) or even in different VCSs.

Components have explicit sets of dependencies. In the context of Sisyphus these are specified in a dedicated file within the source tree of the component based on the open source utility `pkgconfig` [48]. A small excerpt is shown below:

```
Name: pandora
Version: 1.4
Requires: asc-support,aterm,pt-support,toolbuslib
Maintainers=developer@cwi.nl
```

This file, which resides in the top-level source directory of the component, identifies this source tree as belonging to the `pandora` component on the first line. The second line declares that if this component is formally released it will receive 1.4 as informative version number. This number is not used for identification but serves to indicate a level of maturity to the actual users of this component. The following line lists the components it requires. Note the absence of version numbers here: which versions are used for these dependencies follows from the version control system at the time of integration. The presence of this file is one of the few assumptions Sisyphus makes.

Integration and release of a software product in this setting means integrating and releasing each component individually as well as together with its dependencies. More specifically, a component should be built against its dependencies to test whether a component can be compiled using the interfaces of the dependencies, and it should be released *including* its dependencies to obtain a complete, working software system; i.e. releasing a software system consists of publishing the transitive closure the dependencies of a software component.

8.2.2 Platform Independent

The requirement of platform independence entails that no technology specific to a certain platform should be used in the implementation of the continuous integration and release system. This requirement affects both the back-end and front-end sides of Sisyphus. The back-end lives at the vendor site; it consists of tooling to continuously build and release software. When portability of the product is a concern, it can be essential that the integration and release is done on multiple platforms at the same time. This entails that the back end should be portable across these platforms as well. The Sisyphus back-end is implemented in the programming language Ruby[1] which is portable across many platforms.

Similarly, the front-end should be accessible from any platform the product is released for. Hence its implementation should impose no constraints on the platform of eligible users. In the context of Sisyphus we have chosen for a dynamic web application as a front-end which can be accessed from anywhere on any platform. The front-end is discussed in more detail in Section 8.6.

8.2.3 Configurable

Configurability means that the system can be tailored for application in different settings. More specifically it entails that the system is parametrized in aspects that may differ across vendors or products. For instance, the system should not be tied to one specific programming language since different vendors may make different choices in this area. Moreover, in component-based settings, different components may have been implemented in different languages. This leads to heterogeneous, component-based software. If such software should be built and released automatically, clearly a continuous integration and release system must not make any assumptions on the

[1] See `http://www.ruby-lang.org`.

programming languages that are used. The same holds for other elements of the development environment, such as the kinds of build tools (e.g. Make [37]) that are used.

Sisyphus accommodates this requirement by being parametrized in what constitutes a build. What kind of commands are executed in order to build a component is fully configurable. The configuration interface of Sisyphus furthermore allows the configuration of platform dependent dependent variables separate from the variables that applies to all platforms. The configuration of Sisyphus is discussed in more detail in Section 8.4.

8.2.4 Traceable

The final and most essential requirement for a continuous release system is that release packages (as well as user installations) can be traced back to the sources that were used to construct a particular version of the product. In other words, releases produced by system must be accompanied by accurate *bills of materials* (BOMs) [70]. BOMs are common in manufacturing and are equally important in software engineering. A BOM exactly describes what parts or components went into a certain product (release). In the context of component-based software release this boils down to explicitly managing which versions of which components were used to construct a particular release. Additionally it requires an identification of platform (consisting of, e.g., operating system, hardware class etc.) and the tools used to create the release (e.g., compilers, build tools etc.).

In theory, one would like every software artifact that contributes to the end-product to be included in the BOM. We note that the Nix deployment system [31] actually achieves this level of preciseness in identifying software releases. However, this comes at the cost of having to explicitly manage (build and release) *every* software artifact that may contribute to the end-result, including compilers and additional tools. In Sisyphus we have taken an approach that is much more light-weight so it is less intrusive to the development, build and test environment. Tracing the configuration of Sisyphus is discussed in Section 8.4. The general data model underlying the release database is described in Section 8.5.

8.3 Architectural Overview

8.3.1 Introduction

The Sisyphus system consists of a back-end and a front-end. Figure 8.1 shows a high-level overview of the system. The back-end handles the continuous integration and release of software products developed in a component-based fashion. Input to the back-end are the sources of the product and a description of the development environment (e.g., compilers, build tools, etc.). Both the sources and the definition of the environment can be changed by the developers on the vendor side. If a change is made, the back-end integrates the changes and makes a release available through the front-end. Both customers and testers are then able to obtain new releases via the front-end

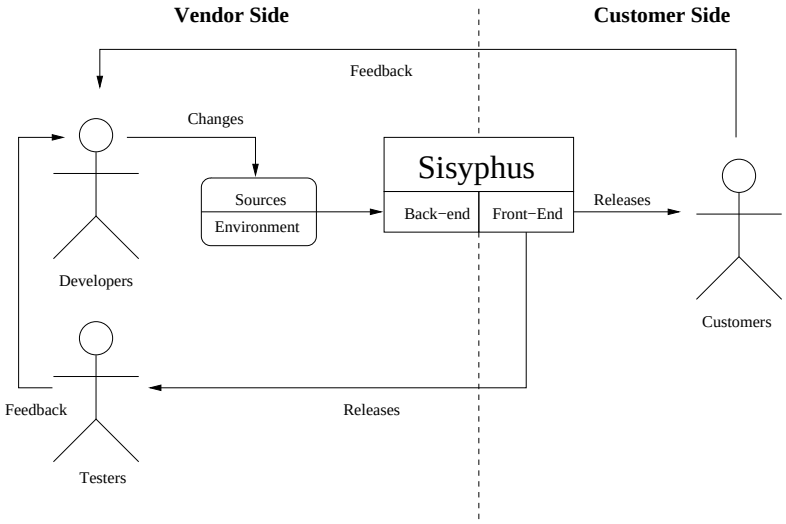


Figure 8.1: Overview of the Sisyphus system

and provide feedback to the developers. Below we will discuss in more detail how the back-end and front-end are deployed.

8.3.2 Deployment Architecture

To introduce the Sisyphus system in more detail we present the deployment architecture in Figure 8.2 using a UML-like deployment diagram. The nodes represent independently running subsystems ("servers", or "active objects"). The arrows between the nodes can be read as "has access to". The nodes that have been encircled are the proper parts of the Sisyphus implementation. The lower half of each node shows how the component has been implemented. For instance, the Sisyphus builder (on the left) is implemented in the programming language Ruby. The front end (the encircled node at the right), consisting of a web application is constructed in Ruby as well and uses the Web framework *Ruby on Rails*[2]. An Apache web server servers as a proxy to the *Ruby on Rails* web application.

The left hand side of the figure shows the "builder" side. During operation there may be multiple Sisyphus builders (indicated by an asterisk). Each builder runs independently on individual hosts in a network of computers. This allows the software product to be built on different platforms. Each builder monitors one or more source repositories (also indicated by the asterisk in the upper right corner).

The middle of the figure represents the stateful parts of Sisyphus. It consists of a database, a configuration repository and a file server. The database contains the bills of materials for each build. The data model of this database is discussed in more detail

[2] See `http://www.rubyonrails.com`.

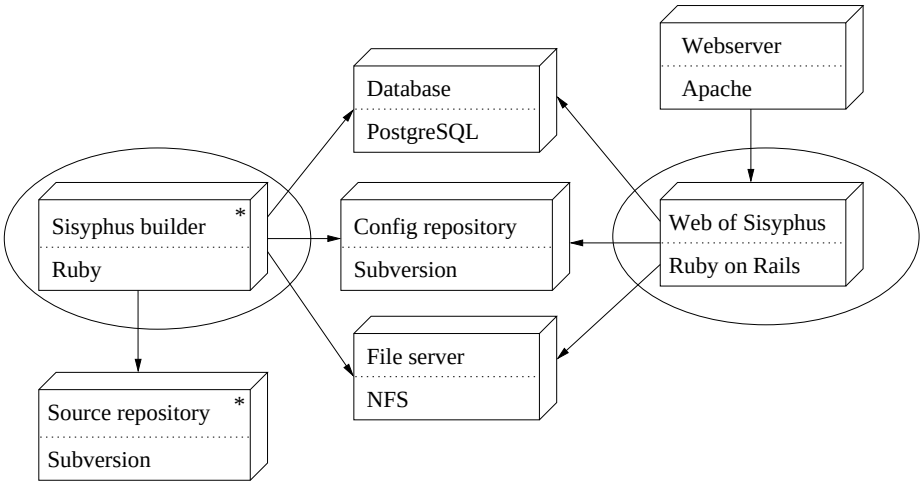


Figure 8.2: Deployment architecture of Sisyphus

below, in Section 8.5. Each builder has access to this database (through TCP/IP) to check whether a certain component has to be built or not. If so, the builders executes the build and stores the results in the database. These results can then be retrieved from the database via the front-end. Build and release artifacts (e.g., binaries, release packages etc.) are uploaded to the file server component. Every bill of materials in the database corresponds to a location on the file server. This allows the front-end to link every release to the files belonging to it.

Currently this file server is based on NFS. However, we prototyped the use of Subversion for this part of the architecture as well. This is covered in Chapter 7. Currently however, the implementation of the binary change set technique is in pre-alpha stage and thus is not part of the production version of Sisyphus.

The configuration repository serves a special purpose. Sisyphus builders access this repository to find out in which repository components reside and to find out what commands have to be executed in order to actually build a software component. Additionally, the configuration repository may contain host-specific information about, for instance, the location of certain compilers or the presumed value of certain environment variables. In other words, the configuration repository contains the knowledge that Sisyphus builders require for finding, building and releasing a software component.

The configuration of the Sisyphus builders is fully versioned using a Subversion repository. This is an essential feature, since the configuration parameters that were used during a component build are part of the bill of materials stored in the database. This makes changes in the configuration to be accurately traceable. Without this two builds executed using different compilers would not be distinguishable in the database. As a side effect, changes in the configuration will trigger a new build. Even if there are no changes in the sources, it is still possible that a component will be built because, for

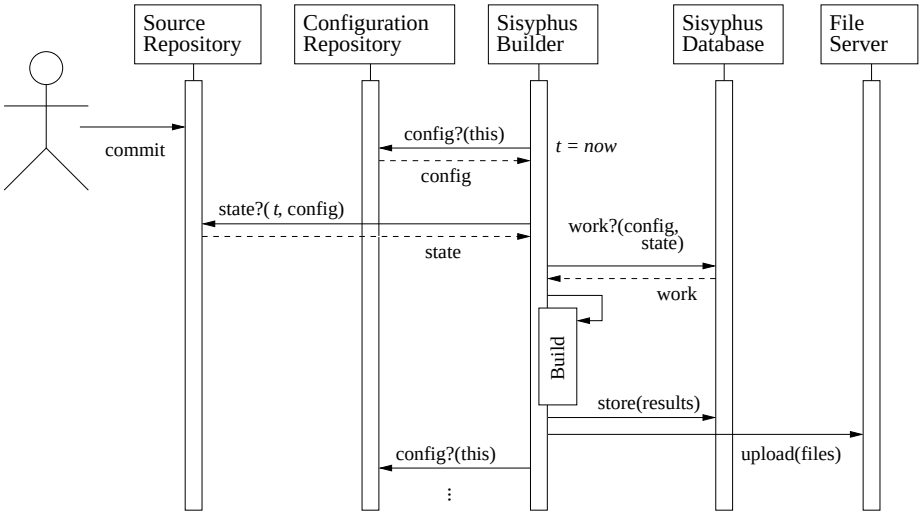


Figure 8.3: The basic Sisyphus scenario

instance, the changes in the configuration require the use of a new compiler. The bills of materials in the database record links to the exact version numbers of the configuration files that were used during a build.

In addition, the Subversion repository provides developers with a transparent way of accessing and updating this configuration information. Configuration files can be checked out, modified, and checked in again. In the next build cycle, each Sisyphus builder will use the new configuration information. The configuration model is discussed in more detail in Section 8.4.

The database populated by independently running Sisyphus builders is accessed from the outside world through a web interface. This front-end web application, called "Web of Sisyphus", presents a read only view of the database. Both the builders and the web front-end ensure that the database only grows monotonically. In other words: nothing is *ever* deleted from the database. As a side-effect, this ensures that releases are immutable. Developers may investigate the causes of build failures by inspecting log files. Users may download release packages associated to successful component builds.

8.3.3 Dynamics

We will now consider the basic scenario implemented by Sisyphus. Figure 8.3 shows a message-sequence chart to illustrate the dynamics of Sisyphus.

Sisyphus builders run in cycles. In other words, at a fixed interval in time a builder initiates a new integration cycle. The figure shows one such cycle. The initiative is with the Sisyphus builder which starts the cycle by requesting its configuration data from the configuration repository. This moment in time is fixed at t. The next step involves

requesting the state of the sources at time *t* with respect to the configuration obtained earlier. The configuration data include which components this builder should integrate and where they are to be found. The state returned by querying the source repositories (only one is shown in the diagram) is thus parametrized in the configuration applicable to *this* builder.

Then the database is queried to find out what has to be done with respect to the state of the sources. The builder subsequently obtains a work assignment. This work assignment may be empty if there have been no changes to the sources and/or configuration files in between the time of the previous cycle and *t*. Any work done by a builder is always relative to what has been stored in the database as being completed. The database also stores what has been done by which builder, under which configuration. If the configuration data for *this* builder has changed since the last cycle, this means that components may have to be built, even though there are no changes to the sources.

The computation of a work assignment with respect to the database and the configuration captures the core algorithms of backtracking and incrementality, as discussed in Chapters 4 and 5. The notion of work assignment captures *what* has to be built and released (which components), and *how* (in which order, with which dependencies).

If there is any work to be done, the builder performs the necessary build and release steps. Finally the results are stored in the database (e.g., success/failure of the build, log files etc.) and build and release artifacts (if any) are uploaded to the file server. If a component has build-time dependencies then the build artifacts of those dependencies are retrieved from the file server in order to be passed into the build. If there is no more work left to do, the builder then goes to sleep until it is time for the next cycle.

8.4 Build Configuration

As mentioned in the architectural overview, Sisyphus's configuration is accessed through a Subversion repository. What does this mean? Simply said, it means that dedicated configuration files are stored in a Subversion repository and the versions of this files form the identification used in the bill of materials that accompanies releases. The database stores these version numbers as part of the identity of builds and releases. In this section we describe the configuration model obeyed by the files in the configuration repository.

The configuration of Sisyphus is divided in two parts: global configuration and per host configuration. The global configuration parameters applies to all Sisyphus builders, whereas the per builder configuration contains customizations and overrides specific to a single builders. We will discuss each part in turn.

8.4.1 Global Configuration

The global configuration of Sisyphus consists of the definition of build actions (called the "script") and locations of components. The script is a list of build actions identified by name. The locations of components are specified by mapping repository locations to component names.

Currently Sisyphus only supports Subversion repositories, but different types can be easily added. The configuration file for specifying source locations already caters for this. This file, `sources.yml` is specified in YAML[3] format, which is easy to deal with for both humans and computers. The following listing is a small example:

```
-
 type: subversion
 protocol: svn+ssh
 location: svn.cwi.nl
 components:
  - aterm
  - toolbus
  - toolbuslib
```

The dashes are list constructors and the colons indicate mappings. The source configuration thus consists of list of repositories, each repository is a map containing configuration parameters one of which consists of the list of components residing in the repository. The components listed in this files are the ones that Sisyphus builders know about. For instance, if a component, say toolbuslib, requires the component aterm as a dependency, the builder performs a reverse look-up in in this mapping in order to find the location of aterm.

The script is also specified in YAML and consists of a mapping of build actions to shell script templates. An example build action could for instance be “configure” or “test”. The configure action could capture the invocation of configure script generated by AutoConf [1]; a tool commonly used in open source projects to instantiate Makefiles. The “test” action then could be configured to invoke “make check” in the top-level directory of the checkout of a certain component to compile and test the component.

Build actions are mapped to shell script templates. These templates may contain embedded Ruby code[4] to access certain internal values provided by the builder. For instance, the “configure” build action in the setting that Sisyphus is used, could have been specified as follows:

```
configure: ./configure --prefix=<%=install_dir%> \
             <%deps.each do |d|%>
              --with-<%=d.name%>=<%=install_dir%> \
             <%end%>
```

The text between `<%` and `%>` is Ruby code that will be executed to instantiate this template. This template accesses two variables of the builder: *install_dir* and *deps*. The first variable contains the directory where compiled binaries should be put; this variable is configured per builder (see below). The second variable, deps, contains the dependencies for the current component. The template uses this variable to construct command line flags of the form `--with-`*d*`=`*install_dir* for each dependency *d*. These flags allow the configure script to instantiate Makefiles in such a way that libraries and header files are found by the C compiler. In this case the files of all dependencies

[3] See `http://www.yaml.org`.

[4] Using the template language Embedded Ruby (ERB); contained in Ruby’s standard library.

are to be found below the directory *install_dir*. These action templates are instantiated and subsequently executed by the builder. The result of the execution of the complete sequence of actions constitutes a component build.

8.4.2 Per Builder Configuration

In addition to the global configuration of all builders, some configuration parameters are specific to a builder; this is done using *profiles*. This may have to do with certain paths that have to be different on different hosts or other platform differences. Simple platform dependencies can be ironed out using the *environment* entry in the profile. The environment consists of a shell script that is prepended before *every* build action before the action is executed. This way, for instance, the search path (PATH) for executables can be adapted just before a build action gets executed. Although the build actions are the same across platforms, the tools that will be used to execute the build action obviously are not.

Two other important builder specific configuration parameters concern the location of the working directory (the *build_dir*) and the location where to put built artifacts (the *install_dir*). These are both configured through the profile. Profiles are again YAML files in the configuration repository identified using a name. This name is passed on the commandline to a builder so that it knows the profile it should use.

Finally, profiles contain a fake variable, called the *world_version*. This number can be incremented to trigger a rebuild of everything for a certain builder. Sometimes the build environment changes in a way that is not visible in either the sources or the profile configuration parameters. Using the world version one can prevent a Sisyphus builder from assuming that "nothing has changed" whereas the developers might know better.

8.5 Data Model

In earlier chapters (notably Chapter 4, 5 and 7) we presented formal versions of the data model underlying the Sisyphus approach to continuous release. In this section we take a more implementation oriented view point. The data model as it is currently implemented is shown in Figure 8.4. The nodes in this figure represent classes that are mapped to tables in the database. Similarly, the arrows indicated both associations and foreign key constraints; they can be read as "has many" (due to the multiplicity indication "*"). The self-referential relation "use (a dependency relation) on Item is implemented using an intermediate table which is not shown.

Both the builder and the front-end use the object-relational mapping (ORM) that is provided by the *Ruby on Rails* component *ActiveRecord*. The diagram in Figure 8.4 is automatically derived from this mapping. Below we will speak of classes, but keep in mind that these are mapped to tables in the Sisyphus database using the ORM. The database itself is specified using an SQL schema.

The central class in the data model is called Item: it captures component integrations and releases. Every component build that has been attempted is represented by an object in this class. The fields progress, success, and released indicate the status of the build. If progress is true, the build is currently executing. If success is true, this

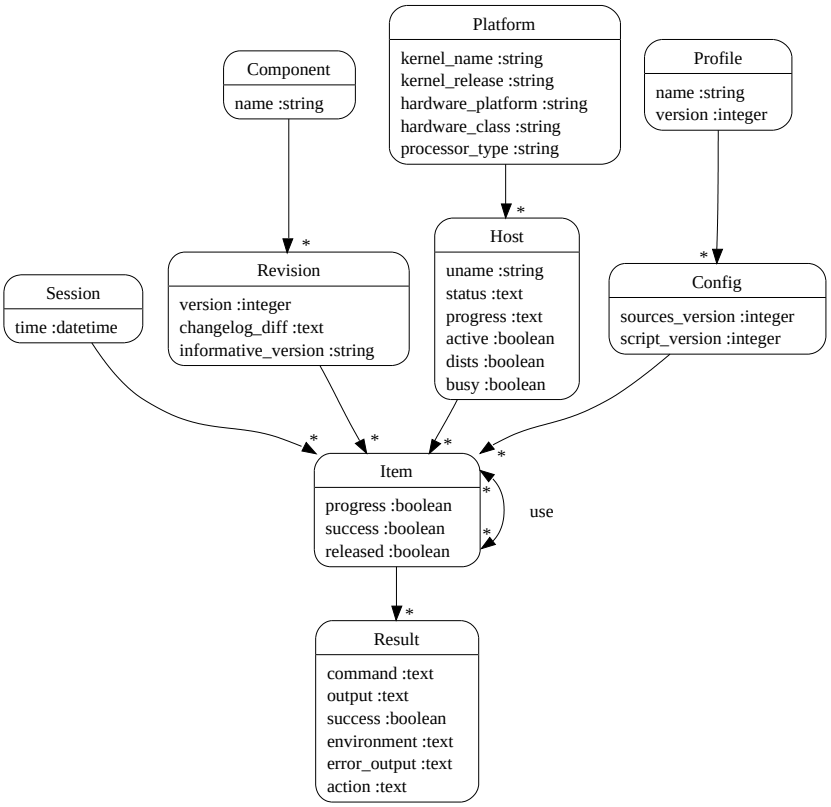


Figure 8.4: Data model of the current implementation of Sisyphus

build has completed and is a viable release candidate. If released is true, this build is formally released using the informative_version of the associated revision. Through the use relation the set of used dependencies can be retrieved. This allows the construction of *closures* for a certain build: a collection of the build artifacts of the component in question including the artifacts of all transitively reachable dependencies.

Apart from Session (which captures build cycles), all other classes (Revision, Component, Platform, Host, Config and Profile) contribute to the identity of an Item. In other words, they are (indirectly) part of Item's primary key. The Revision class captures the state of a component (which is identified just by name in class Component) at a particular moment in time: it has a revision number that derives from the version control system. Additionally it contains ChangeLog excerpt to indicate what has changed since the previous version. The informative_version field has the version number declared in the `pkgconfig` file. For instance, for the `pandora` component described above, this would have been 1.4.

Builds executed on different hosts and this may imply that two builds have been executed on different platforms. Builds different platforms should be distinguished. Therefore, the Host is part of the primary key of Item and Host has a foreign key on Platform. The Platform class captures differences in hardware and OS. Note that currently two builds executed on different hosts are distinguished while they may have exactly the same platform. In this sense, build identification is slightly more conservative than needed.

The Config class captures the source, script and profile configuration (and thus builder identity) discussed in the previous section. The version fields correspond to versions of the corresponding files in the configuration repository.

8.6 Details About the Implementation

In the previous sections we described the general architecture of Sisyphus and described underlying configuration and data model of the current implementation used by both the back-end and the front-end. The core algorithms used by the back-end (the builders) are discussed in detail in Chapters 4 and 5. In this section we focus on some technical details of the implementation. We will briefly discuss the front-end, discuss the size of the system and provide some pointers to additional resources.

8.6.1 Front-End: Web Of Sisyphus

The front-end of Sisyphus consists of a web application implemented using the web development framework *Ruby on Rails* (RoR). RoR provides default scaffolding for Model-View-Controller (MVC) paradigm web applications. This means that it provides Model classes (the Object Relational Mapping subsystem ActiveRecord), Conroller classes (ActiveController), and a way to construct views. For the latter we used the default technique, which consists of embedding Ruby code in HTML files (Embedded Ruby, ERB). ActiveRecord was used to map the formal models of the earlier chapters to database tables; after a few iterations this resulted in the data model of Figure 8.4. In RoR the URLs a user accesses are mapped to methods of controller

Figure 8.5: Main page of the *Web of Sisyphus*. The right three columns in the middle section represent the hosts that have running Sisyphus builders. Each row corresponds to a component and lists the outcome of the most recent build. The side-bar on the right shows the most recently performed builds globally; this information is also provided as an RSS feed

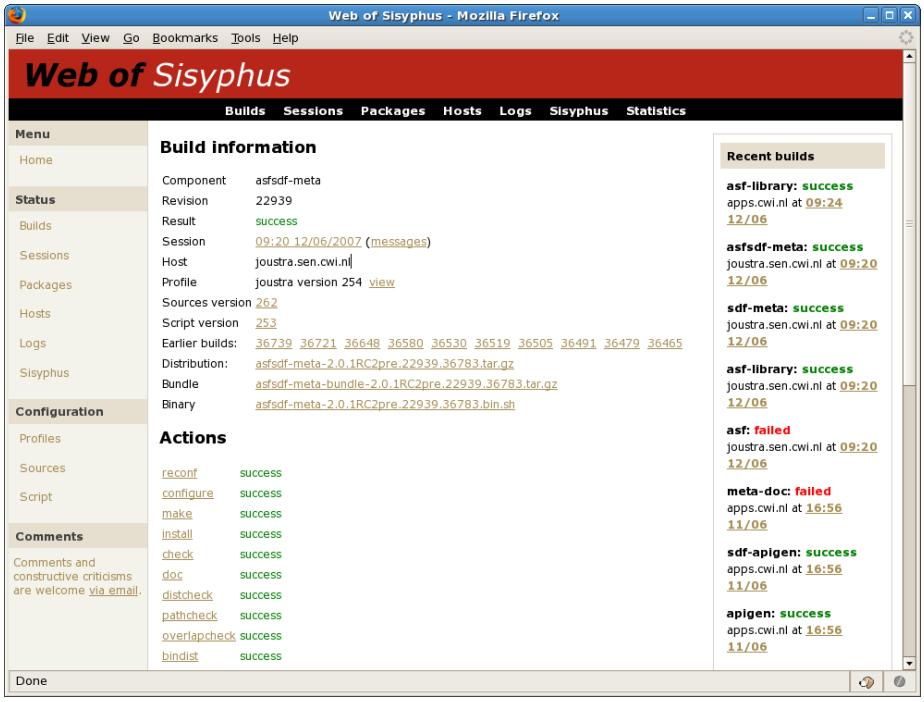


Figure 8.6: Build and release page of the `asfsdf-meta` component. All actions actions (ten in total) have succeeded, therefore this build represents a valid release candidate. The bottom three links of the “Build Information” section can be used to obtain the release

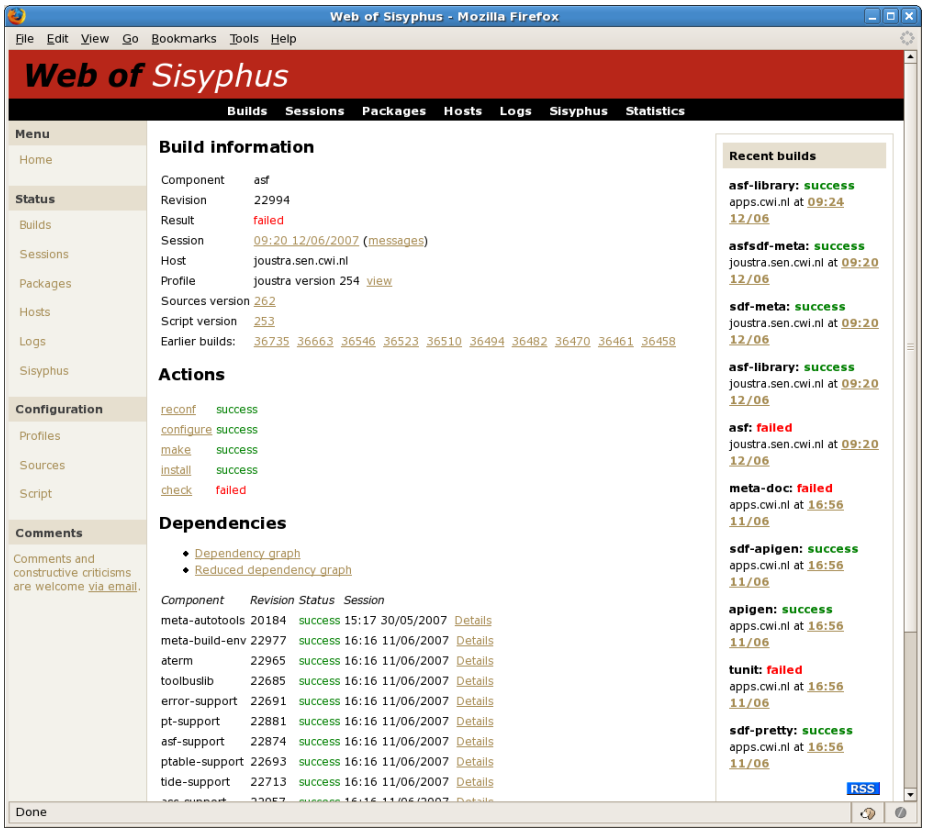


Figure 8.7: Failed build of the `asf` component. In this case the “check” action has failed. The “Dependencies” section at the bottom lists the dependencies (the table) that were used during this build. The two links give access to a graphical representation of this build’s bill of materials (see Figure 8.8)

classes. Within these methods the models can be accessed. On completion of a method the framework instantiates the template corresponding to the method, and returns the result to the browser. The HTML templates were styled using a stock open source style sheet called Gila[5].

RoR supports Ajax out of the box. Ajax is a new paradigm for developing web applications based on JavaScript. Using Ajax one can send requests to the server in the background and update fragments of a page in the browser without having to do a full page refresh. The Sisyphus main page is dynamically updated this way when a builder is executing a build, thus leading to a "real time" view on build activity.

To get an impression of the front-end of the Web of Sisyphus, Figures8.5, 8.6, 8.7 show three screen shots of the main page of Sisyphus and two release pages of the ASF+SDF Meta-Environment [92] and the `asf` component respectively. The main page of Sisyphus (Figure 8.5) shows the state of each hosts running a Sisyphus builder. The release pages show all information related to a single build. For instance, for the Meta-Environment (Figure 8.6), it shows that the current revision of the component corresponding to the Meta-Environment has revision 22311. Below the revision number, some details are shown on this particular build: the date and time of the build cycle (called "session" here), the host the build was performed on and the versions of the files that configure the build process: the profile for host specific configuration, the list of source locations managed by Sisyphus and the script (discussed in Section 8.4.

The last three lines of the section "Build information" show three release packages for the Meta-Environment: the distribution (a source package of only this component, i.e. a source package of `asfsdf-meta`), the bundle (a composite source package containing the closure of this component) [26], and a binary install script for immediately installing the Meta-Environment on Linux.

The lower part of the page shows the links for each of the build actions that have been performed during the build of this component. In this case, all build actions have been completed successfully.

The screen shot of Figure 8.7 shows the release page for a failed build. At the bottom links that can be clicked to find dependency information: which builds were used during this build and how has this build itself *been* used. Additionally there is a link for showing the dependency graph of this build. This in essence captures the bill of materials. An example is shown in Figure 8.8. The figure shows the bill of materials for the failed build of the `asf` subsystem of the ASF+SDF Meta-Environment. The nodes in the graph represent builds, the edges capture the "use" relation between builds. Clustering of nodes indicates the sessions.

As can be seen from labels of the clusters, the build of `asf` was executed in the session (build cycle) of 12th of June 2007, starting at 9:20. However, the dependencies originate from earlier sessions, notably, from one day earlier. The most recent build of the bottom component, `meta-autotools`, is from May 30th. This means that this component has not received any changes since that day.

[5] Found on `http://www.oswd.org`.

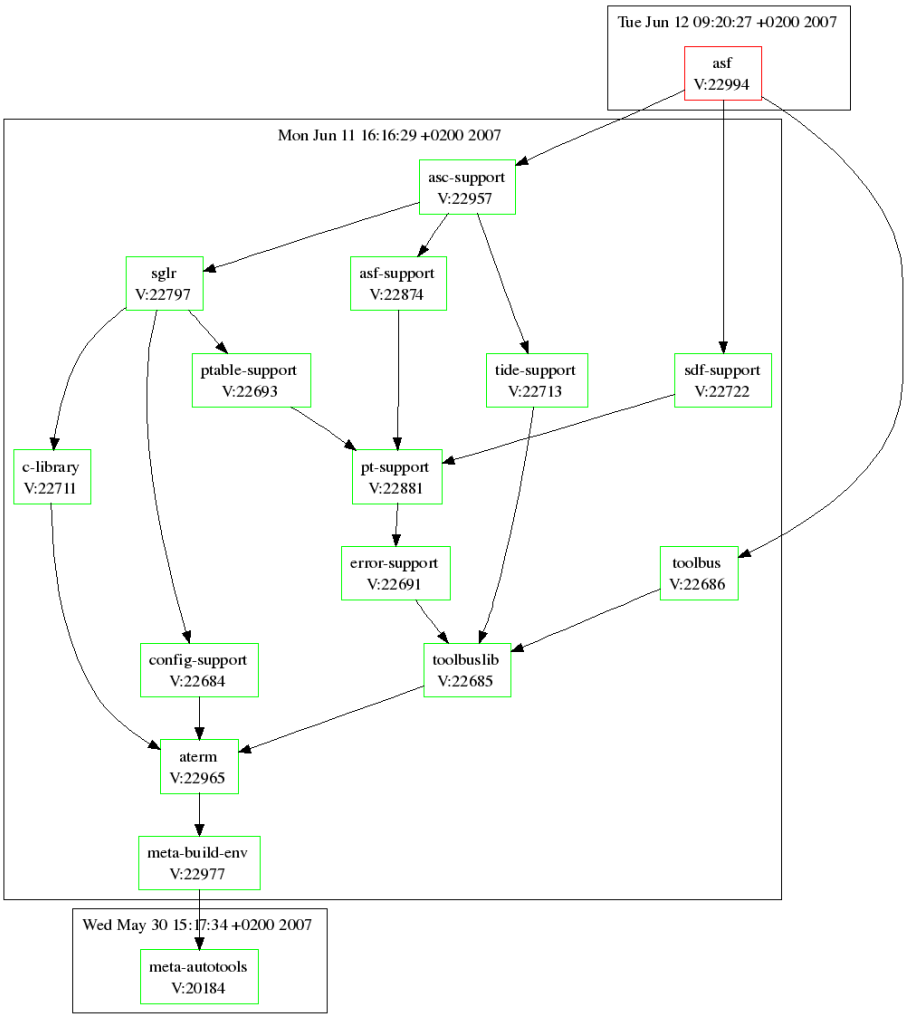


Figure 8.8: Build dependency graph of the `pandora` component

Subsystem	Language	SLOC
Back-end	Ruby	4866
ORM	Ruby	168
Front-end	Ruby	1096
Templates	HTML	1049
DB Schema	SQL	122

Table 8.1: Sizes of different parts of Sisyphus

8.6.2 The Size of Sisyphus

The implementation of Sisyphus exploits a number of third-party software packages: ActiveRecord (ORM), RoR (Web Framework), Gila (style sheet), PostgreSQL (database), YAML (configuration file parser), Subversion (configuration repository), and Apache (Web server). This surely has a impact on the size of the implementation. Table 8.1 shows the sizes of the different parts of Sisyphus in source lines of code (SLOC), i.e. without counting empty or commented lines.

The table shows the figures for Sisyphus' back-end (the builder subsystem), the ORM, which is shared among the back-end and the front-end, the front-end, the HTML templates and finally the SQL schema. The back-end is most interesting: the code in it contains the core algorithms described in previous chapters and interfacing with the build environment and the file server. Accessing the database has been made completely transparent by the ORM.

The front-end deals with serving release packages and presenting build results stored in database in an informative way. The complete bill of materials can be inspected as well as the logs resulting from each build action. Apart from the just displaying such results, the front-end generates dependency graphs using the graph visualization tool GraphViz[6]. An example of this is displayed in Figure 8.8.

8.7 Related Work

8.7.1 Similar Tools

Many continuous integration tools exist both open source and commercial. For an overview of many systems, the reader is referred to:

```
http://damagecontrol.codehaus.org/Continuous+Integration+
Server+Feature+Matrix
```

This page consists of a feature matrix for each of the systems. A system-by-system comparison with Sisyphus is outside the scope of this chapter. However, we note that the use of a database to store bill of materials, versioning configuration parameters, incremental builds and backtracking are distinguishing features for Sisyphus. Below

[6] See `http://www.graphviz.org`.

we compare Sisyphus to some (research) tools in the domain of release, delivery and deployment.

The Nix [30] deployment system is also used for continuous integration and release in an incremental fashion. However, the focus of Nix is on deployment and requires all software dependencies, including compilers, to be managed by Nix. This way tool versioning and tracing is obtained for free. In Sisyphus versioned configuration files are used to capture changes in the environment. Sisyphus and Nix both perform well on automating integration and release, as well as performing these processes incrementally. The Software Dock [45] and Software Release Manager [97] however, do not perform automatic integration and require explicit release version numbers and dependency specification before it automatically creates a release. On the other hand, these tools are better at dealing with distributed release repositories.

Software delivery and deployment tools can be broadly divided in two categories: those that are explicitly component-based and those that are not [53]. In the comparison of Sisyphus to related systems from the perspective of delivery and deployment we only consider component-based tools. These include Nix, the Software Dock, JPloy [68], the package managers à la APT/RPM [5, 85], and the source deployment systems similar to FreeBSD's Ports system [24, 79, 109].

Unlike most other tools, the Sisyphus approach to delivery is most light-weight. In the current prototype the delivered application has to be extended to include Subversion client support. The user does not have to install any additional infrastructure, unlike is the case with all the other tools. Moreover Nix, APT/RPM and Ports are particularly heavy weight since they work best under the assumption that *all* software is managed by these respective systems.

The Nix system excels at maximally sharing of dependencies while at the same time ensuring safety and side-by-side installation. Sisyphus takes the opposite route for attaining safety: not sharing anything and implementing a flexible (and fast) undo feature. Not supporting side-by-side installation is only a problem if such components are shared among different closures representing different end-user *applications* which is our focus anyway; there is no deployment time sharing. Not sharing is also followed by approaches used in Mac OS X (application bundles) and Linux deployment tools like AutoPackage [2] and Klik [59]. These approaches may waste certain amounts of disk space but lead to considerably simpler models of deployment.

JPloy also ensures that different versions of the same components do not interfere and hence satisfies the safety requirement. It is, however, bound to the Java language and platform. Finally, Ports like systems also allow side-by-side installation, but these systems only delivery source code packages which can be too time consuming for continuous delivery[7].

To conclude this section we state that Sisyphus can be described as component-based continuous integration system maintaining accurate bills of materials in order to automate the practice of release. It emphasizes simplicity, genericitiy and traceability: it requires a source repository, dependency meta-data and build templates,—nothing more. Inevitably, simplicity comes at the cost of something else. In this case this is not being able to select specific versions (although this could be super imposed) and

[7] As an aside, both Sisyphus and Nix deliver source-based releases too.

assuming a rather simplistic deployment model. Nevertheless, we think that Sisyphus fills in the following niche: automating the release of complex user applications that are released as a whole, where no heavy, formal release processes are required. We have experienced that the low risk, low overhead approach of Sisyphus is quite useful in such a setting.

8.7.2 Discussion

One of the design goals of Sisyphus was to minimize the additional maintenance that is required to achieve automation. This, for instance, precluded versioning of dependency relations since keeping these synchronized can be a true maintenance headache. In the Sisyphus approach this is not required. Only unversioned dependencies have to be written in a dedicated file among the sources of each component. Such files only change when the architecture changes.

Another design goal, which has been achieved quite satisfactory, is that Sisyphus would be platform independent. In principle, Sisyphus is completely independent of programming language, operating system, build tools and version control systems. This means that it has a low barrier to adoption.

A distinctive feature of Sisyphus is backtracking. Experience shows that this feature is essential for improved build feedback; this is corroborated by comments of programmers, who do not have to wait until somebody else has fixed the build failure.

Finally, we think that Sisyphus is sufficiently light-weight for implementing continuous delivery to many users. The technique of binary change set composition can be used for efficiently self-updating applications, without requiring any additional effort from users apart from installing the first release. This can be clear advantage over other, more intrusive, deployment systems, such as Nix.

Automatic release means automatic version selection; especially in the presence of backtracking this may lead to a impression of lack of control. It is not always easy to assess the contents of a release. More tooling is required to, for instance, clearly visualize the differences between two releases in order to improve feedback value for the programmers. Additionally, automatic derivation of release notes from developer change logs has not received enough attention.

Another problem with automatic version selection is that it is not possible to specify manually which set of revisions of components *should* be released (and this may sometimes be required). Sisyphus only builds and releases the bleeding edge; however, Sisyphus attempts to release *as many as possible*. The same holds for binding requires interfaces to source trees: the revisions of source *at the time of building* are the ones that will be used.

A disadvantage of Sisyphus, related to binding of dependencies, is that there is no support for branches. Currently only one code line per component will be built and released. This is discussed in more detail below where we survey directions for future work.

Sisyphus does not support the configuration interfaces of Part II, and hence the releases that Sisyphus publishes are not statically configurable. However, these configuration interfaces present almost insurmountable problems from the perspective of

traceability and updating. It is not in general possible to deal with change in configuration interface; for instance, how would feature selections be automatically migrated to satisfy a new feature description?

Finally, communication between Sisyphus and users is uni-directional (from Sisyphus to users). For collecting usage statistics and problem reports this communication must be bidirectional. Currently, Sisyphus' integration with issue-tracking software is particularly weak, whereas there are obvious opportunities and benefits here, for instance, for automatically reproducing a release based on the exact release version attached to a bug report. The Pheme knowledge distribution framework could be exploited for feeding user data back into the Sisyphus database [52].

8.8 Conclusion & Future Work

In this chapter we have introduced the design of the Sisyphus Continuous Integration and Release system. We posed that a continuous integration and release system should be platform independent, highly configurable and produce traceable release packages. These technical requirements were evaluated in the context of Sisyphus.

We then discussed different aspects of the implementation of the system, such as the deployment architecture, its dynamic execution model, how the system is configured and the internal data model used to produce traceable release packages. Finally, we described the front-end part of Sisyphus called "Web of Sisyphus" and presented some details about the size of Sisyphus and the use of third-party software.

We have compared the Sisyphus to similar tools and discussed its advantages and disadvantages. Below we discuss some directions for future work. These directions particularly concern the future of the Sisyphus system and include the addition of integration support on branches and support for distributed component repositories. Distributed component repositories enable the use of multiple Sisyphus instances as integration and release hubs in so-called software supply networks (SSNs) [56].

8.8.1 Future Work 1: Branch Support

The Sisyphus tool does not address the SCM facet of change isolation: branching. Components are implicitly assumed to correspond to a single code line. There is no support for branching. Branching means that development activities can be divided over separate evolution lines of the same sources. For instance, bug fixing could be performed on a different branch than the development of new features. Thus these two kinds of development activities can be parallelized. Once in a while the bug fixes can then be merged from the bug fix branch to the feature development branch. Because Sisyphus currently does not manage branches, there is only continuous integration feedback on one branch, the "trunk" of main line of a component.

In order to allow multi-branch continuous integration (and hence release) we aim to investigate the notion of dependency injection [38] to make the binding of components (for instance in dependency specifications) to actual source locations a first-class citizen. With this kind of *source tree dependency injection*, one could integrate different

compositions of components in different ways. This would also allow more flexibility in using branches as a way for implementing variations. For instance, one could say that two branches belonging to the same component, in a way, both implement the same interface. That is, if a certain source tree has a dependency on a component *C*, there may be multiple implementations of this "interface" *C*. Initial research on this has started, and it turns out the specifications of binding of source trees to component names are similar to object-oriented classes. This object-oriented nature promotes sharing of common substructures which alleviates the maintenance overhead of keeping system description and branch organization in sync. Ultimately this could lead to a fully object-oriented and versionable description language for *systems of systems*.

8.8.2 Future Work 2: Federated Integration

Another aspect missing from Sisyphus is that component sources may originate from different organizations: the distribution concern. In order to support distributed component sources, the Sisyphus continuous integration system would run on many sites, thus leading to a network of communicating Sisyphus instances. To achieve this, the following requirements need further research:

- Knowledge sharing: replication and distribution of the database; a certain instance of Sisyphus may have to know about the build results of another instance in order to successfully a execute a build.
- Artifact sharing: similarly, the actual build artifacts (binaries) must be made available across the network of instances.
- Security: authentication, authorization and licensing need to be taken care of since sources and releases may have different, configurable levels of propriety.
- Identity: components, revisions, release numbers should be globally unique. Some form of name spacing will be required. Another possibility would be the use of cryptographic hashes, as is used in the Nix system [31].
- Topology: is the network of Sisyphus instances ordered in a hierarchy, for instance, following the dependencies of components, or is the network structured as decentralized peer-to-peer network? How does an instance know (or find out) where releases of a certain required component are found?

Our work on efficient delivery in Chapter 7 could be leveraged to share artifacts across the different instances of Sisyphus.

Part IV

Conclusions

Chapter 9

Summary and Conclusions

In the Introduction we introduced posed the following research question:

> **General Research Question** *Can we automate the release of application updates in order to realize continuous delivery? This question is asked in the context of heterogeneous component-based software product lines.*

More specifically this thesis addressed this question by elaborating on three central enablers for continuous delivery: techniques for automating configuration, integration and delivery in the context of heterogeneous component-based product lines. Below we summarize each part and present conclusions on how specific research questions have been addressed.

9.1 Configuration

Automating Consistent Configuration Continuous delivery requires that users can configure released product variants according to their specific needs, without human intervention from the vendor side. To realize this in the context of component-based development we addressed the following research question:

Q.1 Can we formally ensure consistent configuration of components and their compositions?

Configuration consists of selecting the desired set of features for a certain software component and subsequently instantiating it. In order to automate configuration our techniques must satisfy the following subsidiary requirements:

- Configuration interfaces must be specified abstractly and formally so that automatic tool-support can be leveraged for checking such interfaces and validating selections of features.
- For every valid set of features, selected by an application engineer or a user, the required product variant must be instantiated. In other words: features must map to solution space variation points that will have to be automatically bound after the configuration task has completed.

In the context of this thesis these requirements have to be realized in the setting of component-based product lines. Since every component theoretically represents a product in its own, the configuration interfaces are specified at the level of the component. This raises the question as to how the consistency of configuration interfaces relates to the compositional structure of the product variant eventually delivered to the requesting user. Additionally it is not clear how the selection of certain features influences the configuration of dependency components or even the composition of the components itself.

Chapter 2 introduced a formalization of component-level configuration interfaces. These configuration interfaces, based on a textual variant of feature diagrams [105], were given an logic-based semantics which provides the means for automatically ensuring that users only make consistent selections of features. The configuration interfaces were then related to component dependencies in such a way that consistent configuration did not violate the dependency relations between individual components.

Software configuration interfaces often capture exponentially large configuration spaces. This makes scalable techniques for checking such interface a *sine qua non* for continuous delivery. The techniques of Chapter 2 exploit Binary Decision Diagrams (BDDs) [16] in order to make such large search spaces manageable. BDDs are very well researched and are heavily used in model-checking. In the context of model-checking they are applied at (state) spaces much larger than are required for configuration. Hence, BDDs provide a viable way for implementing the consistency checking of configuration interfaces.

Automating Product Instantiation If customers can configure product variants in a consistent fashion without any manual vendor intervention, it is then necessary that the actual configuration is enacted in the artifacts of the product variant itself. In other words, the selected features have to become *bound*. This is the subject of the second chapter in this thesis, Chapter 3. The specific research question for this chapter is:

Q.2 Can abstract configurations be mapped to compositions of sets of software components at the programming level?

Binding in this case is enacted through mapping selected features to certain software artifacts (e.g., libraries, executables, classes etc.) and then using the dependencies between those artifacts to determine the set of artifacts that constitutes the completely configured system.

The model introduced in Chapter 2 is generalized in Chapter 3 in which sets of configuration interfaces, in the form of feature descriptions, are mapped to arbitrary software artifacts which obey some kind of, implementation defined, dependency graph. As a consequence, selecting a feature thus always induces the inclusion of certain software artifacts including their transitive dependencies.

The mapping of features to artifacts is specified by the vendor beforehand and allows complex combinations of features and artifacts. For instance, it is possible to specify that *iff* feature a and b are enabled, then both artifacts X and Y are to be included. However, if only a is selected, then the only artifact should be Z. Arbitrarily complex combinations are possible. At the same time, the model-checking techniques

of Chapter 2 can still be used to ensure that feature selection is done consistently and to verify that every valid configuration leads to a valid artifact composition.

9.2 Integration & Delivery

The chapters in the first part of this thesis addressed automatic configuration and instantiation of component-based product populations. The second part, however, leaves configuration for what it is and adds the dimension of time. Software is in constant evolution, and from the perspective of continuous delivery this entails that there should be a way to continuously release these new versions. This part focused on two aspects. First, techniques are presented for extending the practice of continuous integration to a practice of continuous release. Secondly, we investigated how this can be done efficiently. The contributions in these two areas are summarized below.

From Continuous Integration to Continuous Release If a software product is to be delivered to users in a continuous fashion this software product has to be made available to users first. This process is called *release*. The second part of this thesis deals with this process in the context of component-based development. The first research question addressed is:

Q.3 Can the practice of continuous integration be extended to a practice of continuous release and delivery?

Continuous integration entails that a software system is built and tested from scratch, after every change to the sources. Most continuous integration systems (e.g., CruiseControl [90]) merely publish the results of this process in the form of a web site. Extending this process to a process of continuous release, however, leads to additional requirements. Software releases should exhibit two important properties: they should be immutable and they should be traceable. The first property states that a release should never change after it is made available to users. One can only create *new* releases. The second property requires that releases can always be traced back to the originating sources. This is important for accurately diagnosing problems encountered in the field. Without traceability no bug could reliably be reproduced.

Chapter 4 introduces the basic model underlying the Sisyphus continuous integration and release tool [103] and its implementation has been discussed in more detail in Chapter 8. Sisyphus automatically creates releases that satisfy the aforementioned properties. Moreover, it achieves this in the context of component-based development. Releases are identified with integration builds in a layer on top of the version control system. The identity of the release derives from the identity of all the sources of all the components that participated in the corresponding integration build. As a consequence, a change in the sources always induces a new release. Release information (bill of materials) are maintained in a database so that any released configuration can accurately be reproduced.

The different parts of a software product during development have to be put together before release. This process is called *integration* and is crucial for the release

and delivery of component-based systems. Since we aim for continuous delivery, a practice of continuous integration [39] is of particular value here.

Continuous integration consists of building (i.e. compiling, linking, etc.) and testing (automated unit and integration tests) the complete system, ideally after every change to the sources. This is a process that can be easily automated, and many continuous integration tools exist that perform this task[1].

If an integration fails, it should be top priority to "fix the build", because without a successfully integrated system, you have no "working version", and hence you cannot ship. Continuous integration is about rapid feedback on the evolving state of a software system. It provides a kind of serialization of changes so that every change is integrated in isolation. Postponing integration leads to those changes to interact in unpredictable ways thus creating a situation of "integration hell" where strange bugs start to surface that originate from the *interaction* between subsystems and are consequently very hard to track down. Because of the imperative of serializing changes, the development process stalls if the build fails. Changes queue up without developers obtaining any feedback, hence heading for integration hell.

Sisyphus operates by monitoring a set of (Subversion) repositories. Every time there is a change to one of the components managed by any of the repositories, the component in question is built and tested. If the build succeeds, Sisyphus proceeds by building all components that depend on the changed component. This way, Sisyphus implements *incremental* continuous integration and this saves a lot of time. Only components affected by a change in a repository are rebuilt, otherwise earlier build results (binaries, libraries etc.) are reused and thus shared among integrations.

Build sharing is shown in Figure 9.1. The figure depicts three integration cycles (identified by the number in the upper left orner of each column). The product that is being integrated in each of these cycles is identified with the top-level component, `pandora`, which represents the pretty-printing subsystem of the ASF+SDF Meta-Environment [93]. The `pandora` component has a single dependency indicated by the single edge going out from the `pandora` node. The graph in each column represents how the components have been build in this cycle: each node represents a component revision (revision numbers are however not shown).

The first cycle, 22, shows that all components transitively required by `pandora` have been built. However, in the next cycle, number 23, there were no changes to any of the components below `pt-support`. There were, however, *at least* changes in the `pt-support` component. Hence, Sisyphus only builds the components above and including `pt-support` because all those components are affected by changes in `pt-support`. During the builds of those components the build results (binaries, libraries etc.) of unaffected dependencies are reused from cycle 22. Similarly, in the third cycle, a change did occur in the sources of `error-support`: builds of lower components are reused, higher components are rebuilt. Note that build results of the first cycle are reused *again* in this case.

Sisyphus, however, is not only a continuous integration system but also a continuous release system. The tool accurately maintains how a certain component (which

[1] See `http://damagecontrol.codehaus.org/Continuous+Integration+Server+Feature+Matrix` for an extensive overview of systems.

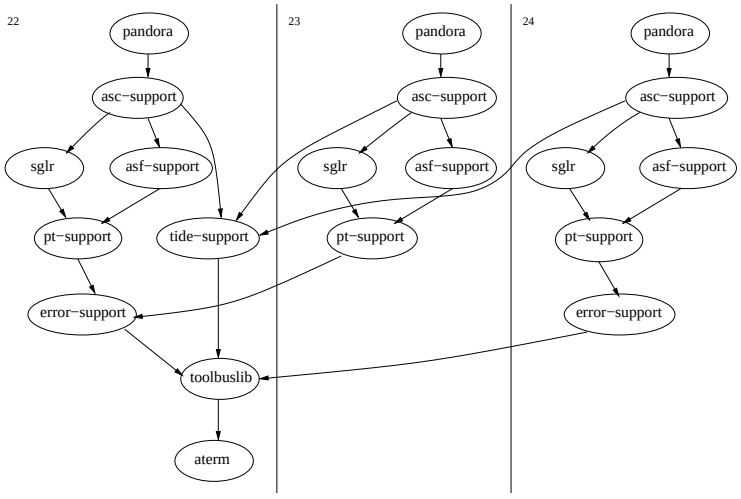


Figure 9.1: Three integrations of the `pandora` component of the ASF+SDF Meta-Environment

version using which versions of its dependencies) has been built. In fact, Figure 9.1 has been automatically derived from Sisyphus' database, and in effect represents the BOMs for three subsequent releases of `pandora`. The Web front-end of the tool uses the knowledge in the database to automatically generate a release package for every successful build that the back-end has performed, for every component and its closure.

Optimizing Continuous Integration for Delivery In order to maximize opportunities for delivery, we must ensure that integration is executed efficiently (since it is the precondition for release). Executing builds that do not lead to new releases are a waste of valuable resources; other, relevant changes may have been checked in, in the meantime. This requirement led to the following research question:

Q.4 Can we increase the frequency of delivery in the context of such a continuous release practice?

In this thesis Chapters 5 and 6 address this question. In Chapter 5 the Sisyphus model of continuous integration and release is extended with a backtracking feature that mitigates the effects of build failures. Normally, a build failure would stall the integration process; it makes no sense to try and build components against dependencies the builds of which have failed. Backtracking eliminates this propagation of failure by always building a component to a set of successfully built dependencies (if such a set exists), even if such a set is not the most recent one. This way some bleeding-edgeness is traded for increased feedback and release opportunity.

The chapter introduces two forms of backtracking: simple backtracking and true backtracking. The first form has been in operation for quite some time. Historic results from the Sisyphus database indicate that the stalling impact of build failures has indeed

been eliminated almost completely. However, simple backtracking is not optimal with respect to up-to-detentes: it misses changes. True backtracking is designed to solve this problem. However, since it has been added to Sisyphus only recently, no empirical data is available for validation yet.

Chapter 6 explores incremental system integration strategies in the context of a large-scale C/C++ based industrial system. In this case, the system is decomposed in individual components with explicit interfaces. Both interfaces and components (called *bodies*) have independent version histories. A subsystem consists of both an interface and a body and may import several interfaces; these are the dependencies of the subsystem. A subsystem is furthermore the responsibility of a single team.

When the system goes into integration, the build results of the *local* builds of each team are reused. However, each of these builds may have been executed under different assumptions with respect to the versions of interfaces that have been used. One subsystem build might have used version 1.2 of a certain interface, whereas another build used version 1.3 of the same interface. If the individual subsystems are put together to go into testing (this constitutes integration in this case), such differences in team assumptions surface as version conflicts. One way of resolving such conflicts is requesting rebuilds using the right interface versions. However, such rebuilds can take a lot of resources and effort, so simply rebuilding every subsystem that is in conflict is not an optimal solution.

Chapter 6 introduces the *build penalty* metric in order to compute the minimum set of rebuilds that is required to resolve all conflicts. Moreover, it elaborates on how existing SCM tooling can be extended to explicitly manage the notion of interface compatibility in order to prevent rebuilds at all when two conflicting interface versions turn out to be compatible. An empirical investigation of past integrations showed that the application of minimal build penalty as a way for assessing integration conflicts would have save considerable (re)build effort. Build penalty conceptually relies on a precise formalization of bill of material (BOM), which is positioned as an important—often implicitly used—concept in software engineering.

In addition to introducing, formalizing and evaluating build penalty, the chapter elaborates on how the notion of interface compatibility can be made a first-class SCM citizen in order to prevent even more rebuilds. If the changes between two interface baselines are compatible, a version conflict does not constitute a *real* conflict and no rebuild is required. We concluded by tentatively describing the requirements for extending SCM tooling with the notion of subsystem dependency and interface compatibility.

Efficiently Updating User Configurations The final contribution of this thesis consists of an efficient technique to deliver composite releases (releases of *closures*) to actual users. It provides an answer to the final research question:

Q.5 Can we realize the delivery of release packages in an efficient way?

The Sisyphus continuous integration and release system developed in Part III publishes releases for every component closure after every change on a website. In order to install such releases users still have to download and install each version manually. This is

negative incentive for updating to the most recent version in a continuous fashion. This problem is addressed in the last chapter of this thesis, Chapter 7.

Binary change set composition is a technique to efficiently and safely update user configurations to new releases published by Sisyphus. For this to work, all build results of each component build that Sisyphus executes are stored in Subversion [19]. Subversion is a VCS that basically implements a versioned file system. By storing the build results of two subsequent builds of the same component at the same location of the versioned file system, these results are stored differentially, just like source files usually are stored by any VCS. The first advantage of this is that keeping all build artifacts requires far less storage, because subsequent builds usually only differ just slightly.

For each component version in the closure that corresponds to a release published by Sisyphus, the build artifacts are put in a single directory in the version file system of Subversion by *shallow copying*. Copying in Subversion thus is similar to symbolic linking in Unix: it is an atomic operation that requires only constant space. The result is a path in the Subversion repository that "contains" (= references) all files in the closure of a release.

For Subversion clients—programs that access Subversion repositories—there is no difference between a shallowly copied file and an ordinary file that has been checked in. This means that clients can check out the path (possibly over the Internet) containing the files of the closure just like any other repository path. Checking out such path thus is a way to deliver the closure to a user. The user will obtain exactly the files belonging to the release in a single directory on the local file system. More interestingly, Subversion has a feature to update such a local copy to arbitrary paths in the repository in constant time; this feature is called *switching* a local copy. Since every build corresponds to a release, and every release has an associated closure, and finally every closure is stored at a single path in a Subversion repository, users can upgrade (and downgrade) with a single Subversion command, in constant time. Moreover, since Subversion stores only the difference between two versions, switching a working copy involves transferring just the change sets between two releases which makes this a very efficient operation. Of course, users should not be required to operate Subversion, but the interface for upgrading and downgrading has now become so simple that it can be easily implemented as part of the software product itself, which ultimately leads to continuously self-updating software.

Bibliography

[1] AutoConf. Online: `http://www.gnu.org/software/autoconf`.

[2] AutoPackage. Online: `http://www.autopackage.org`.

[3] E. H. Baalbergen. Design and implementation of parallel make. *Computing Systems*, 1(2):135–158, 1988.

[4] W. A. Babich. *Software Configuration Management: Coordination for Team Productivity*. Addison-Wesley, 1986.

[5] 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).

[6] V. R. Basili and A. J. Turner. Iterative enhancement: A practical technique for software development. *IEEE Trans. Software Eng.*, 1(4):390–396, 1975.

[7] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analyses of feature models: Challenges ahead. *Communications of the ACM*, December 2006. To appear.

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conf. on Software Engineering (ICSE-03)*, pages 187–197, Piscataway, NJ, May 3–10 2003. IEEE Computer Society.

[9] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley, 2004.

[10] D. E. Bellagio and T. J. Milligan. *Software Configuration Management Strategies and IBM Rational ClearCase. A Practical Introduction*. IBM Press, 2nd edition, 2005.

[11] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés. Automated reasoning on feature models. In O. Pastor and J. F. e Cunha, editors, *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*, volume 3520 of *LNCS*, pages 491–503. Springer, 2005.

[12] S. Berczuk and B. Appleton. *Software Configuration Management Patterns: Effective Teamwork, Practical Integration*. Addison-Wesley, 2002.

[13] E. H. Bersoff. Elements of software configuration management. *Transactions on Software Engineering*, SE-10(1):79–87, January 1984.

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

[15] 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.

[16] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.

[17] F. Cao, B. R. Bryant, C. C. Burt, Z. Huang, R. R. Raje, A. M. Olson, and M. Auguston. Automating feature-oriented domain analysis. In *Proc. of the International Conf. on Software Engineering Research and Practice (SERP'03)*, pages 944–949, 2003.

[18] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, Department of Computer Science, University of Colorado, April 1998.

[19] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Version Control with Subversion*. O'Reilly Media, 2004. Online: `http://svnbook.red-bean.com/`.

[20] R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[21] M. L. I. Crnkovic. New challenges for configuration management. In J. Estublier, editor, *Proceedings of 9th Intl. Workshop on Software Configuration Management (SCM-9)*, volume 1675 of *LNCS*, pages 232–243. Springer Verlag, 1999.

[22] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In R. Glück and M. R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings*, volume 3676 of *LNCS*, pages 422–437. Springer, 2005.

[23] S. Dart. Concepts in configuration management systems. In P. H. Feiler, editor, *Proc. of the 3rd International Workshop on Software Configuration Management*, pages 1–18, Trondheim, Norway, June 1991.

[24] Darwinports. Online: `http://darwinports.com` (May 2007).

[25] H. Dayani-Fard, Y. Yu, J. Mylopoulos, and P. Andritsos. Improving the build architecture of legacy C/C++ software systems. In *Fundamental Approaches to Software Engineering, 8th International Conference (FASE)*, pages 96–110, 2005.

[26] M. de Jonge. Source tree composition. In C. Gacek, editor, *Proceedings: Seventh International Conf. on Software Reuse*, volume 2319 of *LNCS*, pages 17–32. Springer-Verlag, Apr. 2002.

[27] M. de Jonge. Package-based software development. In *Proc.: 29th Euromicro Conf.*, pages 76–85. IEEE Computer Society Press, 2003.

[28] M. de Jonge. Build-level components. *IEEE Trans. Software Eng.*, 31(7):588–600, 2005.

[29] M. de Jonge and J. Visser. Grammars as feature diagrams. draft, Apr. 2002.

[30] E. Dolstra. *The Purely Functional Software Deployment Model*. PhD thesis, Faculty of Science, University of Utrecht, 2006.

[31] E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In L. Damon, editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, 2004.

[32] E. Dolstra, G. Florijn, and E. Visser. Timeline variability: The variability of binding time of variation points. In J. van Gurp and J. Bosch, editors, *Workshop on Software Variability Modeling (SVM'03)*, number 2003-7-01 in IWI preprints, Groningen, The Netherlands, February 2003. Reseach Institute of Computer Science and Mathematics, University of Groningen.

[33] E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 583–592, Washington, DC, USA, 2004. IEEE Computer Society.

[34] 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.

[35] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology*, 14(4):1–48, October 2005.

[36] 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.

[37] S. I. Feldman. Make – A program for maintaining computer programs. *Software – Practice and Experience*, 9(3):255–265, Mar. 1979.

[38] M. Fowler. Inversion of control containers and the dependency injection pattern. Online: `http://www.martinfowler.com/articles/injection.html`, January 2006.

[39] M. Fowler and M. Foemmel. Continuous integration. Online: http://martinfowler.com/articles/continuousIntegration.html, February 2007.

[40] T. Gilb. Evolutionary development. *SIGSOFT Softw. Eng. Notes*, 6(2):17–17, 1981.

[41] T. Gilb. Evolutionary delivery versus the "waterfall model". *SIGSOFT Softw. Eng. Notes*, 10(3):49–61, 1985.

[42] J. F. Groote and J. Pol. Equational binary decision diagrams. Technical Report SEN-R0006, Centre for Mathematics and Computer Science (CWI), Amsterdam, 2000.

[43] E. Grossman. An update on software updates. *ACM Queue*, 3(2), March 2005.

[44] C. A. Gunter. Abstracting dependencies between software configuration items. *ACM SIGSOFT Software Engineering Notes*, 21(6):167–178, Nov. 1996. SIGSOFT '96: Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, San Francisco, California.

[45] R. S. Hall, D. Heimbigner, and A. L. Wolf. A cooperative approach to support software deployment using the software dock. In *Proceedings of the 1999 International Conf. on Software Engineering (ICSE'99)*, pages 174–183, New York, May 1999. Association for Computing Machinery.

[46] D. Heimbigner and A. L. Wolf. Post-deployment configuration management. *LNCS*, 1167:272–276, 1996.

[47] I. Heitlager, S. Jansen, R. Helms, and S. Brinkkemper. Understanding the dynamics of product software development using the concept of co-evolution. In *Second International IEEE Workshop on Software Evolvability*, 2006.

[48] J. Henstridge and H. Pennington. Pkgconfig. Online: `http://pkgconfig.freedesktop.org` (May 2007).

[49] A. Heydon, R. Levin, T. Mann, and Y. Yu. The Vesta approach to software configuration management. Technical Report SRC-TN-1999-001, Hewlett Packard Laboratories, June 20 1999.

[50] R. C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, 1998.

[51] A. Jansen. Feature based composition. Master's thesis, Rijksuniversiteit Groning, 2002.

[52] S. Jansen. *Customer Configuration Updating in a Software Supply Network*. PhD thesis, Faculty of Science, University of Utrecht, 2007.

[53] 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.

[54] 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.

[55] S. Jansen and S. Brinkkemper. Definition and validation of the key process of release, delivery and deployment for product software vendors: turning the ugly duckling into a swan. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06*, pages 166–175, 2006.

[56] S. Jansen, S. Brinkkemper, and A. Finkelstein. Providing transparency in the business of software: a modeling technique for software supply networks. In *Proceedings of the 8th IFIP Working Conference on VIRTUAL ENTERPRISES (PRO-VE)*, 2007.

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

[58] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353. Springer-Verlag, 2001.

[59] Klik. Online: `http://klik.atekon.de` (May 2007).

[60] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, April 1993.

[61] P. Klint. How understanding and restructuring differ from compiling—a rewriting perspective. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC03)*, pages 2–12. IEEE Computer Society, 2003.

[62] P. Klint and T. van der Storm. Reflections on feature-oriented software engineering. In C. Schwanninger, editor, *Workshop on Managing Variabilities Consistently in Design and Code held at the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, 2004. Available from: `http://www.cwi.nl/~storm`.

[63] R. Krikhaar. *Software Architecture Reconstruction*. PhD thesis, University of Amsterdam, 1999.

[64] R. Krikhaar and I. Crnkovic. Software configuration management. *Science of Computer Programming*, 65(3):215–221, April 2007.

[65] D. A. Lamb. Relations in software manufacture. Technical report, Department of Computing and Information Science, Queen's University, Kingston, Ontario K7L 3N6, october 1994.

[66] 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.

[67] M. Lippert, S. Roock, R. Tunkel, and H. Wolf. *Extreme Programming Perspectives*, chapter Stabilizing the XP Process Using Specialized Tools. XP Series. Addison-Wesley, 2002.

[68] C. Lüer and A. van der Hoek. JPloy: User-centric deployment support in a component platform. In *Second International Working Conference on Component Deployment*, pages 190–204, May 2004.

[69] M. Mannion. Using first-order logic for product line model validation. In G. Chastek, editor, *Proc. of The 2nd Software Product Line Conf. (SPLC2)*, number 2379 in LNCS, pages 176–187, 2002.

[70] H. Mather. *Bills of Materials*. Dow Jones-Irwin, 1987.

[71] S. McConnell. Daily build and smoke test. *IEEE Software*, 13(4), July 1996.

[72] E. Meijer and C. Szyperski. Overcoming independent extensibility challenges. *Communications of the ACM*, 45(10):41–44, October 2002.

[73] B. Meyer. The software knowledge base. In *Proc. of the 8th Intl. Conf. on Software Engineering*, pages 158–165. IEEE Computer Society Press, 1985.

[74] Microsoft. Windows installer. online, 2007. `http://msdn2.microsoft.com/en-us/library/aa372866.aspx`.

[75] H. D. Mills, D. O'Neill, R. C. Linger, M. Dyer, and R. E. Quinnan. The management of software engineering. *IBM Systems Journal*, 19(4):414–477, 1980.

[76] E. C. Nistor, J. R. Erenkrantz, S. A. Hendrickson, and A. van der Hoek. ArchEvol: versioning architectural-implementation relationships. In *Proceedings of the 12th Intl. Workshop on Software Configuration Management (SCM-12)*, 2005.

[77] K. Pohl, G. Böckle, and F. J. van der Linden. *Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[78] M. Pool. DistCC, a fast free distributed compiler. In *Proceedings of* `linux.conf.au`, 2004.

[79] FreeBSD Ports. Online: `http://www.freebsd.org/ports` (August 2005).

[80] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443, 1997.

[81] R. Prieto-Diaz and J. M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4):307–334, November 1986.

[82] E. S. Raymond. The CML2 language. In *9th International Python Conference*, 2001. Available at: `http://www.catb.org/~esr/cml2/cml2-paper.html`. (accessed October 2006).

[83] D. Redmiles, A. van der Hoek, B. Al-Ani, T. Hildenbrand, S. Quirk, A. Sarma, R. S. S. Filho, C. de Souza, and E. Trainer. Continuous coordination. a new paradigm to support globally distributed software development projects. *Wirtschaftsinformatik*, 2007(49):28–38, 2007.

[84] D. E. Rogerson. *Inside COM. Microsoft's Component Object Model*. Microsoft Press, 1997.

[85] G. N. Silva. *APT HOWTO*. Debian, 2004. Online: `http://www.debian.org/doc/manuals/apt-howto/index.en.html` (August 2005).

[86] M. H. Sørensen and J. P. Secher. From type inference to configuration. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*. Springer Verlag, 2002.

[87] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2nd edition, 2002.

[88] A. Tarski. On the calculus of relations. *J. Symbolic Logic*, 6:73–89, 1941.

[89] E. Thiele. CompilerCache. `http://www.erikyyy.de/compilercache`.

[90] ThoughtWorks. Cruisecontrol. `http://cruisecontrol.sourceforge.net` (May 2007).

[91] Q. Tu and M. W. Godfrey. The build-time software architecture view. In *Proceedings of International Conference on Software Maintenance (ICSM)*, pages 398–407, 2001.

[92] M. van den Brand, M. Bruntink, G. Economopoulos, H. de Jong, P. Klint, T. Kooiker, T. van der Storm, and J. Vinju. Using The Meta-environment for Maintenance and Renovation. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE Computer Society Press, 2007.

[93] M. van den Brand, A. Kooiker, J. Vinju, and N. Veerman. A Language Independent Framework for Context-sensitive Formatting. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 103–112, Washington, DC, USA, 2006. IEEE Computer Society Press.

[94] M. van den Brand, P. Moreau, and J. J. Vinju. Environments for Term Rewriting Engines for Free! In R. Nieuwenhuis, editor, *Proc. of the 14th International Conf. on Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *LNCS*, pages 424–435. Springer-Verlag, 2003.

[95] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, 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 *LNCS*, pages 365–370. Springer-Verlag, 2001.

[96] A. van der Hoek, R. S. Hall, A. Carzaniga, D. Heimbigner, and A. L. Wolf. Software deployment: Extending configuration management support into the field. *CrossTalk The Journal of Defense Software Engineering*, 11(2):9–13, Feb. 1998.

[97] A. van der Hoek, R. S. Hall, D. Heimbigner, and A. L. Wolf. Software release management. In *Proceedings of the Sixth European Software Engineering Conference with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1301 of *LNCS*, pages 159–175, 1997.

[98] A. van der Hoek and A. L. Wolf. Software release management for component-based software. *Software—Practice and Experience*, 33(1):77–98, 2003.

[99] 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 *LNCS*, pages 86–100. Springer, June 2004.

[100] T. van der Storm. Continuous release and upgrade of component-based software. In E. J. Whitehead, Jr. and A. P. Dahlqvist, editors, *Proceedings of the 12th International Workshop on Software Configuration Management (SCM-12)*, pages 41–57, 2005.

[101] T. van der Storm. Binary change set composition. In H. W. Schmidt, editor, *Proceedings of the 10th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'07)*, LNCS. Springer, 2007. To appear.

[102] T. van der Storm. Generic feature-based composition. In M. Lumpe and W. Vanderperren, editors, *Proceedings of the Workshop on Software Composition (SC'07)*, LNCS. Springer, 2007. To appear.

[103] T. van der Storm. The Sisyphus continuous integration system. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 335–336. IEEE Computer Society Press, 2007.

[104] A. van Deursen, M. de Jonge, and T. Kuipers. Feature-based product line instantiation using source-level packages. In *Proceedings Second Software Product Line Conf. (SPLC2)*, LNCS, pages 217–234. Springer-Verlag, 2002.

[105] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, March 2002.

[106] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Proceedings of the Working IEEE/IFIP Conf. on Software Architecture (WICSA'01)*, pages 45–54. IEEE Computer Society, 2001.

[107] R. van Ommering. Configuration management in component based product populations. In *Proceedings of the 10th Intl. Workshop on Software Configuration Management (SCM-10)*, number 2649 in LNCS, pages 16–23. Springer, May 2001.

[108] R. van Ommering and J. Bosch. Widening the scope of software product lines: from variation to composition. In G. J. Chastek, editor, *Proc. of The 2nd Software Product Line Conf. (SPLC2)*, number 2379 in LNCS, pages 328–347. Springer, 2002.

[109] 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).

[110] Y. Yu, H. Dayani-Fard, and J. Mylopoulos. Removing false code dependencies to speedup software build processes. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 343–352, 2003.

[111] Y. Yu, H. Dayani-Fard, J. Mylopoulos, and P. Andritsos. Reducing build time through precompilations for evolving large software. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 59–68, 2005.

Acronyms & Used Notation

Acronyms

AOP Aspect Oriented Programming

APT A Package Tool

ASF Algebraic Specification Formalism

BDD Binary Decision Diagram

BOM Bill of Materials

CBSE Component-Based Software Engineering

CDL Component Description Language

CML Configuration Menu Language

COM Component Object Model

CRM Customer Relation Management

CSS Cascading Style Sheet

CVS Concurrent Versioning System

DLL Dynamically Linked Library

DSL Domain Specific Language

ERB Embedded Ruby

ERP Enterprise Resource Planning

FAME Feature Analysis and Manipulation Environment

FDL Feature Description Language

FODA Feature Oriented Domain Analysis

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IDL Interface Definition Language

LOC Lines of Code

MIL Module Interconnection Language

MSI Microsoft Installer

OO Object Oriented

ORM Object-Relational Mapping

PLE Product Line Engineering

PLOC Parsed Lines of Code

RKB Release Knowledge-Base

RoR Ruby on Rails

RSS Really Simple Syndication

RPM Redhat Package Manager

SBOM System Bill of Materials

SCM Software Configuration Management

SDF Syntax Definition Formalism

SKB Software Knowledge-Base

SLOC Source Lines of Code

SRM Software Release Management

SQL Structured Query Language

UCM Unified Change Management

UML Unified Modelling Language

VCS Version Control System

XML Extensible Markup Language

XP Extreme Programming

YAML YAML Ain't Markup Language

Relational Calculus

Operation	Notation	Definition
Domain	$\text{domain}(R)$	$\{x \mid \langle x,y\rangle \in R\}$
Range	$\text{range}(R)$	$\{y \mid \langle x,y\rangle \in R\}$
Carrier	$\text{carrier}(R)$	$\text{domain}(R) \cup \text{range}(R)$
Top	$\text{top}(R)$	$\text{domain}(R) \setminus \text{range}(R)$
Bottom	$\text{bottom}(R)$	$\text{range}(R) \setminus \text{domain}(R)$
Right image	$R[X]$	$\{y \mid \langle x,y\rangle \in R, x \in X\}$
Composition	$R \circ S$	$\{\langle x,z\rangle \mid \langle x,y\rangle \in R, \langle y,z\rangle \in S\}$
Product	$X \times Y$	$\{\langle x,y\rangle \mid x \in X, y \in Y\}$
Inverse	R^{-}	$\{\langle y,x\rangle \mid \langle x,y\rangle \in R\}$
Reflexive closure	$R^{=}$	$R \cup \{\langle x,x\rangle \mid x \in \text{carrier}(R)\}$
Transitive closure	R^{+}	$\bigcup_{i=1}^{\lvert R\rvert} R^i$ where $R^i = R \circ_1 \ldots \circ_i R$
Transitive reflexive closure	R^{*}	$(R^{+})^{=}$

Nederlandse Samenvatting

Software producenten brengen nieuwe versies uit van hun producten en gebruikers installeren ze. Dit proces wordt *software delivery* genoemd. Deze dissertatie presenteert een aantal technieken die kunnen helpen bij het vereenvoudigen en automatiseren van dit proces. De technieken worden gepresenteerd in de context van heterogene, componentgebaseerde software product families. Dit houdt in dat software producent meerdere varianten van een software product ontwikkelt, gebaseerd op gedeelde een verzameling componenten die in hun eigen tempo evolueren en mogelijk geïmplementeerd zijn in verschillende programmeertalen.

Automatisering van het delivery proces is met name relevant wanneer een software producent zich richt op het *frequent* uitleveren van nieuwe versies (*releases*). Het voorbereiden van een release is een tijdrovend en foutgevoelig proces: de software moet gebouwd worden op zo'n manier dat alle afhankelijkheden van de componenten onderling vervuld zijn, en dit alles moet worden samengevoegd worden in een formaat waarmee de gebruiker de nieuwe versie kan installeren. Als er veel componenten, veel afhankelijkheden, en veel product varianten onderhouden worden dan is duidelijk dat automatisering een grote kostenbesparing kan betekenen.

Ook voor de gebruiker zitten er haken en ogen aan frequente uitlevering van software. Het installeren van een nieuwe versie is vaak geen sinecure. Bovendien is het steeds weer handmatig installeren van nieuwe versies vervelend en foutgevoelig. Van een oude versie overgaan op een nieuwe versie van een software product heet *software updating*. Ook hier is automatisering gewenst.

Er zijn dus twee kanten aan het delivery proces. Aan de ene kant wil de producent op frequente basis een nieuwe versie beschikbaar maken. De kosten hiervan zouden zo laag mogelijk moeten zijn. Aan de andere kant zal de gebruiker deze nieuwe versie willen installeren zonder dat er teveel handelingen aan te pas komen en zonder dat hij een verhoogd risico loop dat de software niet meer werkt na de update.

In deze dissertatie ben ik uitgegaan van een wens tot *continue* uitlevering; dat wil zeggen, dat er liefst na *elke* wijziging in de software een nieuwe versie naar de gebruiker verstuurd wordt. De technieken voor het automatiseren van het uitleveren van software zijn vanuit dit perspectief gemotiveerd. Voordat ik echter concreet inga op de specifiek bijdragen, is het belangrijk om eerst een indicatie te geven waarom continue uitlevering wenselijk is in de eerste plaats. Deze vraag kun je opnieuw van twee kanten bekijken, vanuit het perspectief van de ontwikkelaars en het pespectief van de gebruikers.

Ontwikkelaars brengen constant wijzigingen aan op bepaalde plaatsen in softwa-

re. Een manier om te weten te komen wat het globale effect is van die wijzigingen, is het bouwen en testen van de software; dit proces wordt *integratie* genoemd. Als de software niet bouwt of wanneer bepaalde tests een negatief resultaat geven, is dat een indicatie dat er iets mis is sinds de vorige integratie. Het is hier essentieel dat wijzigingen geïntegreerd worden *zodra* ze beschikbaar zijn. Dit kan als volgt gezien worden. Stel dat een programmeur één bepaalde module wijzigt sinds de laatste integratie. Deze wijziging wordt vervolgens geïntegreerd met de rest van het systeem, en het bouwen faalt (een "failing build"). Het is nu duidelijk dat de oorzaak van dit falen hoogstwaarschijnlijk in dat betreffende bestand zit. Dat is waardevolle informatie, want het betekent dat ontwikkelaars dáár als eerste kunnen kijken om het probleem op te lossen. Als echter de integratie pas veel later plaatsvindt, en er dus niet één enkele wijziging geïntegreerd wordt, is het diagnosticeren van een falende build veel moeilijker. Niet alleen zijn er nu meerdere wijzigingen die de oorzaak kunnen zijn van het probleem, maar ook hun *interactie*: twee wijzigingen in uiteenliggende modules kunnen onverwacht tot een fout leiden. De praktijk wijst uit dat zulke fouten heel moeilijk te vinden en te corrigeren zijn. Het is daarom dat continue integratie een "best practice" in software ontwikkeling is.

Voor continue *uitlevering* is een soortgelijk argument aan te voeren. Draait continue integratie om de onmiddelijke feedback omtrent het effect van een wijziging,—bij continue uitlevering gaat het om feedback die een bepaalde groep gebruikers aanlevert. Hierbij valt vooral te denken aan foutrapportage ("bug reports"). Ook hier is het waardevol om snel te vernemen van de problemen en wensen die een gebruiker heeft. Snellere feedback betekent sneller handelen en uiteindelijk een tevredener gebruiker.

Als de software producent sneller in staat is verbeteringen door te voeren, en in staat is die wijzigingen ook zo snel mogelijk bij de gebruiker door te voeren, dan heeft dit een positief effect op de kwaliteit en functionaliteit van het product. Dit gezegd zijnde, betekent dit wel dat het updaten van een oude naar een nieuwe versie wel zo onzichtbaar mogelijk moet zijn; frequente uitlevering van nieuwe versies houdt namelijk ook in dat de wijzigingen per release relatief klein zijn. Dat het installeren van een nieuwe versie meer werk kost dan de verbetering in het product rechtvaardigt, moet ten alle tijden voorkomen worden.

Nu ik de achtergrond van deze dissertatie geschetst heb, is het tijd om de vraag te stellen wat er nodig is om het release en update process te automatiseren. Ik heb me in dit werk geconcentreerd op de automatiseren van drie aspecten van software delivery: configuratie, integratie en delivery. Hieronder introduceer ik elk van deze termen en beschrijf in het kort de bijdragen van dit werk.

In de context van een product familie wordt er niet slechts één enkel product uitgeleverd, maar verschillende product *varianten* gebaseerd op dezelfde verzameling componenten. Configuratie is dan gedefinieerd als het instantiëren van de wenselijke variant door bepaalde kenmerken (*features*) van een software product te selecteren. Het eerste deel van deze dissertatie behandelt technieken om te zorgen dat er in het configuratie proces geen inconsistente keuzes gemaakt kunnen worden. Dit is mogelijk door middel van een formalizering van feature beschrijvingen die gebruikt worden om configuratieruimtes van product families inzichtelijk te maken. Ik heb deze feature beschrijvingen geherformuleerd in de context van componentgeoriënteerde software ontwikkeling en daar formele analyses op losgelaten. Tenslotte behandel ik hoe op

basis van een bepaalde selectie features een product variant afgeleid kan worden.

In het tweede deel komen technieken aan bod om de integratie en delivery processen te automatiseren en optimalizeren. Deze technieken zijn onderdeel van het prototype systeem Sisyphus. Met als doel de *feedback loop* tussen integratie en ontwikkelaar te verkorten, voert Sisyphus de integratie incrementeel uit: als een component geen wijzigingen heeft dan worden eerdere integratie resultaten opnieuw gebruikt. Hierdoor duur een integratie cyclus door de bank genomen minder lang, met als gevolg dat ontwikkelaars eerder de effecten van hun wijzigingen kunnen waarnemen.

Een tweede techniek die de informatieve waarde van integratie vergroot presenteer ik onder de naam *backtracking* (letterlijk "terugtreden"). Dit betekent, dat wanneer de integratie van een zeker component mislukt, maar deze resultaten nodig zijn voor de integratie van een ander component, Sisyphus in het verleden op zoek gaat naar de meest recent geslaagde integratie van het vereiste component. Dit betekent dat ondanks dat één bepaald component niet door de integratie heen komt, dit geen reden hoeft te zijn dat wijzigingen in een ander (daarvan afhankelijk) component óók niet door integratie heen komt. Zonder backtracking zou het überhaupt niet mogelijk zijn de wijzigingen in afhankelijke componenten te integreren.

Sisyphus is niet alleen een systeem voor continue integratie, het kan ook gebruikt worden voor de continue uitlevering van componentgebaseerde software systemen. Continue integratie is een hier een vereiste voor. In feite zijn in Sisyphus de voordelen van continue integratie gecombineerd met de voordelen voor continue release. Continue integratie alleen is echter niet genoeg. Omdat een release traceerbaar moet zijn naar de originele broncode is het belangrijk dat accurate versie informatie bijgehouden wordt over wat er in een release belandt. Dit is bijvoorbeeld essentieel wanneer een gebruiker een probleem rapporteert: om het probleem te reproduceren moet bij de ontwikkelaars bekend zijn welke versies van welke componenten bij die gebruiker geinstalleerd is. Dit soort meta-data wordt ook wel de *bill of materials* (BOM) genoemd. Sisyphus zorgt hiervoor door de integratie resultaten van alle componenten in het systeem met exacte versie nummers opgeslagen worden in een databank. Dit maakt het mogelijk om uit de integratie resultaten automatisch releases te genereren.

Een andere een reden waarom het belangrijk is kennis te hebben van welke versie reeds geïnstalleerd is bij de gebruiker is de manier waarop updates verstuurd worden. Tegenwoordig gebeurt software updating voornamelijk via het internet. De gebruiker krijgt een nieuwe versie van het product die vervolgens de oude versie zal vervangen. Accurate versie informatie maakt het mogelijk om een nieuwe versie efficiënter naar de gebruiker over te hevelen, en wel door niet elke versie opnieuw, in zijn geheel, op te sturen, maar alleen het verschil tussen de geïnstalleerde en de nieuwe versie. Dit is belangrijk omdat een snelle, efficiënte manier van updaten, bijdraagt aan de onzichtbaarheid ervan. In het laatste deel van mijn dissertatie beschrijf ik hoe dit eenvoudig gerealiseerd kan worden voor applicaties bestaande uit heterogene, binaire componenten. De infrastructuur die hiervoor nodig is aan de kant van de gebruiker kan geïncorporeerd worden in de applicatie zelf. Gecombineerd met de continue uitlevering van Sisyphus leidt dit, met minimale overhead voor zowel gebruiker als ontwikkelaar, tot een vorm van *self-updating software*.

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in µCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty

of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trcka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery Universiteit van Amsterdam, CWI*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-15