Safe Component Updates

Alexander Stuckenholz
Department of Data Processing Technologies,
FernUniversität in Hagen, Germany
Alexander.Stuckenholz@FernUni-Hagen.de

Andre Osterloh
Department of Algorithms and Complexity,
FernUniversität in Hagen, Germany
Andre.Osterloh@FernUni-Hagen.de

Abstract

Component updates always imply the risk of negatively influencing the operability of software systems. Because of wrong combinations of component versions, dependencies might break, methods which do no longer exist, might be called or methods might not be compatible to provided interfaces, objects, or classes. In this paper we model the problem of finding a well-configured system consisting of multiple component versions as a *Boolean Optimization Problem*. We introduce objective functions and constraints, which use Branch-and-Bound for restricting the search space and yield most recent, minimal systems.

Categories and Subject Descriptors D.2.9 [*Software Engineering*]: Management

General Terms Management

Keywords Component Based Software Development, Updates, Component and System Evolution, Compatibility, Boolean Optimization, System Synthesis

1. Introduction

Never change a running system! Every system administrator knows this rule to prevent unforeseen incompatibilities, which often cause breakdowns and sleepless nights. But sometimes one has to replace parts like components with newer versions because of serious security issues, functional limitations or quality improvements.

By updating a system, i.e. replacing components with newer versions, dependencies may supervene or disappear. The system changes over time; a process which is often referred to as *system evolution*. Most recent investigations (cf. [3]) concerning the source of defects of object oriented software systems indicate that missing components or wrong component versions are the most frequent reasons for configuration problems of current systems.

However, there are almost no tools that are able to prevent these situations. Configuration management lacks tools for automated configuration which combine compositional reasoning with automated versioning and dependency analysis (cf. [21]).

In this paper, we will introduce a mechanism which constructs well-configured component based systems by combining available component versions and adhering to constraints like favoring recent versions, minimizing the number of components in a system or minimizing the number of substitutions between updates.

The system is specially designed for situations in which new component versions are *not* exact substitutes for their predecessors. Those components introduce new dependencies or change parts, on which other components in the system rely. We assume that, in those situations, a balanced configuration can be found by smartly combining available component versions. That is the reason why we do not require backward compatibility between component versions, which is the basis of most packaging systems in this area, but allow unpredictable evolution of all involved components.

The paper is structured as follows. The next section introduces the original problem of incompatible component updates in more detail. Section 1.2 lists related approaches like Linux packaging to solve the update-problem and the application of optimization methods. In section 2, we introduce a formal model to automatically derive dependency information on available components. Section 3 introduces a simple component based system, which we will use as a running example for the rest of the paper. In section 4 we introduce objective functions and constraints for the problem in the normal form of a Boolean Optimization Problem. Furthermore we prove the NP-hardness of the combinatorial problem of finding well-configured system configurations. Section 5 addresses the search for solutions by calculating upper and lower bounds and their usage in cutting the search-tree by Branch-and-Bound algorithms. Section 6 mentions real-world projects in which these methods are currently evaluated and presents the results attained so far. Finally section 7 summarizes the results and gives some prospects to open questions and further research.

1.1 The Problem

Component Based Software Development (CBSD) is defined as the planned integration of preproduced software components (cf. [4]). The main aim is to reduce development costs and shorten the time-to-market by massive reuse of binary components. Moreover the product quality is increased by frequently utilizing software artifacts.

Even today many software developers spend their time with reinventing the proverbial wheel, although the reuse of software components in an early development state would eliminate the necessity to redevelop similar functionality again. However, CBSD has certain drawbacks if compared to conventional software architectures: Direct and indirect dependency relations between the components of a system arise when software components are reused throughout the system's applications. Figure 1 clarifies this structural difference between monolithic and component based architectures.

In conventional, monolithic software architectures, all required functions of the applications A and B are implemented internally. With such a structure, a substitution of an erroneous or non-performant function $F(x)$ is almost impossible, because all

GPCE'06 October 22–26, 2006, Portland, Oregon, USA.

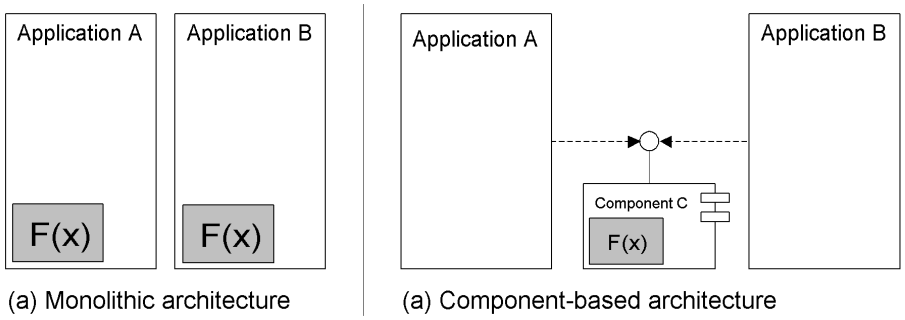

Figure 1. Code-redundancy and coupling with and without the usage of component based architectures

applications of the platform have to be analyzed with regard to their usage of $F(x)$ or similar functions.

Those situations precariously affect the maintainability of software systems. In December 2004, a critical security hole was detected in the GDI+ component, which renders JPEG files on Windows platforms (see [6]). As multiple applications locally deployed the component as a DLL, it was installed in parallel in many cases. Thus, the security hole could not be fixed by simply substituting one centrally installed component.

The CBSD tries to avoid those problems. Here the shared functions are bundled and rolled out into components, reused and utilized in the system's applications. In figure 1, component C provides function $F(x)$ for multiple applications. This structure avoids code-redundancies but introduces coupling and indirect dependencies between applications to some extend: If a shared component (like C in figure 1) has to be substituted, this may lead to incompatibilities, because the provided interface of such a provider has been changed. Those changes do not necessarily impact all dependent components, but only those, that make use of the changed or removed interface parts.

A real-world example of such a situation was the so called DLL-Hell (cf. [7, 15]). Here, the setup procedure of different applications on Windows platforms have substituted shared components with their own, not necessarily newer versions. As a consequence the new applications worked as expected, but some of the previously installed programs behaved unexpectedly, if operating at all.

The original problem of incompatible changes at signature, behavior or other contract levels may arise in all situations, where components have to be subsequently replaced by newer versions. But their impact is worst where components are used simultaneously by several applications.

A conservative approach simply prohibits any changes to previously installed components or interfaces[1]. New versions of components are entirely new components, and new interfaces wrap around their predecessors. Thus, different evolutionary states of one component exist in parallel on a platform, which leads to the above mentioned drawbacks. In case of an update, the new version is installed additionally. Therefore an update does not influence the operability of the system, but it is nigh impossible to replace erroneous components with other versions. The capability to subsequently change systems is pruned or even prevented by those approaches, contradicting the strength of component based architectures.

1.2 Related Work

There are a couple of approaches that are able to check the conformity between different evolutionary states of a component and automatically detect incompatible changes. The most auspicious systems in this area have been compared by us in [21]. Depending on the system, one or more contract levels (syntax, behavior, synchronization, quality) are included into the analysis, if a certain component may substitute another completely. If an administrator wants to upgrade n components and the conformity check assures the substitutability for all of them, the upgrade is allowed.

In situations where one or more components are not exact substitutes of their predecessors, this kind of checks do not provide useful results. If the resulting or different configurations would form a conflict-free configuration again, a more holistic approach is required. We therefore propose a system, where in case of non substitutability of one or more components, a new, well-balanced configuration is synthesized automatically. In that case we revert to the multiplicity of component versions, which are available in most cases to combine them so they form a well-configured system.

Thus, the problem is to find a valid combination of component versions, where the final system contains at most one version of a component and fulfills further constraints like up-to-dateness. Such combinatorial problems are well known to mathematicians, but still belong to the most complex problems.

The graph coloring problem sketched in [18] shows another example in which wireless network providers search for optimal frequency allocation for their adjacent transmitters in order to reduce interference occurrences. These problems often contain a large state space where each state has to be evaluated against domain-specific heuristics and tested to see whether they are goal states. Although most of these problems are NP-complete, by introducing smart heuristics to prune the search space, solutions can be found very efficiently in average cases.

In open source software systems there also exists a notion of a component, namely a software package (like TeX, the C compiler etc.) which can be installed, updated, or removed. A package may require other packages or will not work with others. Vendors of Linux distributions have sophisticated management tools based on package formats, like the Debian [9] or the Red Hat [1] package format, that check whether the state of the system is consistent (i.e. there are no incompatibilities). But there are certain drawbacks of these systems, which in worst-case may lead to inconsistent system states.

Both systems and their tools rely on different heuristics, that are pragmatic but constricts the solution space for finding compatible systems. In case of an update, tools like *apt* or *rpm* will always try to install the newest packages only. If a well-defined system state could only be derived by moving different packages back to an older version, these tools are not the best choice. Furthermore it is accepted, that deb- or rpm-packages are always downward compatible to older versions. Packages that would violate this rule are simply renamed (e.g. with the new major-revision-number in

[1] The component model COM prohibits interface evolution (cf. [16].

the package name) and henceforth exist in parallel with the older version in the package-repositories.

Our mechanisms for finding best system states do not make those kinds of assumptions and consider all potential candidates in form of component versions for the search for compatible system states.

2. Interface Matching

In order to synthesize an optimal system configuration from a set of component versions, we initially need a detailed notion of components, versions and their connectors, the interfaces. Therefore we introduce a formal model to specify components in order to derive information about the dependencies between component versions. These dependencies constitute the constraints of the optimization problem, which determine the criteria for well-configured systems.

Usually such dependency information is given by manually created specifications, evolution rules and versioning policies. In the Linux packaging systems (see section 1.2), package maintainers manually define these dependencies in the meta-data of the package. Systems like the .Net-framework assume that required assemblies with the same minor-release numbers as the one they were built with, are compatible (backward compatibility).

Such rules always restrict the development and sometimes cause situations, in which multiple component versions have to be installed in parallel to ensure compatibility. Additionally, manual specifications always imply risks of human errors or simply wrong assumptions about the usage of components.

Based on the following model we therefore implemented methods to automatically derive the required information. By static code analysis it is possible to extract the signatures of both exported and imported interfaces for each component version. We utilize these fine grained signature specifications to automatically find compatible matches between required and provided interfaces of available components.

In the next section we define the fundamental terms interface, component, and system that are required for the rest of the paper.

2.1 Interfaces, Components and Versions

Components are encapsulated against their environment. The details of the implementation are not visible for the component user and should not have any importance for their usage. The whole interaction with the implementation is carried over their interfaces. We define as follows:

DEFINITION 1 (Interfaces). *An **interface** is a finite set of **method signatures** $i = \{m_1, ..., m_{i_n}\}$, and I denotes the finite set of all interfaces.*

Interfaces represent one part of the contracts of a component with its environment. In the component based software development, several levels of contracts can be identified. These are (1) simple or syntactical, (2) behavioral, (3) synchronization and (4) quality (cf. [2]). Our approach aims for the automatic detection and reduction of component incompatibilities between component contracts. The levels of behavior, synchronization, and quality can be only included in such a compatibility analysis by using specifications which must have been manually created by the component developers. As components only rarely come with those specifications, our approach will initially be reduced to the syntactical level of component contracts.

As we are interested in the evolution of component based software systems, we assume that components are realized by at least one version. The terms revision, a modification which leads to a new version, and a variant, a component which implements the same interfaces but uses another implementation, can directly be taken from the area of *Software Configuration Management* (see [5]). In accordance with other areas of software engineering, the time dependency of software artifacts is expressed by adding version identifiers to distinguish the elements. This procedure also found its way to the Component Based Software Development.

An instantiation of a component (i.e. a version) provides services for and requires services from its environment via its interfaces. We define as follows:

DEFINITION 2 (Components and Versions). *Let R be the set of all possible **component versions**, C be the set of all possible **components**, and **isVersionOf** $\subseteq R \times C$ (or **IVO** for short) be a right-unique and left-complete relation.*

*A **component version** $r \in R$ is a tuple $r = (t, I_R, I_P)$ where t is a version identifier (e.g. the* major.minor.build *scheme[2]), I_R is a finite set of required interfaces, and I_P is a finite set of provided interfaces.*

The relation IVO connects versions $r \in R$ to their components $c \in C$. For every version $r \in R$ there should be at most one component $c \in C$ such that $(r, c) \in IVO$. Hence we demand IVO to be right-unique. A version without any component makes no sense, so IVO has to be left-complete.

In this paper we assume that a partial ordering of the version identifiers exists, such that we are able to make decisions about the age of different component versions with the help of this ordering.

In component based software development components are put together to form bigger systems. This is their original meaning (cf. [24, 20]). We define a system S as follows:

DEFINITION 3. *A **system** S is a finite set of component versions $S = \{r_1, ..., r_n\}$ such that for every component $c \in C$ we have at most one version $r_i \in S$, i.e. $\forall r, r' \in S. \forall c \in C. (r, c) \in IVO \wedge (r', c) \in IVO \Longrightarrow r = r'$.*

As a general rule the set of component versions in S consists of components from third parties (so called COTS[3]), proprietary components, configurators, which connect multiple components, and system components, which offer fundamental services like persistence or transaction safety. The system developer combines the components to create an executable application. Through this specific selection, creation and connection of multiple components, the ultimate semantic of the system S arises.

As software developers and system maintainers, we are certainly only interested in these kinds of systems which are well-composed, which means that all required interfaces in S are covered by compatible counterparts.

DEFINITION 4 (Conflict Value). *We define $R^S := \bigcup_{r_i \in S} r_i.I_R$ as the set of all required interfaces and $P^S := \bigcup_{r_i \in S} r_i.I_P$ as the set of all provided interfaces in a system S.*

It is now possible to define a relation isCompatibleTo (or ICT for short) which exists for all couples $(i_a, i_b) \in I \times I$ where the provided interface i_a is compatible to the required interface i_b. In that case we call i_a the compatible counterpart to i_b in S.

The set of uncovered interfaces $I_{conflicts} \subseteq R^S$ is now defined as the subset of R^S whose elements i_a do not have a compatible counterpart in P^S: $I_{conflicts} = \{i_a | \nexists i_b \in I^P : (i_b, i_a) \in ICT\}$.

[2] The *major-minor-build* and related versioning schemes are the most common way for versioning software artifacts where the version number 3.2.1 usually denotes the third major, the second minor and the first build-release of the versioned object.

[3] commercials off-the-shelf

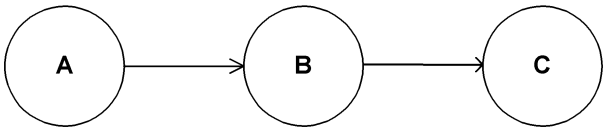

Figure 2. Component dependency graph of a system consisting of three components.

*The **conflict value** Θ^S of a system S is equal to the number of elements in $I_{conflicts}$. A System S is well composed, iff the conflict value Θ^S is zero.*

With the help of the above mentioned relation ICT, which matches required against provided interface-signatures, it is possible to automatically derive dependency information for all component versions. Thus, it is possible to generate specifications like component A requires component B. These dependency information is required in the next part of this paper where we formulate the problem of finding well-composed system configurations as a boolean optimization problem.

Before we continue and introduce our concepts for automatic synthesis of component based systems in case of incompatible updates, we first introduce a simple example, which gives us leeway enough to explain the details and the complexity of the problem in the following.

3. Example

An online Content Management System may contain a component for accessing the front-end (component A), a rendering engine (component B), and a component encapsulating basic services like object persistence (component C). To access the content in a database, the rendering engine B requires the services of component C. Moreover, the front-end A shows the results of the rendering engine B to the user by invoking its services. Hence, using the notation of the previous section this results in the following situation: $C = \{A, B, C\}$ is the set of components, $R = \{A_1, B_1, C_1\}$ is the set of possible component versions, where $A_1 = (front, \{r_1\}, \emptyset)$, $B_1 = (render, \{op_1\}, \{r_1\})$, $C_1 = (basic, \emptyset, \{op_1, op_2\})$, and IVO $= \{(A_1, A), (B_1, B), (C_1, C)\}$. Thus, we have dependency relations between the three components in the system. The usual way to visualize these dependencies is to plot them in a dependency graph[4]. Figure 2 shows the dependency graph for the example, where the components are represented as nodes and the dependencies as directed edges.

Because of bug-fixing, performance correction or feature enhancements over time new versions of the components come to existence. In worst case these components are not simple substitutes for the old versions but introduce new dependencies or remove old interfaces. Each version may introduce different dependencies to other component versions. Finding correct system configurations is, if even possible, a non-trivial task, especially in cases with large quantities of components and component versions, frequent incompatibilities and constraints like minimizing component substitutions for keeping the downtime of a system as low as possible.

To respect component evolution and the requirements of different versions the *version reachability graph* was introduced in [22]. The version reachability graph is an extended dependency graph containing all available versions of the components together with their dependencies. Dependencies to more than one version indicates an alternative; for example a component version A_1 requires either the existence of a component version B_1 or B_2. Assuming that for each of the three components of the above mentioned example two versions with different dependencies are available, for example $R = \{A_1, A_2, B_1, B_2, C_1, C_2\}$, IVO $= \{(A_1, A), (A_2, A), (B_1, B), (B_2, B), (C_1, C), (C_2, C)\}$ where A_1, B_1, and C_1 are defined as in the previous example and $A_2 = (front2, \{r_2\}, \emptyset)$, $B_2 = (render2, \{op_1, op_2\}, \{r_1, r_2\})$, and $C_2 = (basic2, \emptyset, \{op_1, op_3\})$. Figure 3 shows the according version reachability graph.

[4] The component diagram type in UML2 is also based on this method

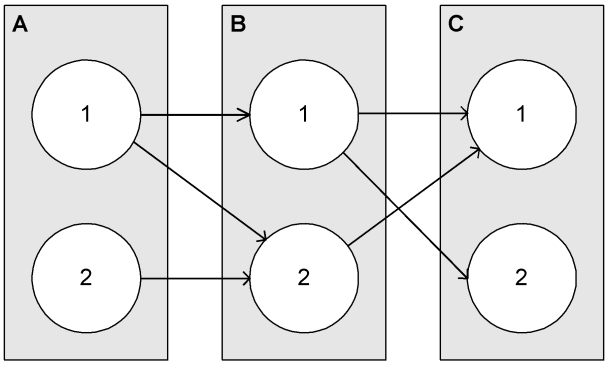

Figure 3. Version reachability graph of a simple system consisting of three components and two versions each.

Within this simple example it is easy to find the four well composed systems just by looking at the version reachability graph $((A_1, B_1, C_1), (A_1, B_1, C_2), (A_1, B_2, C_1), (A_2, B_2, C_1))$. Starting from an already existing configuration we can evaluate the solutions against constraints like the up-to-dateness of the system or the number of required component substitutions. Bigger systems require an automated method to solve this problem.

4. Modelling the problem as a Boolean Optimization Problem

In order to use standardized methods with known complexity properties we will model the ultimate problem as a Boolean Optimization Problem (BOP) in which the decision variables, in this case indicating the existence of a component version in a system, can only contain a value of zero or one. The standard-form of these problems is:

$$\begin{aligned} &min\ c^T x \\ &s.t.\ Ax \geq b \\ &x_i \in \{0, 1\} \end{aligned} \tag{1}$$

where A is an $m \times n$ matrix of rational numbers, c is an n component rational column vector and b an m component rational column vector (cf. [14], p. 6).

4.1 Objective Functions

In order to get the most recent system, we have to recompense the usage of component versions with highest version numbers during the system synthesis, which has to be transferred into an objective function of the BOP-problem.

To respect the up-to-dateness of the component versions in the objective function we first need a mapping of the version identifiers of the component versions to natural or rational numbers. For all versioning schemes it is possible to find such a mapping either by simply counting the versions starting from the oldest version or by defining methods for version identifiers like the *major-minor-*

build scheme. Consecutively we assume that $v(R_i) \in \mathbb{N}$ denotes the mapping of the version identifier of a component version R_i to a natural number.

The first objective function calculates the sum over all component versions used in the target system, which has to be maximized to get the most recent system. Here the decision variables x_{V_i}, whose value can only be zero or one, determine, if the according component version will be installed in the system in the end. For our example, the function is defined as follows:

$$\begin{aligned} &x_{A_1} * v(A_1) + x_{A_2} * v(A_2) + \\ &x_{B_1} * v(B_1) + ... + x_{C_2} * v(C_2) = T_1(x_i)\ (max) \end{aligned} \tag{2}$$

Another target was to get a minimal system with the smallest amount of components as possible. Especially with regard to the background of the above defined objective function, we have to punish the usage of unnecessary components in a system that only increase the version sum. Therefore, we have to define another objective function in order to ensure a minimal system regarding the number of used components by accumulating the sum over all decision variables x_{R_i}. For our example this leads to:

$$x_{A_1} + x_{A_2} + x_{B_1} + ... + x_{C_2} = T_2(x_i)\ (min) \tag{3}$$

Finally we want to include the required number of component substitutions between the target system we seek for and a current system into our calculation. Our goal is to keep the number of substitutions as small as possible because every component substitution takes time, in which the system will not usually be available or in a state of limited usability. For our example, we assume that our current system consists of the components A_1, B_1 and C_2. To calculate the number of substitutions we add the decision variables for all versions that are not already used in the system. For the example this yields:

$$x_{A_2} + x_{B_2} + x_{C_1} = T_3(x_i)\ (min) \tag{4}$$

We now have multiple objective functions which we want to optimize to find a solution. It is possible to prove that solutions for the linear relaxation of single objective functions are always located on a vertex of the n-polyhedron given by the constraints of an optimization problem (cf. section 4.2). In best case, the relaxation is also an integer solution, or we found a good upper bound for a search (see section 5.1). In case of multi objective functions, the optimal solution is located on the edge of the n-polyhedron between the vertices of the single optimal solutions.

As a first idea we could create a new objective function T by superposing the sub-goals and weighting them with weighting factors w_i. In fact it is difficult to find constant weighting factors influencing the optimization-process in a favored way. Hence we solve the problem against all objective functions consecutively but add additional constraints to succeeding calculations to minimize the difference to previous results. This procedure is also known as *goal programming* (cf. [17, 19]).

In our example, we would first solve the problem against $T_1(x_i)$ where we derive a maximum sum for the version-numbers of 5. Now we can solve the problem against $T_2(x_i)$ but additionally trying to minimize the difference between the version sum and the resulting version sum of the result of $T_2(x_i)$. By changing the order of applied objective functions and weighting the detected differences to results calculated earlier, we can influence the optimization process.

4.2 Constraints

So far, we only defined characteristics of optimal solutions, but the state space which is searched is not given yet. Therefore we have to introduce constraints which model the requirements of the system.

The first requirement is that we only want to allow the existence of at most one single version of each component in the system. To compensate for their lack of mechanisms for dependency checking and configuration reasoning, multiple component models, frameworks and operating systems allow the parallel existence of multiple component versions (cf. [21]). On the one hand this procedure ensures that dependencies do not break whenever a new component version will be installed but on the other hand it makes it in some cases almost impossible to fix important security problems by replacing a single, centrally installed component.

In our example, we have to ensure that if A_1 is installed, A_2 is not installed, too. Those kinds of requirements can be easily expressed in propositional logic. In the following the logical variables X_{R_i} denote the logical equivalence to the algebraic decision variables used before.

$$\begin{aligned} X_{A_1} &\rightarrow \neg X_{A_2} \\ X_{B_1} &\rightarrow \neg X_{B_2} \\ X_{C_1} &\rightarrow \neg X_{C_2} \end{aligned} \tag{5}$$

By means of the mechanisms from [23] we can transfer these logical constraints from their conjunctive normal form (CNF) into an algebraic form, which is needed to create the matrix B of the BOP normal form (1). Therefore we introduce a 0/1-variable x for every atomic formula X. Furthermore every clause of the CNF is transferred to an inequation, i.e. sum ≥ 1. In this sum, every literal X is replaced with its according 0/1-variable and every negative literal $\neg X$ is replaced with $(1 - x)$.

For our example we derive

$$\begin{aligned} (1 - x_{A_1}) + (1 - x_{A_2}) &\geq 1 \\ (1 - x_{B_1}) + (1 - x_{B_2}) &\geq 1 \\ (1 - x_{C_1}) + (1 - x_{C_2}) &\geq 1 \end{aligned} \tag{6}$$

which can be simplified to

$$\begin{aligned} -x_{A_1} - x_{A_2} &\geq -1 \\ -x_{B_1} - x_{B_2} &\geq -1 \\ -x_{C_1} - x_{C_2} &\geq -1 \end{aligned} \tag{7}$$

Next to the requirement of the existence of only one component version for each component in a system, we need to model constraints for the dependencies between the components, e.g. component version A_1 requires the services of component version B_1. In section 2 we sketched, how these dependencies can be derived automatically from the source code of components.

The dependencies can once again be easily expressed in propositional logic:

$$\begin{aligned} X_{A_1} &\rightarrow X_{B_1} \vee X_{B_2} \\ X_{A_2} &\rightarrow X_{B_2} \\ X_{B_1} &\rightarrow X_{C_1} \vee X_{C_2} \\ X_{B_2} &\rightarrow X_{C_1} \end{aligned} \tag{8}$$

Generally we derive expressions, in which we require the existence of different component versions as an alternative, and the coexistence of several other components.

The algebraic transformation of the example yields

$$\begin{aligned} (1 - x_{A_1}) + x_{B_1} + x_{B_2} &\geq 1 \\ (1 - x_{A_2}) + x_{B_2} &\geq 1 \\ (1 - x_{B_1}) + x_{C_1} + x_{C_2} &\geq 1 \\ (1 - x_{B_2}) + x_{C_1} &\geq 1 \end{aligned} \tag{9}$$

which can be simplified to

$$\begin{aligned} -x_{A_1} + x_{B_1} + x_{B_2} &\geq 0 \\ -x_{A_2} + x_{B_2} &\geq 0 \\ -x_{B_1} + x_{C_1} + x_{C_2} &\geq 0 \\ -x_{B_2} + x_{C_1} &\geq 0 \end{aligned} \tag{10}$$

Modelling the dependencies completes the model. We now have matrix A and vector b of the normal form of a Boolean Optimization Problem as defined in (1). For our example this leads to

$$A = \begin{pmatrix} -1 & -1 & & & & \\ & & -1 & -1 & & \\ & & & & -1 & -1 \\ -1 & & 1 & 1 & & \\ & -1 & & 1 & & \\ & & -1 & & 1 & 1 \\ & & & -1 & 1 & \end{pmatrix} \tag{11}$$

and

$$b = \begin{pmatrix} -1 \\ -1 \\ -1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{12}$$

4.3 Complexity

The specified constraints in propositional logic can be used to determine the complexity of the problem to find valid combinations of component versions. As all constraints have to be fulfilled simultaneously, we can form one single conclusion by superposing the constraints by means of conjunctions. For the example we derive:

$$\begin{aligned} &(\neg X_{A_1} \vee \neg X_{A_2}) \wedge (\neg X_{B_1} \vee \neg X_{B_2}) \wedge \\ &(\neg X_{C_1} \vee \neg X_{C_2}) \wedge (\neg X_{A_1} \vee X_{B_1} \vee X_{B_2}) \wedge \\ &(\neg X_{A_2} \vee X_{B_2}) \wedge (\neg X_{B_1} \vee X_{C_1} \vee X_{C_2}) \wedge \\ &(\neg X_{B_2} \vee X_{C_1}) \end{aligned} \tag{13}$$

The problem to decide if a given formula is satisfiable or not is called SAT. SAT is known to be an NP-complete problem [8]. We now show that our problem, i.e. finding a well composed system (**WCS**), is an NP-complete problem, too. To do so, we describe it more formally. In the following definition of **WCS** R is a set of component versions, C is a set of components, $C' \subseteq C$, I is a set of interfaces, IVO $\subseteq R \times C$ is a right-unique and left-complete relation, $ICT \subset I \times I$ is a relation on the interfaces, and S is a system as defined in section 2.

$$\begin{aligned} \mathbf{WCS} := \{(R, C, C', I, \text{IVO}, ICT) \mid \exists S \subseteq R.\Theta^S = 0 \\ \wedge (\forall c \in C'.\exists s \in S.(s, c) \in \text{IVO})\} \end{aligned} \tag{14}$$

The set C' determines the components that have to be in the system. In our example in section 3 this would be the front-end A. One component version of each element from the set C' has to be in S. If we do not demand such a set C', a well composed system can be found easily, since the empty system $S = \emptyset$ fulfills $\Theta^S = 0$.

WCS $\in$ NP is easy to see. Just guess a system S and test whether $\Theta^S = 0$ and $\forall c \in C'.\exists s \in S.(s, c) \in$ IVO holds. To show that WCS is NP-hard we sketch a polynomial reduction from SAT to WCS. Let ϕ be a formula in conjunctive normal form with m clauses $k_1, \ldots, k_m$ and n variables $x_1, \ldots, x_n$. Without loss of generality we can assume that in every clause k_i every variable x_j appears at most one time[5]. For such a formula ϕ we define $X_\phi := (R_\phi, C_\phi, C'_\phi, I_\phi, \text{IVO}_\phi, ICT_\phi)$ in the following way:

$$\begin{aligned} R_\phi &:= \{x_1^0, x_1^1, \ldots, x_n^0, x_n^1\} \cup \{a\} \\ C_\phi &:= \{A, x_1, \ldots, x_n\} \\ C'_\phi &:= \{A\} \\ I_\phi &:= \{k_1, \ldots, k_m\} \\ \text{IVO}_\phi &:= \{(a, A), (x_1^0, x_1), (x_1^1, x_1), \\ &\quad (x_2^0, x_2), (x_2^1, x_2), \ldots, (x_n^0, x_n), (x_n^1, x_n)\} \\ ICT_\phi &:= \{(k_i, k_i) \mid i \in \{1, \ldots, m\}\} \end{aligned}$$

Now we define the elements of R. It is $a = (a.n, \{k_1, \ldots, k_m\}, \emptyset)$, $x_i^0 = (x_i^0.n, \emptyset, \{k_j \mid x_i \text{ appears in } k_j \text{ as } \overline{x_i}\})$ for $i \in \{1, \ldots, n\}$, and finally $x_i^1 = (x_i^1.n, \emptyset, \{k_j \mid x_i \text{ appears in } k_j\})$ for $i \in \{1, \ldots, n\}$.

The idea of the reduction is the following. Formula ϕ is satisfied iff all clauses are satisfied. Hence, we define a component A with one component version a requesting all clauses. Furthermore each variable x_i is a component with two versions x_i^0 and x_i^1. Here x_i^0 means x_i has value 0 and x_i^1 means x_i has value 1. The component $x_i^j, j \in \{0, 1\}$, provides the clauses satisfied by x_i^j[6]. The definition of a system prevents that x_i^0 and x_i^1 are in the same system.

Now it is not too hard to see that $\phi \in$ SAT iff $X_\phi \in$ WCS. Since the size of X_ϕ is polynomial in m and n we have given a polynomial reduction from SAT to WCS.

5. Solving the problem

So far, we specified constraints which allow the evaluation of results for validity. The next step is to create a mechanism to find such results, thus valid system configurations, by searching through the state space of component versions. As the state space grows exponentially with the number of component versions, it is not sound to use a brute force approach and to just try out all possible combinations one after another. Hence we need a mechanism that cuts off uninteresting branches from the search space, reducing the search investment to a minimum. Branch-and-Bound (see [25]) is one representative of such a mechanism. The basic idea of Branch-and-Bound is to calculate an upper bound $\overline{z}$ (in case of a maximization problem) for an objective function to restrict the search to interesting branches. Such an upper bound is calculated in each search state. Together with a measurement for the not expanded branches, we can reduce the search space to a minimum.

In our case such a bound could be a maximum version sum of a valid system. During the search for valid combinations, only these branches are expanded, that will have at least a version sum of the so far calculated upper bound. If we find a solution with a greater version sum, we found a new upper bound, allowing us to remove other branches from the list.

By miscalculated bounds or a bad measurement for the not yet expanded branches, the probability rises to cut off states that may contain solutions or even the optimum.

5.1 Bounds

A smart way to calculate the required bounds is to drop the condition that the decision variables x_i are only allowed to contain 0/1 values and to solve the problem as a linear optimization problem.

[5] A variable x_i can appear in a clause negated and not negated, e.g. x_2 appears in $x_1 \vee \overline{x_2}$ negated and x_1 appears not negated. For the later case we simply say that x_1 appears in the clause.

[6] $x_1 = 1$ satisfies $x_1 \vee \overline{x_2}$ and $x_2 = 0$ also satisfies $x_1 \vee \overline{x_2}$

In order to accomplish that, we can utilize our model of a Boolean Optimization Problem.

If the model has a feasible solution, the constraints of the model can be interpreted as the border of an n-polyhedron with non empty volume. The search for an optimal solution can be considered as solving a geometric representation. An optimal solution can in principle be found on vertices of that polyhedron. Simplex algorithms (cf. [23]) move from vertex to vertex utilizing the edges of the polyhedron to find the optimal solution while interior point algorithms (cf. [14]) also use interior points of the polyhedron to find an optimal solution. Both methods guarantee to compute an optimal solution after a finite number of calculation steps.

From a theoretical point of view interior point algorithms are superior to simplex algorithms. Interior point algorithms (e.g. [10]) exist, which solve the linear optimization problem in worst-case polynomial time while for simplex algorithms it is an unsolved problem whether polynomial worst-case running time is possible[7]. In practice simplex algorithms perform very well and are successfully used in lots of optimization tools. Furthermore, simplex algorithms are better suited for Branch-and-Bound. Through the relaxation of the 0/1 constraint, we derive continuous values for the decision variables and an overestimation for the optimal version sum, which can be used as an upper bound in every search state. If the calculated bound in a certain search state is less than a previously calculated bound, then we do not have to expand this branch.

5.2 Search

In our example, the task was to find a well composed system containing component version A_2. As A_1 is not simply substitutable by A_2 in our current system configuration of A_1, B_1 and C_2, we now try to find an optimal system by applying our ideas.

We start the search with A_2 and set the current bound to zero. We pick an arbitrary component from the repository and calculate the version sum by relaxation. If we derive a solution, we know, that this branch contains a valid system configuration, but we only expand the branch if the calculated sum is equal or larger than our current bound. If the calculated sum is larger, then we have a new bound. We stop, when we have no more branches with an appropriate version sum to be expanded. Then we either have an optimal solution or no solution at all.

We can scale down the search tree, if we do not just pick arbitrary components from the repository, but sort the list in a specific way. Components that provide services to more components than others should be picked first as their early selection in the search process probably reduces the constraints to be evaluated by the linear relaxation for the rest of the search tree. This can be ensured by weighting the components by a factor named *potential benefit*, which can be calculated by simply counting the number of components that require this specific version.

From the list of available alternatives, component versions with higher version identifiers should be favored during search, which is ensured by sorting the Priority-Queue of component versions according to their version identifiers. This strategy is know as *best first search*.

Figure 4 shows the state space of the combinatorial problem as a tree, which will be traversed by depth-first search, described in [13]. Starting with A_2, every node represents the addition of one component version to the configuration. The branches of the tree that lead to a valid system configuration are colored gray. As every subnode of the tree represents independent combinatorial subproblems, the Branch-and-Bound algorithm is particularly suitable.

[7] (i) For a large class of simplex algorithms it is known that they have exponential worst-case complexity [12]. (ii) A randomized polynomial-time simplex algorithm is known [11].

In section 4.1 we assumed, that the current system consists of the versions A_1, B_1 and C_2. The mission is now to update component A_1 by a newer, bugfixed version A_2. The described mechanisms solve this problem and generate a system that downgrades component C_2 to C_1 as the only possible solution. We know of no approach that would be able to generate such a solution.

6. Evaluation

We are currently in the process of evaluating the results for real-world problems and projects from the open-source scene. These projects often lead to unplanned systems and, in component based architectures, unplanned component evolution.

We created a parser, which is able to derive fine grained specifications both of the provided and required interfaces of PEAR components, a component model for PHP-based applications, by static code analysis. In the project the source of all components is stored in Subversion repositories. Whenever a new version is committed, the parser generates the specification for this version.

The signature matcher introduced in section 2 is defined in first order logic and implemented in Prolog. With the help of SWI-Prolog, the matcher is integrated into our tool, written in Java. Hence, the required dependencies can be generated automatically.

These dependencies can then be utilized to first check the substitutability of the old version against the new one. If substitution is not applicable, we can use our tool, called *Componentor*, to analyze the reasons of this conflict and try to find valid combinations by using the sketched optimization mechanisms. If all this fails, the tool is able to generate proposals, how different components could be changed in order to ensure compatibility again.

Figure 5 shows a screenshot of our tool. On the left detailed descriptions about all available components, applications and the current system configuration are presented. By selecting components or their versions, a dependency graph is shown on the right side of the window. The buttons in the toolbar allow the calculation of the dependencies between all component versions, the validity check of the current system configuration by the calculation of Θ, and the simulation of an update.

We currently work on an Eclipse Plugin to create the required specifications for Java-projects as well, so that we can apply our mechanisms to component models like JavaBeans, EJB or other Java-based architectures, in which different parts can be updated independently.

To apply our methods, the components have to possess direct dependencies. The more generic software becomes, the less applicable are our methods. Dependencies that arise from dynamic instantiation and the usage of reflection are not seizable by static code analysis. Such dependencies could only be discovered by dynamic analyses, e.g. during unit testing.

7. Summary and Prospects

In the previous sections we sketched a mechanism to generate optimal system configurations consisting of component versions for the case that single or multiple components have to be updated. The system is specially designed for situations in which the new versions are not just simple substitutes of their predecessors. As we have shown, the problem of finding a well-configured combination of existing component versions is NP-hard, so optimization mechanisms must be used to derive acceptable results in average time. Thus, we modelled the problem as a Boolean Optimization Problem (BOP) and used Branch-and-Bound for cutting the search space.

The approach acquits of requirements like backward compatibility between different component versions but admits and supports unplanned component and system evolution. We are currently

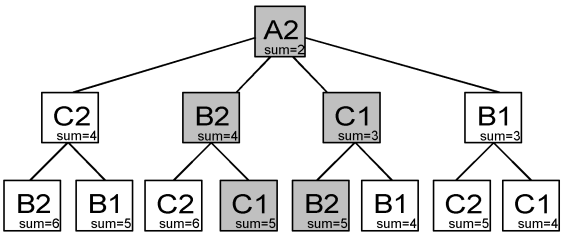

Figure 4. The state space of the search from the example. Sum denotes the version sum in the current search branch.

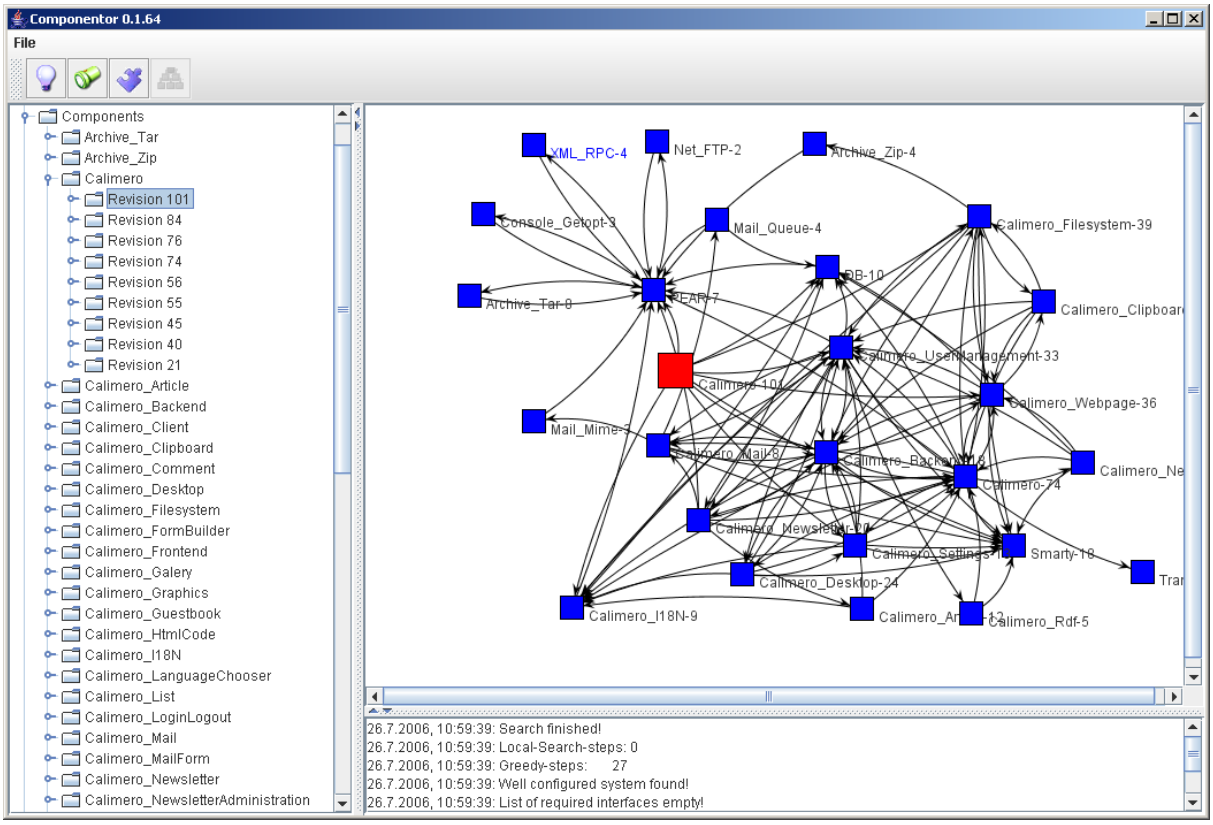

Figure 5. Screenshot from the Componentor update-tool to ensure safe component updates by checking substitutability and performing dynamic system synthesis

in process of validating our results at real-world projects from the open-source scene which are often subject to unplanned evolution.

We recognized, that in some cases it may become difficult to express the static dependencies between components needed to create our model. Especially in environments with interface inheritance over the boundaries of components, compatibility may depend on other components existing in a configuration. We therefore work on other approaches resting upon Simulated Annealing to find configurations for such environments.

Acknowledgments

We would like to thank our colleague Philipp Bouillon for his tireless services in improving this contribution regarding comprehensibility and linguistic quality.

References

[1] Edward C. Bailey. *Maximum RPM*. Sams, 1997.

[2] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. In *IEEE software*, pages 38–45, june 1999.

[3] Lars Borner et al. Fehlerhäufigkeiten in objektorientierten Systemen: Basisauswertung einer Online-Umfrage. Technical report, Arbeitsgruppe Software Systems Engineering, Institut für Informatik, Ruprecht-Karls-Universität Heidelberg, February 2006. Last visited: 03/2006.

[4] A. Brown and B. Barn. Enterprise-Scale CBD: Building complex Computer Systems from Components. In *Software Technology & Engineering Practice 9th International Conference*, page 82, 1999.

[5] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[6] Microsoft Corporation. Microsoft Security Bulletin MS04-028 - Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution (833987). Last visited: 02/2006.

[7] P. Devanbu. The ultimate reuse nightmare: Honey, i got the wrong dll. In *the 5th Symposium on Sofware Reuseability*, 178–180 1999.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[9] Ian Jackson and Christian Schwarz. Debian policy manual, 1998. Last visited: 12/2005.

[10] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984.

[11] Jonathan A. Kelmer and Daniel A. Spielman. A randomized polynomial-time simplex algorithm for linear programming. In *Symposium on Theory of Computing*, 2006. To be published.

[12] V. Klee and G. J. Minty. How good is the simplex algorithm. *Inequalities*, pages 159–175, 1972.

[13] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley Professional, 1998.

[14] Richard Kipp Martin. *Large Scale Linear and Integer Optimization - A Unified Approach*. Kluwer Academic Publishers, 1999.

[15] Steven Pratschner. Simplifying Deployment and Solving DLL Hell with the .NET Framework. Technical report, Microsoft Corporation, November 2001. Last visited: 04/2004.

[16] Dale Rogerson. *Inside COM - Microsofts Component Object Model*. Microsoft Press, Redmond, Washington, 1997.

[17] C. Romero. *Handbook of critical issues in goal programming*. Pergamon Press, Oxford.

[18] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2002.

[19] Marc J. Schniederjans. *Goal programming methodology and applications*. Kluwer publishers, Boston, 1995.

[20] Johannes Siedersleben. *Moderne Software Architektur*. Dpunkt Verlag, 2004. In German.

[21] Alexander Stuckenholz. Component evolution and versioning - state of the art. *SIGSOFT Softw. Eng. Notes*, 30(1):7, January 2005.

[22] Alexander Stuckenholz and Olaf Zwintzscher. Compatible component upgrades through smart component swapping. In Ralf Reusner, Judith Stafford, and Clemenz Szyperski, editors, *Architecting Systems with Trustworthy Components*, LNCS 3938. Springer-Verlag, 2006.

[23] Leena Suhl and Taieb Mellouli. *Optimierungssysteme*. Springer, 2005. (in German).

[24] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming - Second Edition*. Addison-Wesley, 2002.

[25] Laurence A. Wolsey. *Integer Programming*. Wiley-Interscience, 1998.